6-2011

Resizable, Scalable, Concurrent Hash Tables

Josh Triplett
Portland State University

Paul E. McKenney

Jonathan Walpole
Portland State University

Citation Details

"Resizable, Scalable, Concurrent Hash Tables" Josh Triplett, Paul E. McKenney, and Jonathan Walpole, in Proceedings of the USENIX Annual Technical Conference (USENIX ATC'11), Portland, Oregon, June 2011. Also available as PSU Computer Science Department Technical Report 11-01.

Resizable, Scalable, Concurrent Hash Tables

Josh Triplett
Portland State University
josh@joshtriplett.org

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

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

Abstract

We present algorithms for shrinking and expanding a hash table while allowing concurrent, wait-free, linearly scalable lookups. These resize algorithms allow the hash table to maintain constant-time performance as the number of entries grows, and reclaim memory as the number of entries decreases, without delaying or disrupting readers.

We implemented our algorithms in the Linux kernel, to test their performance and scalability. Benchmarks show lookup scalability improved 125x over reader-writer locking, and 56% over the current state-of-the-art for Linux, with no performance degradation for lookups during a resize.

To achieve this performance, this hash table implementation uses a new concurrent programming methodology known as *relativistic programming*. In particular, we make use of an existing synchronization primitive which waits for all current readers to finish, with little to no reader overhead; careful use of this primitive allows ordering of updates without read-side synchronization or memory barriers.

1 Introduction

Hash tables provide a useful data structure for many applications and operating systems, with various convenient properties, including constant average time for accesses and modifications [3, 8].

The performance and suitability of hash tables depend heavily on choosing an appropriate size for the table. Making a hash table too small will lead to excessively long hash chains and poor performance. Making a hash table too large will consume too much memory, reducing the memory available for other applications or performance-improving caches, and increasing hardware requirements.

Many systems and applications cannot know the proper size of the hash table in advance. Software designed for use on a wide range of system configurations with varying needs may not have the option of choosing a single hash table size suitable for all supported system configurations. Furthermore, the needs of a system may change at runtime due to numerous factors, and software must scale both up and down dynamically to meet these needs. Thus our hash tables must provide dynamic resizing.

Hash tables used in concurrent applications require some sort of synchronization to maintain internal consistency. In order for these concurrent applications to scale to many threads on many processors, the hash tables they use must scale as well.

Existing approaches to concurrent resizable hash tables primarily make use of mutual exclusion, in the form of locks. These approaches do not scale, due to contention for those locks. Alternative implementations exist, using non-blocking synchronization or transactions, but these techniques still require expensive synchronization operations, and still do not scale well. Running any of these hash-table implementations on additional processors does not provide a proportional increase in performance.

One solution for scalable concurrent hash tables comes in the form of Read-Copy Update (RCU) [16, 14, 10]. Read-Copy Update provides a synchronization mechanism for concurrent programs, with very low overhead for readers [11]. Thus, RCU works particularly well for data structures with significantly more reads than writes; this category includes many data structures commonly used in operating systems and applications.

Existing RCU-based hash tables use open chaining, with RCU-based linked lists for each hash bucket. These tables support insertion, removal, and lookup operations [11]. Furthermore, our previous work provided an algorithm to move hash items between hash buckets due to a change in the key [21, 20]; this operation makes RCU-based hash tables more broadly usable in place of those based on mutual exclusion.

Resizing a hash table based on mutual exclusion requires relatively little work: simply acquire the appropriate locks to exclude concurrent reads and writes, then move items to a new table. This approach works, but blocks all operations while a resize occurs.

However, existing RCU-based hash tables do not provide the ability to resize, because they cannot exclude readers, and thus they must cope with concurrent reads while resizing. This gap in available solutions forces programs to choose between scalable concurrent hash tables and resizable adaptive hash tables.

The Linux directory entry cache (dcache) uses a scalable concurrent hash table based on RCU [15, 9]; the dcache supports fast lookups concurrent with insertions and removals, but does not support resizing. As a result, the kernel must choose the dcache size at boot time, which it estimates based on the total amount of memory in the system. While functional, this commonly results in a dcache too large or small for the actual filesystem usage on the system. In addition, current server hardware and cloud platforms offer the ability to add storage and memory without rebooting; however, the kernel cannot adjust the dcache size to compensate for these new resources.

We present algorithms which allow resizing of an RCU-based hash table, without blocking concurrent lookups. Because lookups can occur at any time, we keep the hash table in a consistent state at all times, and never allow a lookup to spuriously miss an entry due to a concurrent resize operation. Furthermore, our resize algorithms avoid copying the individual hash-table nodes, allowing readers to maintain persistent references to table entries.

These algorithms make use of a new concurrent programming methodology known as *relativistic programming* (RP). This methodology aims to minimize the amount of synchronization required to perform operations, by allowing these operations to occur concurrently, without blocking them to enforce a system-wide serialization of memory operations. Relativistic programming builds on existing RCU synchronization primitives, and adds rules for using those primitives to enforce ordering of update operations.

Section 2 documents our new hash-table resize algorithms, and the corresponding read-side lookup operation. Section 3 describes the other hash-table algorithms implemented and tested for comparison. Section 4 provides the implementation and benchmark methodology. Section 5 presents and analyzes the benchmark results. Section 6 compares our algorithms to other related work. Section 7 discusses the relativistic programming methodology supporting this work.

2 Resize Algorithms

Any hash table requires a hash function, which maps entries to hash buckets based on their key. The same key will always hash to the same bucket; different keys will ideally hash to different buckets, but may map to the same bucket, requiring some kind of conflict resolution. The algorithms described here work with hash tables using *open chaining*, where each hash bucket has a linked list of entries whose keys hash to that bucket. As the number of entries in the hash table grows, the average depth of a bucket's list grows, and lookups become less efficient, necessitating a resize.

Resizing the table requires allocating a new region of memory for the new number of hash buckets, then linking all the nodes into the new buckets. To allow resizes to substitute the new hash table for the old, readers access a hash-table structure through a pointer; this structure includes the array of buckets and the size of the table.

Our hash-table resize algorithms synchronize with the corresponding lookup algorithm using existing relativistic programming primitives: a lightweight operation delineating the start and end of a reader, and an operation which blocks until all currently running readers have completed. Read-Copy Update (RCU) implementations provide such operations, with minimal overhead for the read-side primitives. However, any semantically equivalent implementation will work. The "wait for current readers" primitive need not prevent new readers from starting; it need only wait for all existing readers to complete.

With the two additions of relativistic reader delineation and a separate hash-table structure, a lookup in this table otherwise follows the standard algorithm for open-chain hash table lookups:

1. Begin an RP reader.
2. Snapshot the hash-table pointer, for later references to the size and buckets.
3. Hash the desired key, modulo the number of buckets.
4. Look up the corresponding hash bucket in the array.
5. Traverse the linked list, comparing each entry's key to the desired key.
6. If the current entry's key matches the desired key, the desired value appears in the same entry; use or return that value.[1]
7. End the RP reader.

[1] If the lookup algorithm needs to hold a reference to the entry after the RP reader ends, it must take any additional steps to protect that entry before ending the RP reader.

To avoid disrupting concurrent lookups, the resize algorithms can never allow a lookup to fail to find a node when traversing the hash table.

The proposed resize algorithms will map the nodes in the buckets of the old hash table to buckets of the new hash table. These algorithms rely on some key assumptions which constrain the relationships between the old and new hash buckets. Typically, a hash function used for hash tables will generate a value the size of a system word (for instance, 32 bits on a 32-bit machine, 64 bits on a 64-bit machine). Hash table algorithms will apply the hash function to an item's key to obtain a hash value, and then map the hash value to a hash bucket using an operation such as a modulus or bitwise and. Furthermore, we assume that any resize will occur via an integral factor. These assumptions provide two constraints:

- When shrinking the table, each bucket of the new table will contain all entries from multiple buckets of the old table.
- When growing the table, each bucket of the new table will contain entries from at most one bucket of the old table.

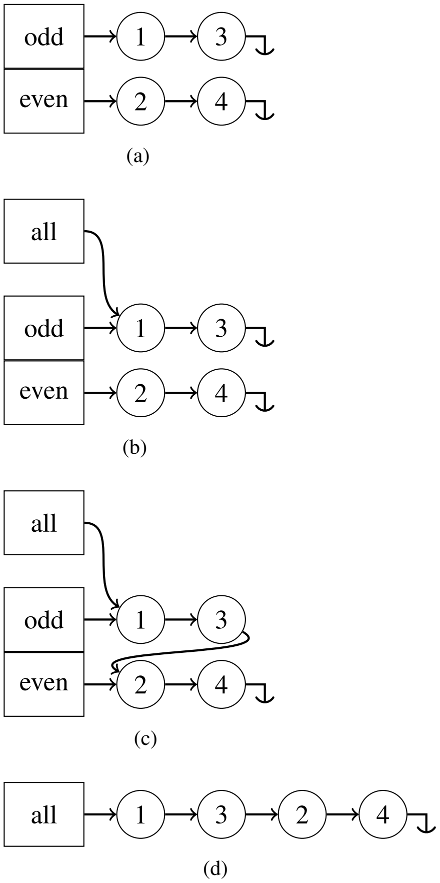
Based on the first of these constraints, we can shrink the table as follows:

1. Allocate the new, smaller table.
2. Link each bucket in the new table to the first bucket in the old table that contains entries which will hash to the new bucket.
3. Link the end of each such bucket to the beginning of the next such bucket; each new bucket will thus chain through as many old buckets as the resize factor.
4. Set the table size.
5. Publish the new, valid hash table.[2]
6. Wait for all current readers to finish. No new readers will have references to the old hash table.
7. Free the old hash table.

For an example of the shrink algorithm, see figure 1.

Based on the second constraint, we can safely expand the table:

1. Allocate the new, larger table.



(a)

(b)

(c)

(d)

Figure 1: Example of relativistic algorithm to shrink a hash table. Figure 1a shows the initial state of the hash table: it has two buckets, one containing odd numbers, and the other containing even numbers. The resize will transform this table into a hash table with one bucket containing all nodes. The resizer first allocates a new one-bucket table, and links the bucket to the first element in the odd bucket, resulting in figure 1b. The resizer then links the last element in the odd bucket to the first element in the even bucket, resulting in figure 1c. This makes the new table valid, so the resizer now publishes the new table, waits for current readers to finish to eliminate all references to the old table, then frees the old table, resulting in figure 1d.

[2] To *publish* a pointer update, we use the relativistic programming *publish* operation, as provided by RCU or equivalent. This operation first executes a write memory barrier to ensure that all updates to a structure become visible before the pointer to that structure. Readers then have an implicit ordering constraint due to the dependent read of the pointer contents after the pointer; on all modern CPU architectures, a dependent read does not require an explicit read memory barrier.

2. For each new bucket, search the corresponding old bucket for the first entry that hashes to the new bucket, and link the new bucket to that entry. Since all the entries which will end up in the new bucket appear in the same old bucket, this constructs an entirely valid new hash table, but with multiple buckets ”zipped” together into a single list.

3. Set the table size.

4. Publish the new table pointer. Lookups may now traverse the new table, but they will not benefit from any additional efficiency until later steps unzip the buckets.

5. Wait for all current readers to finish. All new readers will see the new table, and thus no references to the old table will remain.

6. For each bucket in the old table (each of which contains items from multiple buckets of the new table)

   6.1 Advance the pointer for that old bucket until it reaches a node that doesn't hash to the same bucket as the previous node. Call the previous node p.

   6.2 Find the subsequent node which does hash to the same bucket as node p, or NULL if no such node exists.

   6.3 Set p's next pointer to that subsequent node pointer, bypassing the nodes which do not hash to p's bucket.

7. Wait for all current readers to finish. New readers will see the changes made in this pass, so they won't miss a node during the next pass.

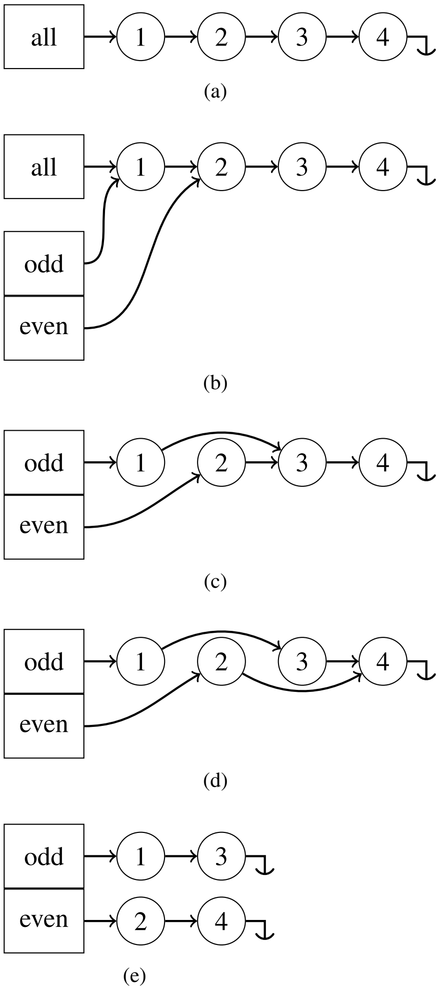
8. If any changes occurred in this pass, repeat from step 6.

9. Free the old hash table.

Notice that the wait in step 7 serves to order the update operations with respect to concurrent readers. Without it, a reader traversing one bucket and currently looking at an entry that hashes to another bucket could follow an updated pointer in that entry and fail to find later entries in its own bucket.

This algorithm requires auxiliary storage for a current entry in each old bucket. Conveniently, the old hash table provides an array of precisely the needed size to store this additional state. Reworking this algorithm to avoid the auxiliary storage would allow freeing the old table earlier, after step 5; this presents a subject for future development.

For an example of the hash-table expansion algorithm, including the error case avoided by waiting for current readers, see figure 2.



(a)

(b)

(c)

(d)

(e)

Figure 2: Example of relativistic algorithm to expand a hash table. Figure 2a shows the initial state of the hash table: it has one bucket, containing all of the entries. The resize will transform this table into a table with two buckets, one containing odd numbers and the other containing even numbers. The resizer first allocates a new two-bucket table, and points each bucket to the first entry in all that hashes to that bucket, resulting in figure 2b. This represents a valid hash table (with “zipped” buckets), so the resizer publishes this table, and waits for all current readers to finish to eliminate references to the old table, which it may then use as auxiliary storage to remember its place in each bucket. The resizer then begins unzipping the buckets, by pointing node 1 at node 3, bypassing the even node 2, resulting in figure 2c. Now, a reader which started traversing the odd bucket in 2b could still hold a reference to bucket 2, and changing 2 to point to 4 would cause that reader to miss node 3. Thus, the resizer first waits for all current readers to finish before pointing 2 to 4, resulting in figure 2d. Finally, the resizer waits once more for current readers before pointing 3 to null, resulting in the expanded hash table in figure 2e.

2.1 Handling Insertion and Removal

The algorithms so far support expanding or shrinking the hash table while handling concurrent lookups, but do not yet address the operations of insertion and removal. Existing RCU-based hash tables synchronize an insertion or removal operation with concurrent lookups via standard RCU linked-list operations on the appropriate bucket; multiple insertion and removal operations synchronize with each other using per-bucket mutual exclusion. Resizes, however, introduce an additional operation that modifies the hash table, and thus require synchronization with insertions and removals.

Herlihy and Shavit describe a common workload for hash tables as 90% lookups, 9% insertions, and 1% removals [6]. This ratio justifies our emphasis on fast concurrent lookups. For insertions and removals, we consider it sufficient to minimize performance degradation versus a non-resizable hash table, particularly with no concurrent resize running.

For the simplest correct implementation of insertion and removal, we can have these operations acquire and release a reader-writer lock for reading, and have the resize algorithm acquire the same lock for writing. However, resizes need to repeatedly wait for readers to finish, an operation that can incur long delays. Holding the write lock over these delays would block all updates to the hash table for extended periods of time. For sufficiently infrequent resizes this may prove acceptable, particularly since it should not result in excessive performance degradation when not running a concurrent resize. We would nonetheless prefer a solution at least as scalable as per-bucket locking, particularly if insertion and removal operations already need to do such locking to coordinate with each other.

To support per-bucket locking, we first need to resize the array of locks themselves, stored either as a standalone array or as a field of each bucket. For this initial memory resizing step the resizer may synchronize with updates using either a reader-writer lock as above (which the resizer need not hold across a wait for readers), or by temporarily acquiring all the per-bucket locks [6]. Given this means of temporarily locking out all updates, the resizer may then perform the initial step of the resize, resulting in a new valid hash table with a sufficient number of per-bucket locks.

After this initial step, our shrink algorithm has already produced a valid table with no unusual properties that can affect concurrent insertions or removals. As soon as the shrink algorithm has finished cross-linking a given bucket, it may drop the per-bucket lock for that bucket. Thus, insertions and removals may immediately take place on earlier buckets while the cross-linking steps complete on later buckets.

The expansion algorithm proves more difficult, as the insertion and removal operations must take extra care when operating on the "zipped" buckets. When performing a single unzip pass on a given set of buckets, the expansion algorithm must acquire the per-bucket locks for all buckets in that set. This alone proves sufficient to handle insertions, which may simply insert at the beginning of the appropriate new bucket without disrupting the next resize pass.

Removal, however, may occur at any point in a zipped bucket, including at the location of the resizer's current pointer into that bucket to mark its place for the next unzip pass. If a removal occurs with a table expansion in progress, the removal must check for a conflict with the resizer's current pointer into the bucket, and update that pointer if it points to the removed node. Given the relatively low frequency of removal versus lookup and insertion (1%, per Herlihy and Shavit) and the even lower frequency of resizes, we consider it acceptable to require this additional check in the removal algorithm.

2.2 Variation: Resizing In Place

The preceding descriptions of the resize algorithms assumed an out-of-place resize: allocate a new table, move all the nodes, free the old table. However, given a memory allocator which can resize existing allocations without moving them, we can adapt the resize algorithms to resize in place. This has two primary side effects: the resizer cannot count on the new table remaining private until published, and the buckets shared with the old table will remain initialized to the same values.

To shrink a hash table in place, we adapt the previous shrink algorithm to avoid disrupting current readers:

1. The smaller table will consist of a prefix of the current table, and the buckets in that prefix already point to the first of the lists that will appear in those buckets.

2. As before, concatenate all the buckets which contain entries that hash to the same bucket in the smaller table.

3. Wait for all current readers to finish. All new readers will see the concatenated buckets.

4. Set the table size to the new, smaller size.

5. Wait for all current readers to finish. No new readers will have references to the buckets beyond the common prefix.

6. Shrink the table's memory allocation.

To expand a hash table in place, we can make a similar adaptation to the expansion algorithm, by adding a single wait for current readers before setting the new size. However, the algorithm still requires auxiliary storage for the traversal state in the old buckets, equal to the size of the current table. Together with the newly expanded allocation, this makes in-place expansion require the same amount of memory as out-of-place expansion.

2.3 Variation: Keeping Buckets Sorted

Typically, a hash table implementation will not enforce any ordering on the items within a hash bucket. This allows insertions to take constant time even if a bucket contains many items. However, if we keep the items in a bucket sorted carefully, modulo-based hashing will keep all the items destined for a given new bucket together in the same old bucket. This allows a resize increasing the size of the table to make only as many passes as the resize factor, minimizing the number of waits for current readers. This approach optimizes resizes significantly, at the expense of slowing down insertions into large buckets.

Furthermore, an application may find sorted buckets useful for other reasons, such as optimizing failed lookups. Sorted buckets do not provide an algorithmic improvement for lookups, nor can they do anything to accelerate successful lookups; however, sorted buckets do allow failed lookups to terminate sooner, providing a constant-factor improvement for failed lookups.

Our hash-table expansion algorithm already performs a stable partition of the entries in a bucket, preserving the relative order of entries within each of the subsets that move to the buckets of the new table. The shrink algorithm, however, simply concatenates a set of old buckets into a single new bucket. To make the shrink algorithm preserve the sorting order for concurrent readers, we must use a careful sort order based on the hash key. Ori Shalev and Nir Shavit presented such a sorting mechanism in their "split-ordered list" proposal [19, 6].

We do not pursue this variation further in this paper, but we do consider it a potentially productive avenue for future investigation.

3 Other Resize Algorithms

To evaluate the performance of our hash-table resize algorithm, we compared it against two other resize algorithms.

First, as a baseline, we implemented a simple resizable hash table based on reader-writer locking. In this implementation, lookups acquired a reader-writer lock for reading, to lock out concurrent resizes. Resizes acquired the reader-writer lock for writing, to lock out concurrent lookups. With lookups excluded, the resizer could simply allocate the new table, move all entries from the old table to the new, publish the new table, and free the old table. We do not expect this implementation to scale well, but it represents the best-known method based on mutual exclusion, and we included it to provide a baseline for comparison.

For a more competitive comparison, we turned to Nick Piggin's "Dynamic Dynamic Data Structures" (DDDS) [18]. DDDS provides a generic algorithm to safely move nodes between any two data structures, given only the standard insertion, removal, and lookup operations for those structures. In particular, DDDS provides another method for resizing an RCU-protected hash table without outright blocking concurrent lookups (though it can delay them).

The DDDS algorithm uses two technologies to synchronize between resizes and lookups: RCU to detect when readers have finished with the old data structure, and a Linux construct called a *sequence counter* or *seqcount* to detect if a lookup has raced with a rename. A seqcount employs a counter incremented before and after moving each entry; the reader can use that counter (together with an appropriate read memory barrier) to check for a resize step running concurrently with any part of the read.

The DDDS lookup occurs entirely within an RCU read-side critical section. Within that critical section, DDDS first checks for the presence of an old hash table, which indicates a concurrent resize. If present, the lookup proceeds via the concurrent-resize slowpath; otherwise, the lookup uses a fastpath. The DDDS fastpath just performs a lookup within the current hash table. The slowpath uses a sequence counter to check for a race with a resize, then performs a lookup first in the current hash table and then in the old table. The slowpath returns the result of the first successful lookup, or loops if both lookups fail and the sequence counter indicates a race with a resize. Note that the potentially unbounded number of retries makes DDDS lookups non-wait-free, and could theoretically lead to a livelock, though in practice resizes do not occur frequently enough for a livelock to arise.

We expect DDDS to perform fairly competitively versus our resize algorithms. However, the DDDS lookup incurs more overhead than our algorithms, due to the additional conditionals, the secondary table lookup, the expensive read memory barrier in the sequence counter, and the potential retries with a concurrent resize. Thus, we expect our algorithms to outperform DDDS significantly when running a concurrent resize, and slightly even without a concurrent resize.

4 Benchmark Methodology

To compare the performance and scalability of our algorithms to the alternatives, we created a test harness and benchmarking framework for resizable hash-table implementations. We chose to implement this framework as a Linux kernel module, `rcuhashbash-resize`. The Linux kernel already includes a scalable implementation of RCU, locking primitives, and linked list primitives. Furthermore, we created our hash-table resize algorithms with specific use cases of the Linux kernel in mind, such as the directory entry cache. This made the Linux kernel an ideal development and benchmarking environment.

The `rcuhashbash-resize` framework provides a common structure for hash-tables, based on the Linux hlist primitives, a doubly-linked list with a single head pointer. On top of this common base, `rcuhashbash-resize` includes the lookup and resize functions for the three resizable hash-table implementations: our relativistic resizable hash table, DDDS, and the simple rwlock-based implementation.

The current Linux memory allocator supports shrinking memory allocations in-place, but does not support growing in-place. Thus, we implemented the in-place variation of our shrink algorithm, and the copying implementation of our expansion algorithm.

`rcuhashbash-resize` accepts the following configuration parameters:

- The name of the hash-table implementation to test.
- An initial and alternate hash table size, specified as a power of two.
- The number of entries to appear in the table.
- The number of reader threads to run.
- Whether to run a resize thread.

`rcuhashbash-resize` starts by creating a hash table with the specified number of buckets, and adds entries to it containing integer values from 0 to the specified upper bound. It then starts the reader threads and optional resize thread, which record statistics in thread-local variables to avoid the need for additional synchronization. When the test completes, `rcuhashbash-resize` stops all threads, sums their recorded statistics, and presents the results via the kernel message buffer.

The reader threads choose a random value from the range of values present in the table, look up that value, and record a hit or miss. Since the readers only look up entries that should exist in the table, any miss would indicate a test failure.

The resize thread continuously resizes the hash table from the initial size to the alternate size and back. While continuous resizes do not necessarily reflect a common usage pattern for a hash table, they will most noticeably demonstrate the impact of resizes on concurrent lookups. In practice, most hash tables will choose growth factors and hysteresis to avoid frequent resizes, but such a workload would not allow accurate measurement of the impact of resizing on lookups. We consider a continuous resize a harsh benchmark, but one which a scalable concurrent implementation should handle reasonably. Furthermore, we can perform separate benchmark runs to evaluate the cost of the lookup in the absence of resizes.

The benchmark runs in this paper all used a hash table with 2^{16} entries. For each of the three implementations, we collected statistics for three cases: no resizing and 2^{13} buckets, no resizing and 2^{14} buckets, and continuous resizing between 2^{13} and 2^{14} buckets. We expect lookups to take less time in a table with more buckets, and thus if the resize algorithms have minimal impact on lookup performance, we would expect to see the number of lookups with a concurrent resizer fall between the no-resize cases with the smaller and larger tables.

For each set of test parameters, we performed 10 benchmark runs of 10 seconds each, and averaged the results.

Our test system had two Intel "Westmere" Xeon DP processors at 2.4GHz, each of which had 6 hardware cores of two logical threads each, for a total of 24 hardware-supported threads (henceforth referred to as "CPUs"). To observe scalability, we ran each benchmark with 1, 2, 4, 8, and 16 concurrent reader threads, with and without an additional resize thread. In all cases, we ran fewer threads than the hardware supported, thus minimizing the need to pass through the scheduler and allowing free CPUs to soak up any unremovable OS background noise. (We do however expect that performance may behave somewhat less than linearly when passing 12 threads, as that matches the number of hardware cores.)

All of our tests occurred on a Linux 2.6.37 kernel using the default configuration (`make defconfig`). We targeted the 64-bit x86-64 architecture, using the hierarchical RCU implementation, and no involuntary preemption.

5 Benchmark Results

To evaluate the baseline reader performance in the absence of resizes, we first compare the lookups per second for all the implementations with a fixed table size of 8192 buckets; figure 3 shows this comparison. As previously predicted, our relativistic programming (RP) implementation and DDDS remain very competitive when not concurrently resizing, though as the number of con-

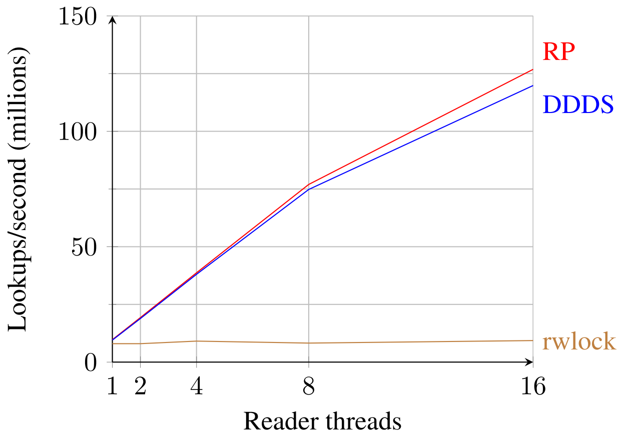


Figure 3: Lookups/second by number of reader threads for each of the three implementations, with a fixed hash-table size of 8k buckets, and no concurrent resizes.

current readers increases, our implementation's performance pulls ahead of DDDS slightly. Reader-writer locking does not scale at all. In this test case, the reader-writer lock never gets acquired for writing, yet the overhead of the read acquisition prevents any reader parallelism.

We observe the expected deviation from linear growth for 16 readers, likely due to passing the limit of 12 hardware cores. In particular, notice that the performance for 16 threads appears approximately 50% more than that for 8, which agrees with the expected linear increase for fully utilizing 12 hardware cores rather than 8.

Figure 4 compares the lookups per second for our implementation and DDDS in the face of concurrent resizes. (We omit rwlock from this figure, because it would vanish against the horizontal axis.) With a resizer running, our lookup rate scales better than DDDS, with its lead growing as the number of reader threads increases. DDDS has sub-linear performance, while our lookup rate improves linearly with reader threads.

To more precisely evaluate the impact of resizing on lookup performance for each implementation, we compare the lookups per second when resizing to the no-resize cases for the larger and smaller table size. Figure 5 shows the results of this comparison for our implementation. The lookup rate with a concurrent resize falls between the no-resize runs for the two table sizes that the resizer toggles between. This suggests that our resize algorithms add little to no overhead to concurrent lookups.

Figure 6 shows the same comparison for the DDDS resize algorithm. In this case, the lookup rate with a resizer running falls below the lower bound of the smaller hash table. This suggests that the DDDS resizer adds significant overhead to concurrent lookups, as previously hypothesized.

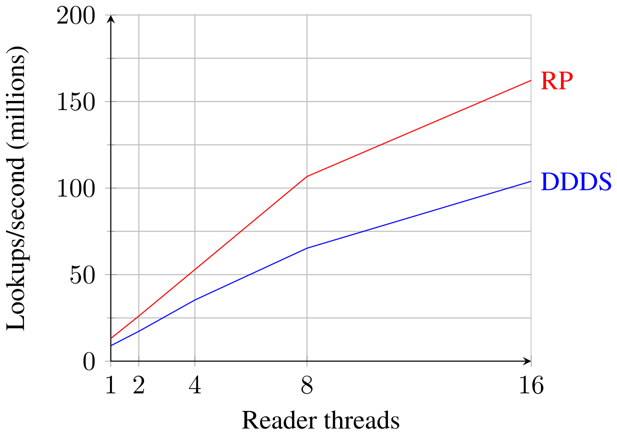


Figure 4: Lookups/second by number of reader threads for our RP-based implementation versus DDDS, with a concurrent resize thread continuously resizing the hash-table between 8k and 16k buckets.

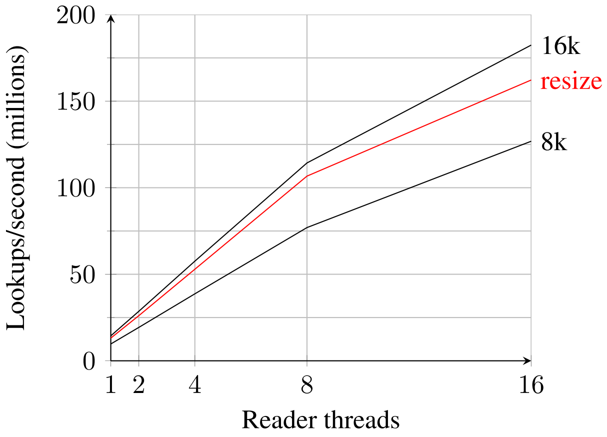


Figure 5: Lookups/second by number of reader threads for our resize algorithms. "8k" and "16k" indicate fixed hash-table sizes in buckets; "resize" indicates continuous resize between the two sizes.

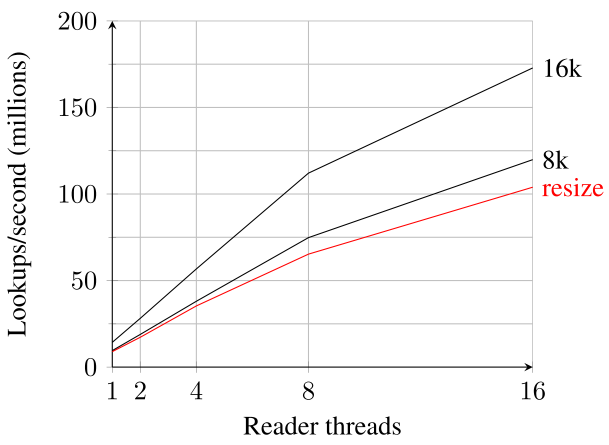


Figure 6: Lookups/second by number of reader threads for the DDDS resize algorithm. “8k” and “16k” indicate fixed hash-table sizes in buckets; “resize” indicates continuous resize between the two sizes.

Finally, figure 7 shows the same comparison for the rwlock-based implementation. With a resizer running, the rwlock-based lookups suffer greatly, falling initially by two orders of magnitude with a single reader, and struggling back up to only one order of magnitude down at the 16-reader mark.

5.1 Benchmark Summary

Our relativistic resizable hash-table provides linearly scalable lookup performance, surpassing DDDS by a widening margin of up to 56% with 16 reader threads. Both implementations vastly dwarf reader-writer locks, with ours providing a 125x improvement with 16 readers. Furthermore, our resize algorithms minimize the impact of concurrent resizing on lookup performance, as demonstrated through the comparison with fixed-size hash tables.

6 Related Work

Our hash-table resize algorithms operate on hash tables using open-hashing with per-bucket chaining, and do not function on closed-hashing tables that use re-probing. Closed hash tables store entries inline in the array rather than using indirection, to minimize lookup cost. However, this prevents the use of indirection to allow existing entries to migrate to the new hash table, forcing a copy of each entry, and breaking any persistent references. Furthermore, closed-address tables require more frequent resizing, as they do not gracefully degrade in performance when overloaded, but rather become exponentially more expensive and then stop working entirely.

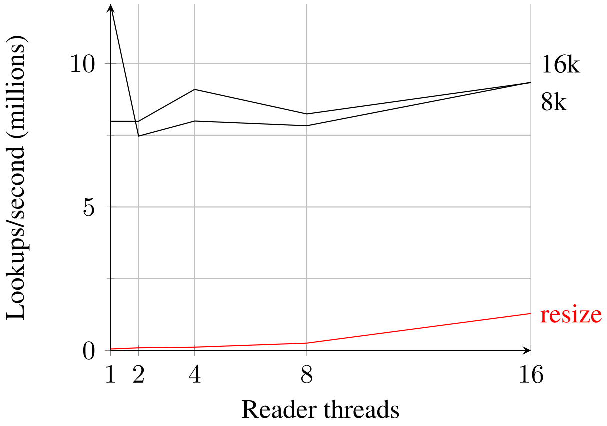


Figure 7: Lookups/second by number of reader threads for the rwlock-based implementation. “8k” and “16k” indicate fixed hash-table sizes in buckets; “resize” indicates continuous resize between the two sizes.

Depending on the implementation, deletions from the table may not make the table any emptier, as the entries must remain as “tombstones” to preserve reader probing behavior.

Various authors have proposed resizable hash tables using closed hashing. Maurice Herlihy and Nir Shavit documented a resizable closed-hashing table based on cuckoo hashing and striped lock arrays [6]. Gao, Groote, and Hesselink proposed a lock-free hash table using closed hashing [5]; their approach relied heavily on atomic operations and on helping concurrent operations complete.

Cliff Click presented a scalable lock-free resizable hash for Java [2]; this hash avoids any synchronization operations for readers, by leaving the ordering of memory operations entirely unspecified and reasoning about all possible resulting memory states and transitions. This assumption holds for the intended application to hash-based caches, in which spurious lookup failures only led to suboptimal performance rather than incorrect behavior. However, to support resizing, readers must probe an alternate hash table; furthermore, as with most closed-hashing approaches, the resizer must copy all entries.

Other approaches to resizable hash tables include that of Ori Shalev and Nir Shavit, who proposed a “split-ordered list” structure consisting of a single linked list with hash buckets pointing to intermediate list nodes [19, 6]. This structure allows resizing by adding or removing buckets, dividing or joining the existing buckets respectively. This approach requires maintaining the linked list in a carefully sorted order based on the hash key, as with the variation of our algorithms proposed in section 2.3. Like that variation, this approach incurs additional costs on insertion to maintain the sort order.

Maged Michael implemented a lock-free resizable hash table based on compare and swap (CAS) [17]. This implementation also maintains sorted buckets. It relies on retries like DDDS, but lookups must retry if a concurrent modification of any kind occurs, not just if a concurrent resize occurs.

Herbert Xu implemented a resizable multi-hash-table structure based on RCU, in which every entry contains two sets of linked-list pointers so it can appear in two hash tables simultaneously [22]. Together with a global version number for the structure, this allows readers to effectively snapshot all links in the hash table simultaneously. However, this approach drastically increases memory usage.

Our previous work developed a relativistic algorithm for moving a hash-table entry from one bucket to another atomically [21, 20]. This algorithm introduced the notion of cross-linking hash buckets to make entries in multiple buckets simultaneously. However, this move algorithm required changing the hash key, and potentially copying the entry.

Paul McKenney originally proposed the elimination of read memory barriers by introducing a write memory barrier that forced a barrier on all CPUs via interprocessor interrupts [12]. McKenney later used the RCU "wait for current readers to finish" operation as a write memory barrier with batching in his work on Sleepable Read-Copy Update (SRCU) [13]. Philip Howard further refined this approach in his work on relativistic red-black trees [7].

While we chose to implement our benchmarking framework `rcuhashbash-resize` as a Linux kernel module, several portable RCU implementations exist outside the Linux kernel. Mathieu Desnoyers reimplemented RCU as a POSIX userspace library for use with pthreads, with no Linux-specific code outside of optional optimizations [4].

7 Relativistic Programming Methodology

Our proposed hash-table resize algorithms demonstrate the use of the "wait for current readers to finish" operation to order update operations. In our algorithms, this operation functions not merely as a write memory barrier, but as a means of flushing existing readers from a structure when their current position could otherwise cause them to see writes out of order. Figure 2 provided a specific example of this, in which a reader has already navigated past the location of an earlier write, but would subsequently encounter a later write if the writer did not first wait for such readers to finish.

We have developed a full methodology for ordering writes to any acyclic data structure while allowing concurrent readers, based on the order in which readers traverse a data structure. This methodology allows writers to consider only the effect of any prefix of their writes, rather than any possible subset of those writes. This proves equivalent to allowing a reader to perform a full traversal of the data structure between any two write operations, but not overlapping any write operation. This methodology forms the foundation of our work on *relativistic programming*.

Relativistic readers traversing a data structure have a current position, or *read cursor*. Writes to a data structure also have a position relative to read cursors: some read cursors will subsequently pass through that write, while others have already passed that point. In an acyclic data structure, readers will start their read cursors at designated entry points, and advance their read cursors through the structure until they find what they needed to read or reach the end of their path.

When a writer performs two writes to the data structure, it needs to order those writes with respect to any potential read cursors that may observe them. These writes will either occur in the same direction as reader traversals (with the second write later than the first), or in the opposite direction (with the second write earlier than the first). If the second write occurs later, read cursors between the two writes may observe the second write and not the first; thus, the writer must wait for readers to finish before performing the second write. However, if the second write occurs earlier in the structure, no read cursor may observe the second write and subsequently fail to observe the first write in the same pass (if it reaches the location of the first); thus, the writer need only use the relativistic *publish* operation, which uses a simple write memory barrier.

"Laws of Order" [1] presents a set of constraints on concurrent algorithms, such that any algorithm meeting those constraints must necessarily use expensive synchronization instructions. In particular, these constraints include *strong non-commutativity*: multiple operations whose order affects the results of both. Our relativistic programming methodology allows readers to run without synchronization instructions, because at a minimum those readers do not execute strongly non-commutative operations: reordering a read and a write cannot affect the results of the write.

We originally developed a more complex hash-table resize operation, which required readers to perform lookups in a secondary hash table if the primary lookup failed; this approach mirrored that of the DDDS lookup slowpath. Our work on the RP methodology motivated the simplified version that now appears in this paper. We plan to use the same methodology to develop algorithms for additional data structures not previously supported by RCU.

As an immediate example, the RP methodology allows a significantly simplified variation of our previous hash-table move algorithm [21, 20]. This variation will no longer need to copy the moved entry and remove the original, a limitation which breaks persistent references, and which made the original move algorithm unsuitable for use in the Linux dcache.

8 Availability

The authors have published the code supporting this paper as Free and Open Source Software under the GNU General Public License. See the repository at `http://git.kernel.org/?p=linux/kernel/git/josh/rcuhashbash.git` for details.

9 Acknowledgments

Thanks to Nick Piggin, Linux kernel hacker and implementer of the DDDS algorithm, for reviewing our implementation of DDDS to ensure that it fairly represents his work. Thanks to Intel for access to the 24-way system used for benchmarking. Thanks to Jamey Sharp and Phil Howard for their review, feedback, and advice.

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

References

[1] ATTIYA, H., GUERRAOUI, R., HENDLER, D., KUZNETSOV, P., MICHAEL, M. M., AND VECHEV, M. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated. In *Proceedings of the ACM POPL'11* (2011).

[2] CLICK, C. Lock-free hash table. In *JavaOne Conference* (2007).

[3] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, second ed. MIT Press, 2001, ch. Chapter 11: Hash Tables.

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

[5] GAO, H., GROOTE, J. F., AND HESSELINK, W. H. Lock-free dynamic hash tables with open addressing. *Distributed Computing 18*, 1 (July 2005).

[6] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008, ch. Chapter 13: Concurrent Hashing and Natural Parallelism.

[7] HOWARD, P. W., AND WALPOLE, J. Relativistic red-black trees. Tech. Rep. 1006, Portland State University, 2011. `http://www.cs.pdx.edu/pdfs/tr1006.pdf`.

[8] KNUTH, D. *The Art of Computer Programming*, second ed. Addison-Wesley, 1998, ch. Section 6.4: Hashing.

[9] LINDER, H., SARMA, D., AND SONI, M. Scalability of the directory entry cache. In *Ottawa Linux Symposium* (June 2002), pp. 289–300.

[10] 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.

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

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

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

[14] MCKENNEY, P. E., SARMA, D., ARCANGELI, A., KLEEN, A., KRIEGER, O., AND RUSSELL, R. Read-copy update. In *Ottawa Linux Symposium* (June 2002), pp. 338–367.

[15] MCKENNEY, P. E., SARMA, D., AND SONI, M. Scaling dcache with RCU. *Linux Journal 2004*, 117 (2004).

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

[17] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures* (2002), SPAA '02, pp. 73–82.

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

[19] SHALEV, O., AND SHAVIT, N. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM 53* (May 2006), 379–405.

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

[21] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Scalable concurrent hash tables via relativistic programming. *ACM Operating Systems Review 44*, 3 (July 2010).

[22] XU, H. bridge: Add core IGMP snooping support. Linux netdev mailing list. http://news.gmane.org/find-root.php?message_id=<E1NlbuT-00021C-0b@gondolin.me.apana.org.au>, February 2010.