Resizable, 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

We present algorithms for shrinking and expanding a hash table while allowing concurrent, wait-free, linearly scalable lookups. These resize algorithms allow Read-Copy Update (RCU) hash tables 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 call the resulting data structure a *relativistic hash table*.

Benchmarks of relativistic hash tables in the Linux kernel show that lookup scalability during resize improves 125x over reader-writer locking, and 56% over Linux's current state of the art. Relativistic hash lookups experience no performance degradation during a resize. Applying this algorithm to memcached removes a scalability limit for get requests, allowing memcached to scale linearly and service up to 46% more requests per second.

Relativistic hash tables demonstrate the promise of a new concurrent programming methodology known as *relativistic programming*. Relativistic programming makes novel use of existing RCU synchronization primitives, namely the *wait-for-readers* operation that waits for unfinished readers to complete. This operation, conventionally used to handle reclamation, here allows ordering of updates without read-side synchronization or memory barriers.

1 Introduction

Hash tables offer applications and operating systems many convenient properties, including constant average time for accesses and modifications [3, 10]. Hash tables used in concurrent applications require some sort of synchronization to maintain internal consistency. Frequently accessed hash tables will become application bottlenecks unless this synchronization scales to many threads on many processors.

Existing concurrent 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 many of 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) [18, 16, 12]. Read-Copy Update provides a synchronization mechanism for concurrent programs, with very low overhead for readers [13]. 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, such as read-mostly hash tables.

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 [13]. Our previous work introduced an algorithm to move hash items between hash buckets due to a change in the key [24, 23], making RCU-based hash tables even more broadly usable.

Unfortunately, RCU-based hash tables still have a major deficiency: they do not support *dynamic resizing*.

The performance of a hash table depends heavily on the number of hash buckets. 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, increasing hardware requirements or reducing the memory available for other applications or performance-improving caches. Many users of hash tables cannot know the proper size of a hash table in advance, since no fixed size suits all system configurations and workloads, and the system's needs may change at runtime. Such systems require a hash table that supports dynamic resizing.

Resizing a concurrent 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. However, RCU-based hash tables *cannot exclude readers*. This property proves critical to RCU's scalability and performance, since excluding readers would require expensive read-side synchronization. Thus, any RCU-based hash-table resize algorithm must cope with concurrent reads while resizing.

Solving this problem without reducing read performance has seemed intractable. Existing RCU-based scal-

able concurrent hash tables in the Linux kernel, such as the directory-entry cache (dcache) [17, 11], do not support resizing; they allocate a fixed-size table at boot time based on system heuristics such as available memory. Nick Piggin proposed a resizing algorithm for RCU-based hash tables, known as "Dynamic Dynamic Data Structures" (DDDS) [21], but this algorithm slows common-case lookups by requiring them to check multiple hash tables, and this slowdown increases significantly during a resize.

As our primary contribution, we present the first algorithm for resizing an RCU-based hash table without blocking or slowing concurrent lookups. Because lookups can occur at any time, we keep our *relativistic* 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.

A key insight made our relativistic hash table possible. We use an existing RCU synchronization primitive, the *wait-for-readers* operation, to control which versions of the hash-table data structure concurrent readers can observe. This use of wait-for-readers generalizes and subsumes its original application of safely managing memory reclamation. This general-purpose ordering primitive forms the basis of a new concurrent programming methodology, which we call *relativistic programming* (RP). Relativistic programming enables scalable, high-performance data structures previously considered intractable for RCU.

We use the phrase *relativistic programming* by analogy with relativity, in which observers can disagree on the order of causally unrelated events. Relativistic programming aims to minimize synchronization, by allowing reader operations to occur concurrently with writers; writers may never block readers to enforce a system-wide serialization of memory operations. Inevitably, then, independent readers can disagree on the order of unrelated writer operations, such as the insertion order of two items into separate hash-table chains; however, writers can still synchronize to preserve the order of causally related operations. Whereas concurrent programming methodologies such as transactional memory always preserve the ordering of even unrelated writes—at significant cost to performance and scalability, since readers must use synchronization to support blocking or retries—relativistic programming provides the means to program even complex, whole-data-structure operations such as resizing with excellent performance and scalability.

Section 2 compares our algorithms to other related work. Section 3 provides an introduction to RCU and to the relativistic programming techniques supporting this work. Section 4 documents our new hash-table resize algorithms, and the corresponding lookup operation. Section 5 describes the other hash-table implementations we tested for comparison. Section 6 discusses the implementation and benchmarking of our relativistic hash-table algorithm, including both microbenchmarks and real-world benchmarks. Section 7 presents and analyzes the benchmark results. Section 8 discusses the future of the relativistic programming methodology supporting this work.

2 Related Work

Relativistic hash tables use the RCU wait-for-readers operation to enforce the ordering and visibility of write operations, without requiring synchronization operations in the reader. This novel use of wait-for-readers evolved through a series of increasingly sophisticated write-side barriers. Paul McKenney originally proposed the elimination of read memory barriers by introducing a new write memory barrier primitive that forced a barrier on all CPUs via inter-processor interrupts [14]. McKenney's later work on Sleepable Read-Copy Update (SRCU) [15] used the RCU wait-for-readers operation to manage the order in which write operations became visible to readers, providing the first example of using wait-for-readers to order non-reclamation operations; this use served the same function as a write memory barrier, but without requiring a corresponding read memory barrier in every RCU reader. Philip Howard further refined this approach in his work on relativistic red-black trees [9], using the wait-for-readers operation to order the visibility of tree rotation and balancing operations and prevent readers from observing inconsistent states. Howard's work used wait-for-readers as a stronger barrier than a write memory barrier, enforcing the order of write operations regardless of the order a reader encounters them in the data structure. Relativistic programming builds on this stronger barrier.

Relativistic and RCU-based data structures typically use mutual exclusion to synchronize between writers. Philip Howard and Jonathan Walpole [8] demonstrated an alternative approach, combining relativistic readers with software transactional memory (STM) writers, and integrating the wait-for-readers operation into the transaction commit. This approach provides scalable high-performance relativistic readers, while also allowing scalable writers within the limits of STM. Relativistic transactions could substantially simplify relativistic hash-table writers compared to fine-grained locking, while still providing good scalability.

Prior attempts to build resizable RCU hash tables have arisen from the limitations of fixed-size RCU hash tables in the Linux kernel. Nick Piggin's DDDS [21] supports hash-table resizes, but DDDS slows down all lookups by

requiring checks for concurrent resizes, and furthermore requires that lookups during resizes examine both the old and the new structures; relativistic hash tables do neither. We discuss DDDS further in section 5. Herbert Xu implemented a resizable multi-hash-table structure based on RCU, in which every hash-table entry contains two sets of linked-list pointers so it can appear in the old and new hash tables simultaneously [25]. 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 and cache footprint.

Various authors [7, 5, 19, 2] have proposed resizable concurrent hash tables. Unlike relativistic hash tables, these algorithms require expensive synchronization operations on reads, such as locks, atomic instructions, or memory barriers. Furthermore, like DDDS, several of these algorithms require retries on failure.

Maurice Herlihy and Nir Shavit documented numerous concurrent hash tables, including both open-chained and closed tables [7]; all of these require expensive synchronization, and some require retries. Gao, Groote, and Hesselink proposed a lock-free hash table using closed hashing [5]; their approach relies on atomic operations and on helping concurrent operations complete.

Maged Michael implemented a lock-free hash table based on compare and swap (CAS) [19], though he did not propose a resize algorithm. Michael's table lookups avoid most expensive synchronization operations in the common case (with the exception of read barriers), but must retry on any concurrent modification. To support safe memory reclamation, Michael uses hazard pointers [20], which provide a wait-for-readers operation similar to that of RCU; hazard pointers can reduce wait-for-readers latency, but impose higher reader cost [6].

Relativistic hash tables use open hashing with per-bucket chaining. Closed hash tables, which store entries inline in the array, can offer smaller lookup cost and better cache behavior, but force copies on resize. Closed tables also require more frequent resizing, as they do not gracefully degrade in performance when overloaded, but rather become pathologically more expensive and then stop working entirely. Depending on the implementation, removals from the table may not make the table any emptier, as the entries must remain as "tombstones" to preserve reader probing behavior.

Cliff Click presented a scalable lock-free resizable hash for Java based on closed hashing [2]; this hash avoids most synchronization operations for readers and writers by leaving the ordering of memory operations entirely unspecified and reasoning about all possible resulting memory states and transitions. (Readers require a read memory barrier but no other synchronization. Writers require a CAS but not a write memory barrier.) Click's use of state-based reasoning to avoid ordering provides an interesting and potentially higher-performance alternative to the causal-order enforcement in relativistic writers. In contrast with relativistic hash tables, but like DDDS, Click's hash-table readers must probe alternate hash tables during resizing.

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 [22, 7]. This structure allows resizing by adding or removing buckets, splitting or joining the existing buckets respectively. This approach keeps the underlying linked list in a novel sort order based on the hash key, as with the variation of our algorithms proposed in section 4.3, to allow splitting or joining buckets without reordering. Split-ordered lists seem highly amenable to a simple relativistic implementation, making the lookups scalable and synchronization-free while preserving the lock-free modifications and simple resizes; we plan to implement a relativistic split-ordered list in future work.

Our previous work developed a relativistic algorithm for moving a hash-table entry from one bucket to another atomically [24, 23]. This algorithm introduced the notion of cross-linking hash buckets to make entries appear in multiple buckets simultaneously. However, this move algorithm required changing the hash key and potentially copying the entry.

We chose to implement our benchmarking framework `rcuhashbash-resize` as a Linux kernel module, as documented in section 6.1. However, several portable RCU implementations exist outside the Linux kernel. Mathieu Desnoyers reimplemented RCU as a POSIX userspace library, liburcu, for use with pthreads, with no Linux-specific code outside of optional optimizations [4]. For our real-world benchmarks with the memcached key-value storage engine (documented in section 6.2), we used liburcu to support our modified storage engine.

3 Read-Copy Update Background

Read-Copy Update (RCU) provides synchronization between readers and writers of a shared data structure. In sharp contrast to locking, non-blocking synchronization, or transactional memory, RCU readers perform *no expensive synchronization operations whatsoever*: no locks, no atomic operations, no compare-and-swap, and no memory barriers. RCU readers typically incur little to no overhead even compared to concurrency-unsafe single-threaded implementations; furthermore, by avoiding expensive synchronization, RCU readers avoid the need for communication between threads, allowing wait-free operation and excellent scalability.

RCU readers execute concurrently, both with each

other and with writers, and thus readers can potentially observe writers in progress. (Other concurrent programming models prevent readers from viewing intermediate memory states via locking or conflict detection.) The methodologies of RCU-based concurrent programming primarily address the safe management of reader/writer concurrency. Since writers may not impede readers in any way, programmers must reason about the memory states readers can observe, and avoid exposing inconsistent intermediate states from writers.

Commonly, writers preserve data-structure invariants by atomically transitioning data structures between consistent states. On all existing CPU architectures, aligned writes to machine-word-sized memory regions (such as pointers) have atomic semantics, such that a reader sees either the old or the new state, with no intermediate value; thus, structures linked together via pointers support many structural manipulations via direct updates. For more complex manipulations, such as insertion of a new item into a data structure, RCU writers typically allocate memory initially unreachable by readers, initialize it, and then atomically *publish* it by updating a pointer in reachable memory. The publish operation requires a write memory barrier between initialization and publication, to ensure that readers traversing the pointer will observe initialized memory. Readers may also require compiler directives to prevent certain aggressive optimizations across the pointer dereference; RCU wraps those directives into a *read* primitive.[1]

These operations allow RCU writers to update data structures and maintain invariants for readers. However, RCU writers must also manage object lifetimes, which requires knowing when readers might hold references to an item in memory. Unlinking an item from a data structure makes it unreachable to new readers, but does not stop accesses from unfinished readers; writers may not reclaim the unlinked item's memory until all such readers have completed. This resembles a garbage collection problem, but RCU must support runtime environments without automatic garbage collection.

To this end, RCU provides a barrier-like synchronization operation called *wait-for-readers*, which blocks until all readers which started before the barrier have completed. Thus, once a writer makes memory unreachable from the published data structure, a wait-for-readers operation ensures that no readers still hold references to that memory. Wait-for-readers does not prevent *new* readers from starting; it simply waits for existing *unfinished* readers to complete. This barrier operates conservatively: the currently unfinished readers might not hold references to that item, and the barrier itself may wait longer than strictly necessary in order to run efficiently or batch several reclamations into a single wait operation. This conservative semantic allows much more efficient and scalable implementations, particularly for readers. The portion of a writer that follows a wait-for-readers barrier often consists only of memory reclamation; because memory-reclamation operations can safely occur concurrently and need not occur immediately (assuming sufficient memory), common RCU APIs also provide an asynchronous wait-for-readers callback.

However, writers have more reasons to order operations than just reclaiming memory. Our work on relativistic programming provides a general framework for writers to enforce the ordering of operations visible to readers, using the same wait-for-readers primitive. Relativistic programming builds on RCU's key benefits—minimized communication, minimized expensive synchronization, and readers that run concurrently with writers—for scalability. We present here a specific application of the relativistic programming methodology to maintain the data-structure invariants of a hash table while resizing it; section 8 discusses the future development of the general methodology to support maintenance of arbitrary data-structure invariants.

4 Relativistic Hash Tables

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 atomically substitute the new hash table for the old, readers access the hash-table structure through a pointer; this structure includes the array of buckets and the size of the table.

Our resize algorithms synchronize with the corresponding lookup algorithm using existing RCU programming primitives. However, any semantically equivalent implementation will work.

The **RP hash lookup reader** follows the standard algorithm for open-chain hash table lookups:

1. Snapshot the hash-table pointer in case a resizer replaces it during the lookup.

[1] On certain obsolete architectures (DEC Alpha), readers must also use a memory barrier.

2. Hash the desired key, modulo the number of buckets.
3. Look up the corresponding hash bucket in the array.
4. Traverse the linked list, comparing each entry's key to the desired key.
5. If the current entry's key matches the desired key, the desired value appears in the same entry; use or return that value.[2]

In a concrete implementation, lookup (like any RCU reader) will include explicit operations delimiting the start and end of the reader. Depending on the choice of RCU implementation, these delimiter operations may compile to compiler directives, to requests to prevent preemption, to manipulation of CPU-local or thread-local counters, or to other lightweight operations.

Lookups will traverse the hash table concurrently with other operations, including resizes. To avoid disrupting lookups, we require that a lookup can never fail to find a node, even in the presence of a concurrent resize. This means that each hash chain must contain those items that hash to the corresponding bucket. Most prior hash table resize algorithms ensure that a hash chain contains *exactly* those items. We loosen this constraint, instead allowing hash chains to ephemerally contain items that hash to *different* buckets. We call such hash chains *imprecise* since they include all items which hash to that bucket but may include others as well. Readers and writers must tolerate imprecise hash chains (although some operations, such as lookup, require no adaptation). Imprecise hash chains allow us to resize hash tables and otherwise manipulate buckets without copying items or wasting memory.

For simplicity, relativistic hash tables constrain resizing to change the number of buckets by integral factors—for instance, doubling or halving the number of buckets. This guarantees two constraints: First, when shrinking the table, each bucket of the new table will contain all entries from multiple buckets of the old table; and second, 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 constraint, the **RP hash shrink writer** can shrink a 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.
6. Wait for readers. No new readers will have references to the old hash table.
7. Reclaim the old hash table.

Concurrent inserts and removes must block until the shrink algorithm finishes publishing the new hash table and waits for readers to drop references to the old table. See section 4.1 for further details on insertion and removal.

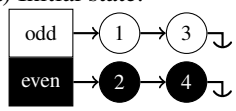
For an example of the shrink algorithm, see figure 1.

Based on the second constraint, the **RP hash expand writer** can expand a table as follows:

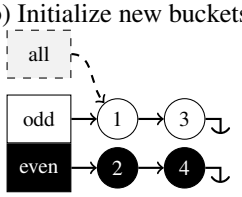
1. Allocate the new, larger table.
2. For each new bucket, search the corresponding old bucket for the first entry that hashes to the new bucket, and link the new bucket to that entry. Since all the entries which will end up in the new bucket appear in the same old bucket, this constructs an entirely valid new hash table, but with multiple buckets "zipped" together into a single imprecise chain.
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 readers. 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 old bucket pointer one or more times 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 readers. New readers will see the changes made in this pass, so they won't miss a node during the next pass.

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

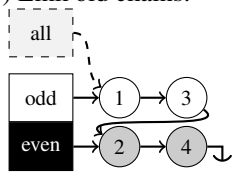
(a) Initial state:



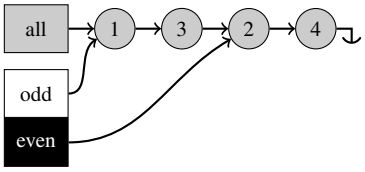
(b) Initialize new buckets:



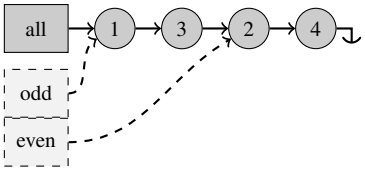
(c) Link old chains:



(d) Publish new buckets:



(e) Wait for readers:



(f) Reclaim:

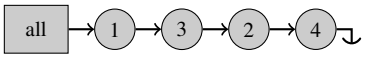


Figure 1: Shrinking a relativistic hash table. (a) The initial state has two buckets, one for odd numbers and one for even numbers. White nodes indicate reachability by odd readers, and black nodes by even readers. (b) The resizer allocates a new one-bucket table and links it to the appropriate old bucket. Dashed nodes exist only in writer-private memory, unreachable by readers. (c) The resizer links the odd bucket's chain to the even bucket, making the odd bucket's chain imprecise. Gray nodes indicates reachability by both odd and even readers. (d) The resizer publishes the new table. (e) After waiting for readers, (f) the resizer can free the old table.

8. If any changes occurred in this pass, repeat from step 6. Note that this loop depends only on the interleaving of nodes with different destination buckets in the zipped bucket, not on subsequent inserts or removals; thus, this loop cannot livelock.
9. Reclaim the old hash table.

The wait in step 7 orders unzip operations for concurrent readers. Without it, a reader traversing a zipped chain could follow an updated pointer from an item in a different bucket, and thus erroneously skip some items from its own bucket.

For an example of the expansion algorithm, see figure 2.

This version of the algorithm uses the old hash table for auxiliary storage during unzip steps. The algorithm could avoid this auxiliary storage at the cost of additional traversals.

Concurrent inserts and removes on a given bucket must block until after all unzips have completed on that bucket.

4.1 Handling Insertion and Removal

Existing RCU-based hash tables synchronize insertion and removal operations with concurrent lookups via standard RCU linked-list operations on the appropriate buckets. Multiple insertion and removal operations synchronize with each other using per-bucket mutual exclusion. (Herlihy and Shavit describe a common workload for hash tables as 90% lookups, 9% insertions, and 1% removals [7], justifying an emphasis on fast concurrent lookups. Nevertheless, several other hash table implementations offer finer-grained update algorithms based on compare-and-swap [2, 22], which we could potentially adapt to improve concurrent update performance.) Resizes, however, introduce an additional operation that modifies the hash table, and thus require synchronization with insertions and removals. We initially consider it sufficient to minimize performance degradation versus a non-resizable hash table, particularly with no concurrent resize running.

During the initial period of initializing new buckets and publishing the new table, our resizers block all updates, either using a hash-table-wide reader-writer lock (where inserts and removes acquire a read lock and resizers acquire the write lock) or by acquiring all per-bucket locks [7]. In the simplest case, concurrent updates can continue to block until the resize completes; however, concurrent updates can potentially run earlier if they carefully handle the intermediate states produced by the resizer. For a sufficiently large hash table, this may prove necessary to avoid excessive delays on concurrent updates.

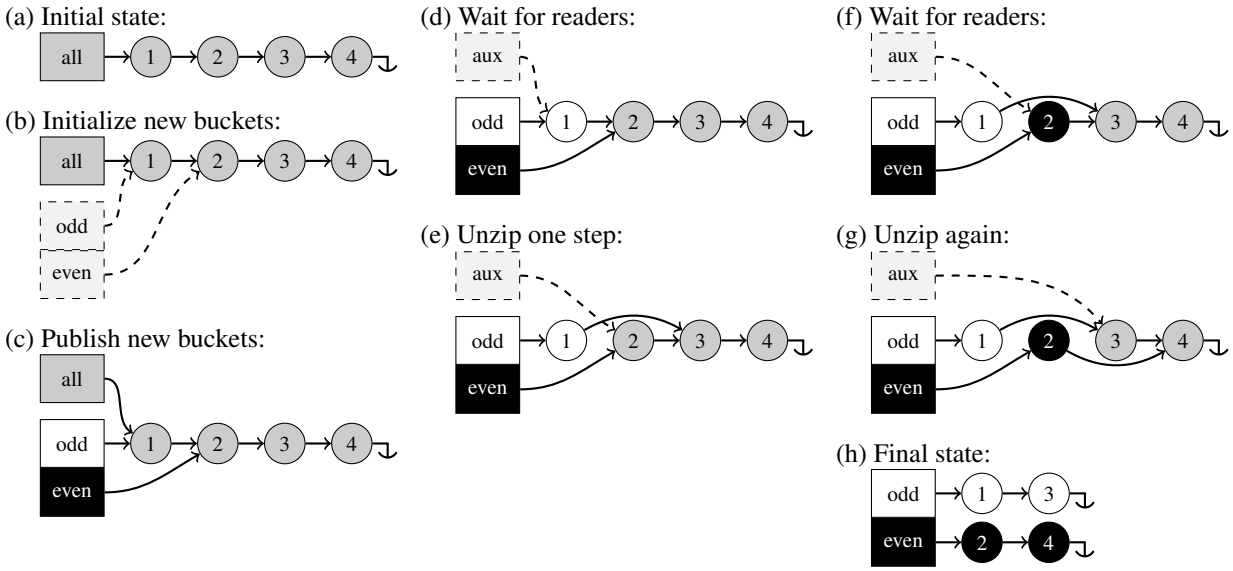


Figure 2: Growing a relativistic hash table. Colors as in figure 1. (a) The initial state contains one bucket. (b) The resizer allocates a new two-bucket table and points each bucket to the first item with a matching hash; this produces valid imprecise hash chains. (c) The resizer can now publish the new hash table. However, an even reader might have read the old hash chain just before publication, making item 1 gray—reachable by both odd and even readers—and preventing safe modification of its next pointer. (d) The resizer waits for readers; new even readers cannot reach item 1. (e) The resizer updates item 1's next pointer to point to the next odd item. (f) After another wait for readers, (g) the unzipping process can continue. (h) The final state.

In our shrink algorithm, the resizer must complete all cross-link steps before publishing the table; once the resizer has published the table, the algorithm has effectively completed with the exception of reclamation, allowing no opportunity for concurrent updates. However, the shrink algorithm could choose to publish the initialized table for updaters as soon as it completes initialization, allowing concurrent updates to proceed while the cross-linking continues. The shrink algorithm may then drop the per-bucket lock for a bucket as soon as it has finished cross-linking that bucket, allowing concurrent insertions and removals on that bucket.

Insertion and removal operations during expansion 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 proves sufficient to handle insertions, which 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 *aux* pointer that marks the start of the next unzip pass. If a removal occurs with a table expansion in progress, the removal must check for a conflict with this aux pointer, and update the pointer if it points to the removed node. Given the relatively low frequency of removal versus lookup and insertion, and the even lower frequency of resizes, we consider it acceptable to require this additional check in the removal algorithm.

4.2 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, reclaim 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 unfinished 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 readers. All new readers will see the concatenated buckets.
4. Set the table size to the new, smaller size.
5. Wait for readers. 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-readers before setting the new size. However, the algorithm still requires auxiliary storage 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.

4.3 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 readers. This approach optimizes resizes significantly.

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 and for removals. Blind inserts without checking for duplicates will incur a performance penalty to find the insertion point; however, insertions which check for duplicates will incur minimal additional cost.

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. A simple sort will not allow concatenation or splitting to preserve the sort, but a well-chosen sort order based on the hash key can allow concatenation without a merge step. Ori Shalev and Nir Shavit presented such a sorting mechanism in their "split-ordered list" proposal [22, 7]: they propose sorting by the bit-reversed key. Alternatively, the bucket selection could use the high-order bits of the hash key.

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

5 Comparisons with Other Algorithms

We evaluated relativistic hash tables through both microbenchmarks on the data structure operations themselves, and via real-world benchmarks on an adapted version of the memcached key-value storage engine. The microbenchmarks directly compare our hash-table resize algorithm with two other resize algorithms: reader-writer locking and DDDS. The real-world benchmarks compare memcached's default storage engine with a modified memcached storage engine based on relativistic hash tables.

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 reclaim 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) [21]. 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 races with a resize. 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 reader first checks for the presence of an old hash table, which indicates a concurrent resize. If present, the lookup proceeds via the concurrent-resize slow path; otherwise, the lookup uses a fast path that simply performs a lookup within the current hash table. The slow path 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. It 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 the-

oretically 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 with relativistic hash tables. However, the DDDS lookup incurs more overhead than relativistic hash tables, 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 relativistic hash tables to outperform DDDS significantly when running a concurrent resize, and slightly even without a concurrent resize.

For a real-world benchmark, we chose memcached, a key-value storage engine widely used in Internet applications as a high-performance cache. Memcached stores key-value associations in a hash table, and supports a network protocol for setting and getting key-value pairs. Memcached also supports timed expiry of values, and eviction of values to limit maximum memory usage.

The default memcached storage engine makes extensive use of global locks. In particular, a single global lock guards all accesses to the hash table. As a result, we expect memcached's default engine to hit a hard scalability limit, beyond which it will not scale to more requests regardless of available resources.

Memcached requires the ability to scale to various workload sizes at runtime; as a result, it requires a resizable hash table. Previous non-resizable RCU hash tables could not provide the flexibility necessary for memcached.

We implemented a new RP-based storage engine in memcached, and modified memcached to support a new fast path for the GET request. memcached's default implementation goes to great lengths to avoid copying data when servicing a GET request; memcached also services multiple concurrent client connections per thread in an event-driven manner. As a result of these two constraints, memcached maintains reference counts on each key-value pair in the hash table, and holds a reference to the found item for a GET from the time of the hash lookup to the time the response gets written back to the client. In implementing the RP-based storage engine, we chose instead to copy the value out of a key-value pair while still within an RP reader; this allows the GET fast path to avoid interaction with the reference-counting mechanism entirely. The GET fast path checks the retrieved item for potential expiry or other conditions which would require mutating the store, and falls back to the slow path in those cases.

We expect that with the new RP-based storage engine, memcached will no longer hit the hard scalability limit observed with the default engine, and GET requests should continue to scale up to the limits of the test machine. Since we added wait-for-readers operations to the SET handling, SET will become marginally slower, but the scalability of SET requests should not change; we believe this tradeoff will prove acceptable in exchange for making GET requests scalable.

6 Benchmark Methodology

6.1 Microbenchmark: rcuhashbash-resize

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 Linux's hlist abstraction, 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, targeting the x86-64 architecture. We used the default configuration (`make defconfig`), with the hierarchical RCU implementation, and no involuntary preemption.

6.2 Real-World Benchmarks: memcached

As a client-server program, memcached required a separate benchmarking program. At the recommendation of memcached developers, we used mc-benchmark, developed by Salvatore Sanfilippo. To minimize the impact of network overhead, we ran the client and server on the same system, communicating via the loopback interface. To generate enough load to reach the limits of memcached, the benchmarking program requires resources comparable to those supplied to memcached. Thus, on the same 24-CPU system, we chose to run 12 memcached threads and up to 12 benchmark processes.

mc-benchmark runs a single thread per process, but simulates multiple clients per process using the same kind of event-driven socket handling that memcached does. Experimentation showed that on the test system, one mc-benchmark process could run up to 4 simulated clients with increasing throughput, but at 4 clients it reached the limit of available CPU power, and adding additional clients would result in the same total request throughput. Thus, we ran from 1 to 12 mc-benchmark processes, each of which simulated 4 clients.

To run the memcached server and mc-benchmark client, and collect statistics on the request rate, we used a benchmark script supplied by the memcached developers. For each test run, the benchmark would start memcached and wait for it to initialize, start the desired number of concurrent mc-benchmark processes, wait 20 seconds for the processing to ramp up (mc-benchmark has to first run SET commands to insert test data, then either SET or GET requests depending on the benchmark), and then collect samples of the rate of processed requests directly from memcached; the benchmark collected three rate samples at 2 second intervals, and took the highest observed rate among those three samples.

7 Benchmark Results

To evaluate baseline reader performance in the absence of resizes, we first compare lookups per second for all the implementations with a fixed table size of 8192 buckets; figure 3 shows this comparison. As predicted, our relativistic hash table, shown as RP, and DDDS remain very competitive when not concurrently resizing, though as the number of concurrent 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 lock 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 16 CPUs, rela-

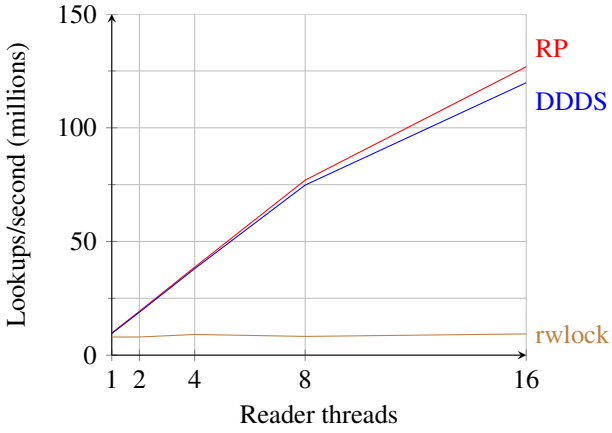


Figure 3: Lookups/second by number of reader threads for each of the three implementations, with a fixed hash-table size of 8k buckets, and no concurrent resizes.

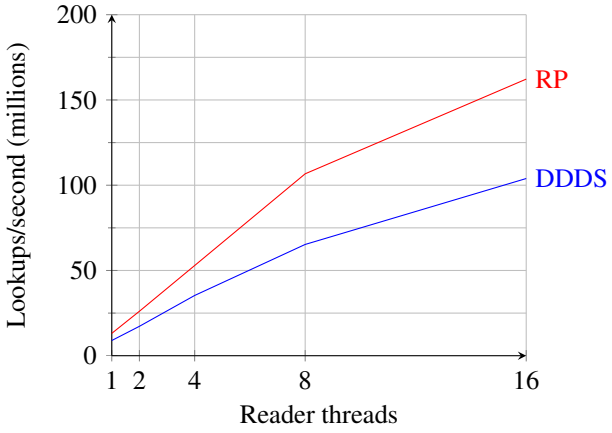


Figure 4: Lookups/second by number of reader threads for our RP-based implementation versus DDDS, with a concurrent resize thread continuously resizing the hash-table between 8k and 16k buckets. rwlock omitted as it vanishes against the horizontal axis.

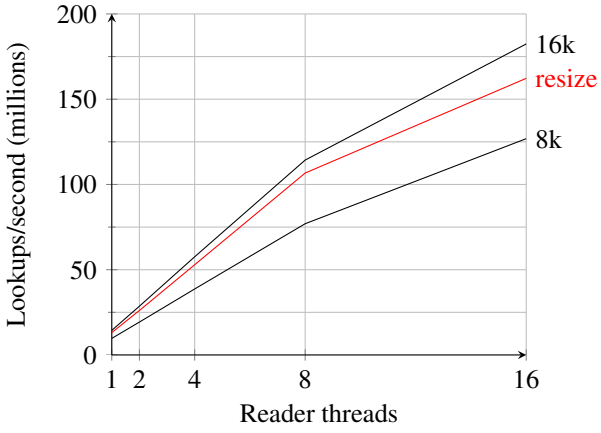


Figure 5: Lookups/second by number of reader threads for our resize algorithms. “8k” and “16k” indicate fixed hash-table sizes in buckets; “resize” indicates continuous resize between the two sizes.

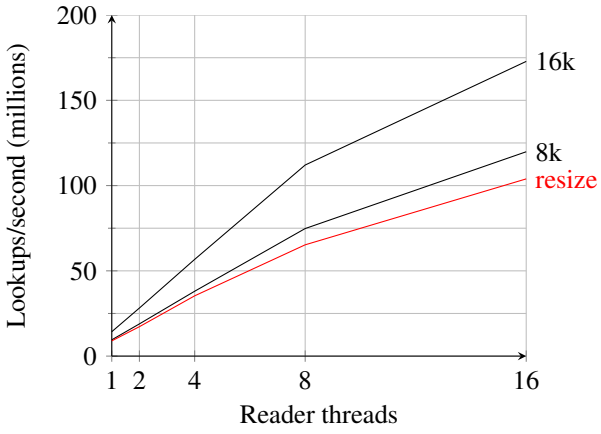


Figure 6: Lookups/second by number of reader threads for the DDDS resize algorithm. “8k” and “16k” indicate fixed hash-table sizes in buckets; “resize” indicates continuous resize between the two sizes.

tivistic hash tables provide 125 times the lookup rate of rwlock.) With a resizer running, our lookup rate scales better than DDDS, with its lead growing as the number of reader threads increases; with 16 threads, relativistic hashing provides 56% more lookups per second than DDDS. 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 predicted.

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.

Figure 8 shows the results of our benchmarks on mem-

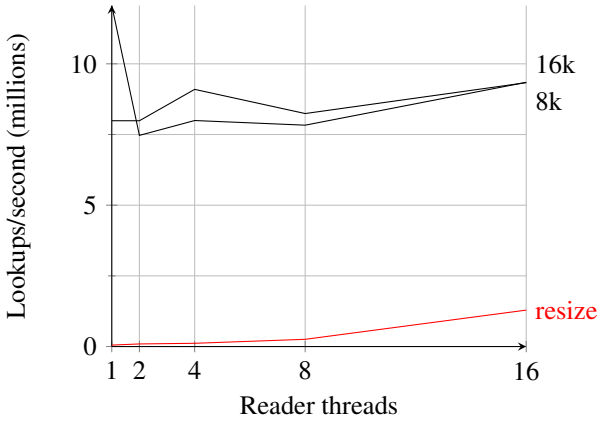


Figure 7: Lookups/second by number of reader threads for the rwlock-based implementation. "8k" and "16k" indicate fixed hash-table sizes in buckets; "resize" indicates continuous resize between the two sizes.

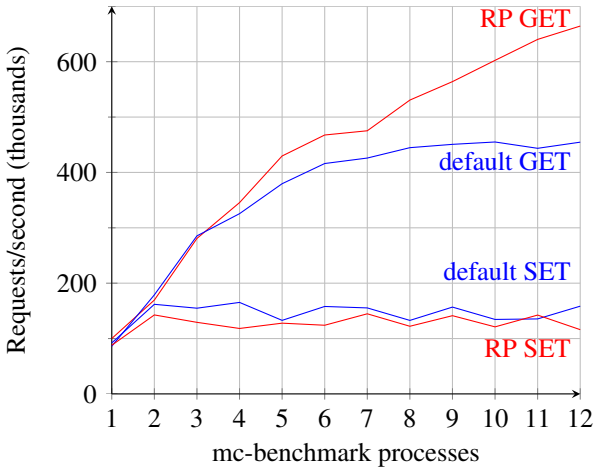


Figure 8: GET and SET operations per second by number of mc-benchmark processes for the default memcached storage engine and our RP-based storage engine. Each mc-benchmark process simulated 4 clients to saturate the CPU.

cached. Note that the default engine hits the expected hard limit on GET scalability, and fails to improve its request processing rate beyond that limit. The RP-based engine encounters no such scalability limit, and the GET rate grows steadily up to the limits of the system. With a full 12 client processes and 12 server threads, memcached with the RP-based engine services 46% more GET requests per second than the default engine.

As expected, SET requests do not scale in either engine. In the RP engine, SET requests incur the expected marginal performance hit due to wait-for-readers operations; however, this tradeoff will prove acceptable for many workloads, particularly when a successful GET request corresponds to a cache hit that can avoid a database query or other heavyweight processing.

We hypothesize that memcached's default engine only managed to scale to as many clients as it did because it spends the vast majority of its time in the kernel rather than in the memcached userspace code, and the kernel code supported more concurrency than the serialized engine code. Profiling confirmed that memcached spends several times as much time in the kernel as in userspace, regardless of storage engine.

We also performed separate runs of the benchmark using the mutex profiler mutrace. By doing so we observed that the default engine spent long periods of time contending for the global lock, whereas with the RP-based engine, GET requests no longer incurred any contention for the global lock.

7.1 Benchmark Summary

Our relativistic resizable hash table provides linearly scalable lookup performance in both microbenchmarks and real-world benchmarks. In our microbenchmarks, the relativistic implementation surpassed DDDS by a widening margin of up to 56% with 16 reader threads; both implementations vastly dwarfed reader-writer locks, with our RP implementation providing a 125x improvement with 16 readers. Furthermore, our resize algorithms minimized the impact of concurrent resizing on lookup performance, as demonstrated through the comparison with fixed-size hash tables. In the real-world benchmarks using memcached, the RP-based engine eliminated the hard scalability limit of the default storage engine, and consistently serviced more GET requests per second than the default engine—up to 46% more requests per second when saturating the machine with a full 12 client processes and 12 server threads.

8 Future Work

Our proposed hash-table resize algorithms demonstrate the use of the wait-for-reader operation to order update operations. We use this operation 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 traversed 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 tra-

verse 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 lookups to retry in a secondary hash table if the primary lookup failed; this approach mirrored that of the DDDS lookup slow path. 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 [24, 23]. 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.

9 Availability

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

10 Acknowledgments

Thanks to Nick Piggin, Linux kernel hacker and inventor 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, Phil Howard, Eddie Kohler, and the USENIX ATC reviewers for their review, feedback, and advice. Thanks to memcached developer "dormando" for providing technical help with memcached, as well as the benchmarking framework. Thanks to Salvatore Sanfilippo for the original mc-benchmark.

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. A 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] HART, T. E., MCKENNEY, P. E., BROWN, A. D., AND WALPOLE, J. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing 67*, 12 (2007), 1270–1285.

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

[8] HOWARD, P. W., AND WALPOLE, J. A relativistic enhancement to software transactional memory. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 2011)* (2011).

[9] 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`.

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

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

[12] 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.

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

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

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

[16] 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.

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

[18] 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.

[19] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures* (2002), SPAA '02, pp. 73–82.

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

[21] PIGGIN, N. ddds: "dynamic dynamic data structure" algorithm, for adaptive dcache hash table sizing. Linux kernel mailing list. `http://mid.gmane.org/20081007064834.GA5959@wotan.suse.de`, October 2008.

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

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

[24] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Scalable Concurrent Hash Tables via Relativistic Programming. *ACM Operating Systems Review 44*, 3 (July 2010).

[25] XU, H. bridge: Add core IGMP snooping support. Linux netdev mailing list. `http://mid.gmane.org/E1NlbuT-00021C-0b@gondolin.me.apana.org.au`, February 2010.