1-1-2012

Relativistic Causal Ordering A Memory Model for Scalable Concurrent Data Structures

Josh Triplett
Portland State University

Relativistic Causal Ordering

A Memory Model for Scalable Concurrent Data Structures

by

Josh Triplett

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Jonathan Walpole, Chair
Barton Massey
James Hook
James McNames
Mark Jones
Paul E. McKenney

Portland State University
2012

Abstract

High-performance programs and systems require concurrency to take full advantage of available hardware. However, the available concurrent programming models force a difficult choice, between simple models such as mutual exclusion that produce little to no concurrency, or complex models such as Read-Copy Update that can scale to all available resources.

Simple concurrent programming models enforce atomicity and causality, and this enforcement limits concurrency. Scalable concurrent programming models expose the weakly ordered hardware memory model, requiring careful and explicit enforcement of causality to preserve correctness, as demonstrated in this dissertation through the manual construction of a scalable hash-table item-move algorithm. Recent research on *relativistic programming* aims to standardize the programming model of Read-Copy Update, but thus far these efforts have lacked a generalized memory ordering model, requiring data-structure-specific reasoning to preserve causality.

I propose a new memory ordering model, *relativistic causal ordering*, which combines the scalabilty of relativistic programming and Read-Copy Update with the simplicty of reader atomicity and automatic enforcement of

causality. Programs written for the relativistic model translate to scalable concurrent programs for weakly-ordered hardware via a mechanical process of inserting barrier operations according to well-defined rules.

To demonstrate the relativistic causal ordering model, I walk through the straightforward construction of a novel concurrent hash-table resize algorithm, including the translation of this algorithm from the relativistic model to a hardware memory model, and show through benchmarks that the resulting algorithm scales far better than those based on mutual exclusion.

Acknowledgments

Thanks to my parents for a lifetime of unwavering support.

Thanks to Paul McKenney for the original invention of Read-Copy Update, for taking the time to talk with a new IBM intern, and for an endless supply of interesting challenges and thought-provoking discussion.

The work presented in this dissertation includes refined versions of work previously published in the Linux Plumbers Conference [76], the ACM Operating Systems Review [77], and the USENIX Annual Technical Conference [78]. Thanks to the reviewers and audiences of those conferences for their feedback. Particular thanks to Eddie Kohler for his extensive review and shepherding of the hash-table resize paper.

Thanks to Nick Piggin, Linux kernel hacker and inventor of the DDDS algorithm, for reviewing my implementation of DDDS (used in chapter 6) to ensure that it fairly represents his work. Thanks to memcached developer "dormando" for providing technical help with memcached, as well as the benchmarking framework. Thanks to Salvatore Sanfilippo for mc-benchmark.

Thanks to IBM and Intel for access to highly parallel systems used for benchmarking.

The term "relativistic programming", and the goal of generalizing and standardizing the Read-Copy Update programming model, originated with Jonathan Walpole, founder of the Relativistic Programming research group. The Relativistic Programming research group also includes Paul McKenney, James Hook, Mathieu Desnoyers, Phil Howard, Eric Wheeler, Jim Cotillier, and myself. Thanks to all the members of this research group.

Particular thanks to my advisor, Jonathan Walpole, and my dissertation committee, Bart Massey, James Hook, James McNames, Mark Jones, and Paul McKenney, for their help, support, guidance, and feedback.

During the course of this research, I received financial support from two Maseeh Graduate Fellowships, and from the National Science Foundation under Grant No. CNS-0719851. Thanks to Dr. Fariborz Maseeh and the National Science Foundation for their support.

Table of Contents

List of Figures

Chapter 1

Introduction

Moore's Law predicts that the density of transistors in an integrated circuit will double every two years [61], and this prediction has consistently held true [62]. The performance of individual processors matched this trend for much of the lifetime of the microprocessor, leading to the expectation that such exponential growth would continue. However, individual processors have long since reached physical limitations that seem to make exponential growth unsustainable [73].

Instead, the foreseeable future of faster computation now involves parallelism [73]. Given the rapid growth of clock frequency when it represented the primary focus of the processor industry, it seems probable that we will one day find systems with merely hundreds of processor cores as quaint as we currently find systems with merely hundreds of Hz.

Thus, a program that requires increasing amounts of processing power cannot simply run faster on a single processor core, but must now spread itself across multiple cores. This requires structuring the program as *threads* that can run in parallel; each thread can then run on a separate core.

Some programs divide perfectly into threads without any shared data required; we refer to such programs as *embarrassingly parallel*. However, most parallel programs require some concurrent access to shared data structures, typically including both reads and writes to those data structures. This introduces the additional problem of *synchronization*: mediating access to those shared data structures to preserve the expected semantics.

Any engineer who has worked in a professional setting knows intuitively that "meetings are toxic" to productivity [17]. Likewise, synchronization represents pure overhead: it forces useful work to stop in favor of serialized coordination, negating the benefits of parallelism for the duration of the synchronization event.

Ideally, CPU-bound programs should scale linearly with the addition of cores, just as they did with increases in single-core performance. However, most parallel programs do not achieve that goal, instead producing diminishing returns and eventually performance degradation. In this suboptimal state, parallel programs spend most of their time synchronizing between processors, rather than performing useful work. These scalability limits arise from a mismatch between parallel programs and the architecture of modern parallel systems.

The same phenomena leading to the *memory wall* of slow communication between processors and memory [82, 46] now arise in the communication between processors. Communication between processors takes far longer than execution of code on a single processor. Furthermore, the same caches used

to mitigate the memory wall on single-processor systems can become a liability in shared-memory architectures with multiple processors, as the caches may take longer to keep coherent than to bypass entirely [12]. As processors become both faster and more numerous, the latency of communication between processors will grow far larger relative to the speed of those processors, exacerbating this problem.

Similarly, processor performance features such as pipelining and out-of-order execution, designed to keep a single core saturated for maximum performance, can interfere with the correct execution of concurrent programs. Thus, concurrent synchronization suppresses these processor features to preserve correctness, further limiting performance.

Mutual exclusion represents the most commonly applied technique for concurrent programming in both research and production use. Implementations of mutual exclusion provide a *lock* that many threads may simultaneously attempt to acquire; only one of those threads will succeed, and the rest will wait. The term *critical section* refers to the section of code between lock acquisition and lock release; only one critical section for a given lock may run at the same time. The fundamental properties of mutual exclusion have remained unchanged since its original introduction by Dijkstra [14] and Hoare [32]: atomic execution and serialization of critical sections.

These properties ensure that mutual exclusion enforces a strict ordering of all operations on shared data, simply by wrapping each such operation with the appropriate lock for that data. The implementations of the lock

and unlock primitives in a mutual exclusion system ensure that no operation within a critical section can become visible to other threads without all other operations from that critical section becoming visible as well. Together with the serialization of critical sections, this effectively allows multithreaded programs to ignore concurrency, using the same algorithms and data structures designed for single-threaded programs, just wrapped in locks. Ordering models such as sequential consistency [39] and linearizability [31] formalize this extension of single-threaded reasoning to multithreaded programs.

However, this strict ordering incurs a high cost [30]. Mutual exclusion limits concurrency to at most one thread accessing any particular piece of shared data at a time, with other threads blocked. Fine-grained locking allows concurrent access to disjoint components of shared data structures, but does not permit concurrent access to the same data. Furthermore, the requirement to enforce ordering forces the locking primitives to perform expensive communication across threads and CPUs, and to disable the previously mentioned processor performance features that could otherwise affect synchronization correctness. Even when uncontended, acquiring a lock has a high cost, and does not scale linearly; threads acquiring uncontended locks will nonetheless contend for memory, cache, bus bandwidth, and other resources.

Thus, data structures that scale need to minimize or eliminate synchronization. However, "Laws of Order" [4] proved that correct implementations of concurrent algorithms necessarily require expensive synchronization instructions to preserve correctness. This result shows that scalable concurrent

algorithms cannot eliminate synchronization entirely. Despite this, an algorithm need not require synchronization in all cases or on all threads; some parts of the algorithm can eliminate synchronization and become scalable, at the expense of other parts of the same algorithm. By making the most performance-critical portions of algorithms scalable, the system as a whole can scale for the expected class of target workloads.

Read-Copy Update (*RCU*) provides one such class of scalable concurrent algorithms [47, 23, 56], which has seen extensive testing, correctness analysis, and proofs [48, 52, 10]. RCU chooses to eliminate synchronization from readers, making those readers scalable, and moving all necessary synchronization to writers. In particular, RCU-based data structures provide scalable readers that can always proceed, even in the presence of concurrent writers; writers must ensure that readers always see a consistent view of the data structure at all times. RCU-based readers can thus scale linearly to the limits of available resources.

RCU's scalable readers make RCU-based algorithms and data structures ideally suited for read-mostly workloads, and such workloads appear quite frequently in many applications. In practice, RCU remains quite scalable for all but the most heavily write-biased data structures, with performance degrading gracefully for write-mostly data.

The original formulation of RCU and the RCU primitives by McKenney [47] provided algorithms for scalable linked lists, including insertion, deletion, and concurrent lookup. The correctness of this linked list followed

trivially from the definitions and properties of the RCU primitives, and each linked-list operation required no non-trivial steps beyond the standard application of the RCU primitives. In particular, each linked-list insertion or deletion consisted of a single semantically significant change, implemented via a single critical pointer manipulation, with no visible intermediate states. Chapter 2 explains the RCU primitives and RCU-based algorithms in detail, using linked lists as an example.

This formulation extends naturally to other simple data structure operations. Hash tables using chaining already have a linked list in each hash bucket, and thus RCU-based hash tables simply use RCU linked-list operations within each bucket [47]. RCU-based radix trees [63] effectively increase the number of "next" pointers per node, but similar reasoning applies, and each tree operation still consists of a single semantic change to the tree, reducing to a single critical pointer manipulation.

However, extending RCU to more complex data structure operations requires significantly more complex reasoning for correctness. RCU hash tables support insertion and deletion, but moving an entry from one bucket to another using those operations would expose an invalid intermediate state to readers, containing either zero or two copies of the entry. A dedicated hash-table move operation [76, 77] makes the move appear atomic, but requires multiple critical pointer manipulations that must appear to the readers in order. Similarly, the application of RCU to red-black trees requires ordered groups of multiple pointer manipulations for rotation operations [34], and

resizing a hash table with concurrent readers requires an ordered group of pointer manipulations to ensure that readers see all elements in the table at all times [78].

None of these data structure operations fit the simple models derived from the original linked-list algorithms. The requirements to order multiple pointer manipulations require novel applications of the RCU primitives beyond their original intended purposes. As a result, each new data structure requires careful one-off reasoning for correctness, focusing on the specific correctness properties desired for each data structure, without a general-purpose programming model. Because RCU readers do not provide atomicity, and because the writers wish to minimize the number of synchronization operations required, the correctness arguments must consider many possible orderings of individual load and store operations—in fact, the number of possible orderings grows exponentially with the number of operations. These orderings require unintuitive reasoning, because they contradict program order and causality.

Each new scalable data structure based on RCU requires similarly careful construction and analysis, limiting the development of such data structures to experts. Chapter 3 gives an example of the manual construction of such data structures, demonstrated via the previously mentioned hash-table move operation; this algorithm represents a novel contribution of this dissertation.

Many more algorithms exist that require multiple pointer manipulations in a single write operation, and these algorithms prove quite complex to im-

plement in a scalable way using techniques such as RCU. Similar problems exist with non-blocking synchronization techniques, whose readers likewise lack any inherent order in which store operations become visible. Many attempts at scalable transactional memory implementations introduce the idea of "invisible readers" that avoid the overhead of making their transactions visible to writers; to the extent these invisible readers can avoid the costs of heavyweight synchronization and ordering, they incur the same issues with unenforced ordering. One of the most scalable implementations of transactional readers uses RCU [33], and inherits the same ordering issues.

All of these cases suggest the need for a general-purpose construction technique for scalable algorithms. Recent research on *relativistic programming* aims to standardize such a programming model by generalizing from the common patterns of RCU-based algorithms. However, thus far these efforts have lacked a generalized memory ordering model for the store operations performed by writers, requiring data-structure-specific reasoning to preserve causality. Such an ordering model would allow the construction of new concurrent algorithms with the same well-explored ease allowed by mutual exclusion, but with the scalability that mutual exclusion cannot provide. To avoid the expensive synchronization required by mutual exclusion, this ordering model must not enforce complete atomicity of all store operations in a writer, only the order those stores become visible to readers.

I propose a new ordering model for scalable algorithms, *relativistic causal ordering*. This model ensures that readers observe a writer's store opera-

tions in an order consistent with the causal ordering specified by that writer. Relativistic causal ordering avoids enforcing an expensive total global ordering of memory operations. Instead, it enforces a causal ordering of stores: any two stores whose ordering matters for the correctness of a write operation will become visible in the writer's program order. This provides an intuitive ordering that allows straightforward construction of scalable concurrent algorithms without complex reasoning about weak memory models and reordering. Chapter 4 documents the relativistic causal ordering model.

The term "relativistic" in "relativistic programming" draws a parallel to the theory of relativity in physics, avoiding an absolute reference frame with a total ordering of events in favor of observer-relative reference frames that may observe events in different orders. Relativistic causal ordering improves on this parallel by preserving the order of causally related events, allowing two events to appear in different orders to different observers only if the two events have no causal relationship.

The relativistic causal ordering model does not directly correspond to the semantics of any particular multiprocessor architecture or memory model. Thus, defining the relativistic model as the ideal memory model for scalable data structures does not by itself allow the construction of such data structures, merely the construction of algorithms that rely on the properties and assumptions of the relativistic model. Running such algorithms on real hardware requires a means of translating them to the native memory model of that hardware, including the insertion of memory barriers or other appropri-

ate forms of synchronization where required to preserve relativistic ordering. The selection of appropriate barriers requires platform-specific knowledge [6], but a relativistic programming framework can provide a portable interface to those barriers suitable for use on many different target platforms.

Chapter 5 shows how to translate programs written for the relativistic causal ordering model to programs suitable for execution on weakly ordered hardware, with minimal requirements on the underlying hardware. Taken together, the relativistic causal ordering model and its translation to hardware ordering model constitute the primary contributions of this dissertation, enabling the construction of scalable concurrent algorithms for arbitrary acyclic data structures.

Section 4.6 provides a worked example of applying the relativistic causal ordering model to recreate the classic RCU-based linked list. Chapter 6 walks through the significantly more complex example of a relativistic resizable hash table, producing novel algorithms for resizing a hash table while allowing concurrent readers. In both cases, the ordering model and the required data structure properties make the generation of the necessary algorithms straightforward, without any need for complex one-off reasoning. These generated algorithms avoid the need for expensive communication or synchronization instructions in readers, allowing those readers to scale. Benchmarks of the relativistic resizable hash table in sections 6.3 and 6.4 show that algorithms based on this methodology can achieve the high scalability expected of RCU-based algorithms.

1.1 Summary of Key Background

Chapter 2 provides a detailed background of concurrent programming techniques and research. However, this dissertation builds most closely on two foundational components of background:

- The *Read-Copy Update* (RCU) synchronization technique, methodology, and primitives, as originally invented by Paul McKenney [47].
- *Relativistic programming* (RP), the concept of generalizing and standardizing the Read-Copy Update programming model to ease the creation of new scalable data structures. This term and concept originated with Jonathan Walpole, founder of the Relativistic Programming research group at Portland State University [65].

1.2 Summary of Contributions

This dissertation presents four novel contributions:

1. A node-move algorithm for scalable concurrent hash tables. Chapter 3 documents this algorithm, both as a novel contribution itself and as a demonstration of the complex reasoning required to manually construct relativistic data structures without using the relativistic causal ordering model.
2. The relativistic causal ordering model, a memory model supporting

straightforward causal reasoning about scalable concurrent data structures. Chapter 4 documents the relativistic causal ordering model.

3. A translation from algorithms designed for the relativistic causal ordering model to algorithm implementations targeted at real hardware memory models, providing concrete implementations of the abstract barriers used in relativistic causal ordering. Chapter 5 documents this translation.

4. An algorithm for resizing a concurrent hash table while supporting concurrent, scalable readers. Chapter 6 documents this algorithm, both as a novel contribution itself and as a demonstration of constructing scalable concurrent data structures using the relativistic causal ordering model.

Chapter 2

Background on Scalable Concurrent Programming

2.1 Mutual Exclusion

Mutual exclusion, or *locking*, represents the most commonly applied technique for concurrent programming in both research and production use. Several approaches exist for mutual exclusion, many of them dependent on features of the underlying hardware and instruction set, but all achieve the same effect: they provide a *lock* that many threads may simultaneously attempt to acquire, only one of which will succeed while the rest wait. The term *critical section* refers to the section of code between lock acquisition and lock release, during which no other code using the same lock can run. Dijkstra's *semaphore* [14] and Hoare's *monitor* [32] provide the archetypal examples of the *locking* form of mutual exclusion, and its basic structure and fundamental properties have remained unchanged since its introduction.

The simplest application of locking to a concurrent program wraps shared data structures with a single global lock. Portions of the program accessing the shared data structure must hold this lock. Correctness of this approach

proves trivial to verify, but it allows no concurrency between critical sections on the same structure; such an approach will not scale unless the critical sections represent a minimal fraction of the total work.

Using the example of a simple hash table, a reader performing a lookup operation using global locking would involve the following steps:

1. Hash the desired key to determine the bucket containing the target entry.
2. Acquire the lock for the entire hash table.
3. Traverse the bucket containing the target entry, comparing each entry to the desired key; stop after finding the entry or reaching the end of the list.
4. Drop the lock for the entire hash table.

A writer based on global locking could insert a new entry via the following steps:

1. Hash the new key to determine the bucket that will contain the new entry.
2. Allocate and initialize the new hash-table entry.
3. Acquire the lock for the entire hash table.
4. Insert the new entry at the head of the target bucket.

5. Drop the lock for the entire hash table.

Notice that only one read or write operation can occur at any given time. This serialization prevents readers or writers from seeing a partially completed write operation, but it does not permit any concurrency or scalability.

Even this simple application of mutual exclusion introduces several potential problems with both correctness and performance, including deadlock [40, 84, 53], priority inversion [40, 53], convoying [19], and minimal fault-tolerance [19, 53]. Nonetheless, a large body of research exists addressing or discussing these various problems, and providing standard implementation techniques for lock-based data structures. Global locking also provides a straightforward approach for adding concurrency to complex data structures and systems not readily amenable to more concurrent forms of synchronization. For example, the original introduction of multiprocessor support into the Linux kernel used a single global "Big Kernel Lock" which allowed only one processor to execute in the kernel at a time; this allowed the existing sequential code in the Linux kernel to function on concurrent systems, and still permitted concurrent execution of userspace processes.

Reader-writer locks (*rwlocks*) attempt to increase concurrency by permitting read-only algorithms to run in parallel with each other, on the assumption that they cannot interfere with each other. Readers and writers use separate operations to acquire a reader-writer lock for reading or for writing. Acquiring the lock for readers excludes writers, but not other readers; acquiring the lock for writers excludes both readers and writers.

For the simple hash-table example, a lookup algorithm based on reader-writer locking proceeds as follows:

1. Hash the desired key to determine the bucket containing the target entry.
2. Acquire the whole-hash-table reader-writer lock for reading.
3. Traverse the bucket containing the target entry, comparing each entry to the desired key; stop after finding the entry or reaching the end of the list.
4. Drop the whole-hash-table reader-writer lock for reading.

The corresponding insert algorithm becomes:

1. Hash the new key to determine the bucket that will contain the new entry.
2. Acquire the whole-hash-table reader-writer lock for writing.
3. Insert the new entry at the head of the target bucket.
4. Drop the whole-hash-table reader-writer lock for writing.

Unlike the implementation based on global locking, this implementation potentially allows readers to run concurrently.

However, reader-writer locks still force read-only algorithms to perform writes (to the locks themselves), leading to additional overhead from cache

coherence algorithms and false sharing. More advanced reader-writer locks avoid this overhead by not caching lock values [12], but this limits the speed of lock operations to that of the uncached memory location shared by all processors. Hseih and Weihl [35] constructed a more scalable reader-writer lock by assigning a separate lock to each CPU, with readers acquiring their local lock and writers acquiring all locks; this approach potentially allows readers to scale when writers remain rare, but it degrades badly in all but the most read-biased workloads. Reader-writer locks may also require additional logic to avoid writer starvation. All these forms of additional overhead make reader-writer locking potentially slower than standard mutual exclusion, since the time required for a lock operation may outweigh the critical section it guards; the lock-protected operation can become effectively serialized due to lock overhead. Even with this additional overhead, reader-writer locking only permits concurrency between readers; readers must still block concurrent writers, and writers must still block concurrent readers.

To allow more concurrency, most lock-based algorithms use *fine-grained locking*: different portions of a shared data structure will have different associated locks, and code accessing shared data need only hold the locks for the portions it accesses. This allows a degree of *disjoint-access parallelism*: access to disjoint portions of shared data can occur concurrently.

A hash table provides obvious boundaries for fine-grained locks: each bucket has a separate lock. With fine-grained locking, the simple hash-table lookup operation becomes:

1. Hash the desired key to determine the bucket containing the target entry.

2. Acquire the lock for that bucket.

3. Traverse the bucket containing the target entry, comparing each entry to the desired key; stop after finding the entry or reaching the end of the list.

4. Drop the lock for that bucket.

The corresponding insert algorithm becomes:

1. Hash the new key to determine the bucket that will contain the new entry.

2. Acquire the lock for that bucket.

3. Insert the new entry at the head of the target bucket.

4. Drop the lock for that bucket.

This implementation allows readers and writers to run concurrently as long as they access separate buckets. Readers and writers will only block each other when attempting to access the same bucket; the per-bucket locks will serialize the reads and writes of each bucket, allowing one to proceed at a time.

Implementing a data structure using fine-grained locking rather than global locking exacerbates the correctness and performance problems of global

locking, including deadlock [40, 84, 53], priority inversion [40, 53], convoying [19], and minimal fault-tolerance [19, 53]. However, existing research on fine-grained locking has provided well-known implementation techniques for fine-grained locking that solve or avert many of these problems. As a result, fine-grained locking enjoys considerable success as the primary concurrency methodology for practical concurrent systems.

However, the complexity of fine-grained locking often leads to implementations using coarser locking than they might otherwise need. Many data structures do not prove amenable to the partitioning necessary for fine-grained locking. Coarse-grained locking simplifies the mental model of a data structure by reducing difficult-to-verify bits of potential concurrency in a set of algorithms, but this loss of concurrency further reduces the scalability of those algorithms.

Conversely, programs that use highly fine-grained locking risk a different scalability problem: if performing an operation requires acquiring many locks, the lock overhead becomes a larger part of the overall cost of the operation. For example, in the simple hash table presented above, a writer moving a node between buckets must acquire the locks for both buckets; a writer resizing the table must acquire the locks for *all* buckets. So while the fine-grained locking might theoretically allow more concurrency when executing the body of a critical section, the multiple locks protecting the critical section may take significantly longer to execute, during which time the critical section generates more contention for system resources such as

bus bandwidth and cache lines. If this overhead significantly outweighs the critical section itself, operations may become effectively serialized.

Furthermore, fine-grained locking still allows only one form of concurrency: disjoint-access parallelism. Accesses to the same location, protected by the same lock, will always proceed in series with no concurrency. This leads to high contention for frequently accessed locations. In particular, no form of mutual exclusion allows concurrency between readers and writers accessing the same location.

Fundamentally, both standard mutual exclusion and reader-writer locking still require global agreement between processors regarding the ownership of locks, which mandates communication between processors. The latency of this communication represents the primary limit on the performance and scalability of lock-based algorithms.

2.2 Non-Blocking Synchronization

Non-blocking synchronization (NBS) provides a solution for some of the complexity of locking [19]. NBS uses primitives such as compare-and-swap (CAS) or load-linked/store-conditional (LL/SC) to copy data before modifying it, optimistically assume that the modification will succeed, and then roll back the operation if a concurrent modification to the same location occurred.

Massalin and Pu [45] presented a lock-free stack using non-blocking synchronization via CAS. For that stack, the pop operation follows these steps:

1. Save the current (old) value of the stack pointer.

2. Compute the new value of the stack pointer.

3. Dereference the old stack pointer to obtain the element currently at the top of the stack.

4. Use Compare and Swap (CAS) to atomically replace the stack pointer with the new value computed in step 2 if the stack pointer still matches the old value saved in step 1. If the CAS fails due to the stack pointer having changed, retry from step 1.

5. Return the element obtained in step 3.

In this example, if two or more attempts to pop a value from the stack occur concurrently, they will conflict, and all but one of them will fail in step 4 and retry. Thus, the stack in this example does not actually provide any concurrency.

However, non-blocking synchronization improves fault-tolerance and mitigates priority inversion and deadlock by forcing slow or hung tasks to roll back (if they ever resume) rather than waiting on them. With mutual exclusion, the first thread to acquire the lock can hold that lock indefinitely, and can thus indefinitely delay all other threads that need to acquire the lock. By contrast, with NBS, the first thread to attempt to complete the operation will win, forcing all other threads to roll back even if they started earlier; thus, a thread which fails to complete its operations will not block other threads

from proceeding. This fault-tolerance makes it potentially reasonable to use non-blocking synchronization even when it does not provide a performance advantage over mutual exclusion.

Furthermore, though this stack pop example does not provide concurrency, in general NBS provides disjoint-access parallelism: accesses to disjoint locations can always proceed without rolling each other back, without introducing the problems of fine-grained locking. (The stack in this example has only a single stack pointer, and thus no disjoint accesses could occur.)

However, non-blocking synchronization introduces several new forms of complexity not present in locking. Most prominently, it allows concurrent execution of conflicting critical sections, even if all but one would roll back. Furthermore, any changes made to a data structure may become visible immediately, which readers may not expect. Thus, critical sections must avoid unrecoverable errors such as accessing invalid pointers, even in the face of concurrent modifications. This makes the reasoning model of NBS significantly more complex than that of locking.

Specific implementations of NBS may also introduce new issues. With implementations such as CAS based on comparison of values, writers may need to take extra steps to avoid the *A-B-A problem*, in which a value (such as a pointer) does not appear changed because it changes twice and returns to the original value. Conversely, with implementations such as LL/SC based on cache coherence protocols or other mechanisms to observe changes, the watched region of memory may have a granularity larger than a single mem-

ory location, which can lead to another form of false sharing: a critical section may roll back due to an unrelated memory operation that invalidates the value saved for comparison.

As a result of these additional forms of complexity, non-blocking algorithms prove significantly less common than those based on mutual exclusion. Various implementations of individual data structures exist, such as linked lists [22], queues [59, 69], and stacks [24, 75, 69]. However, as with fine-grained locking, these implementations lack a general methodology: they do not provide tools to support implementation of other data structures without additional ingenuity. As Michael and Scott [60] put it, "Good data-structure-specific multilock and nonblocking algorithms are sufficiently tricky to devise that each has tended to constitute an individual publishable result."

Herlihy proposed a generalized construction for a non-blocking version of any arbitrary data structure [25]. However, this construction entails making a full copy of the data structure and atomically replacing the old version. While this result provides theoretical generality, the severe performance penalty it imposes on write operations typically proves impractical. Thus, the general result does not scale, and the scalable results do not generalize.

Finally, while non-blocking synchronization inherently provides fine-grained disjoint-access parallelism, it does nothing to address the joint-access case. Concurrent accesses to the same location will effectively devolve to mutual exclusion, with the added problem that all but one concurrent operation will proceed optimistically and then roll back, introducing additional computa-

tion and contention rather than just waiting.

2.3 Transactional Memory

Transactional memory (TM) proposes a solution to the complexity of non-blocking synchronization [27, 26, 15]. In a TM system, readers and writers denote critical sections explicitly by beginning and ending transactions. A transaction behaves as if all memory operations within it occur as an atomic unit with respect to other transactions.

For the simple hash-table example introduced in section 2.1, a transactional lookup proceeds as follows:

1. Hash the desired key to determine the bucket containing the target entry.

2. Begin a transaction.

3. Using transactional load operations, traverse the bucket containing the target entry, comparing each entry to the desired key; stop after finding the entry or reaching the end of the list.

4. End the transaction.

The corresponding transactional insert algorithm becomes:

1. Hash the new key to determine the bucket that will contain the new entry.

2. Begin a transaction.

3. Using transactional load and store operations, insert the new entry at the head of the target bucket.

4. End the transaction.

In this implementation, two concurrent transactions will conflict if one performs a transactional store to a given location and the other performs a transactional load or store on that location. When such a conflict occurs, one of the transactions will succeed, and the other will roll back and retry. Thus, this transactional implementation allows lookup operations to run concurrently with each other on the same or different buckets, and allows insert operations to run concurrently with insert or lookup operations on different buckets. An insert operation will conflict with another concurrent insert or lookup on the same bucket.

Transactional memory provides the same automatic disjoint-access parallelism that NBS does, as well as the fault-tolerant behavior associated with optimistic concurrency. However, unlike non-blocking synchronization, transactional memory offers a more general methodology: wrap all sets of accesses to shared data in transactions. Unlike the general methodology for NBS, the general transactional methodology achieves the same theoretical performance behavior without needing specialized code for each data structure.

It remains unclear, however, just how much performance transactional memory can provide. Software transactional memory (STM) implementa-

tions suffer from severe performance and scalability limitations, particularly in low-contention or contention-free cases [7, 66, 80, 11, 44]. Furthermore, contention for the same memory location will lead to serialized transactions and no concurrency, just as with mutual exclusion. Only when many transactions access disjoint locations can STM begin to achieve any concurrency.

STM also suffers from the performance problems of both fine-grained locking and NBS. As with excessively fine-grained locking, the overhead of the transaction itself may eliminate the possibility of concurrent transactions. Furthermore, as with NBS, transactions that optimistically proceed and then roll back introduce additional computation and contention, adding further to critical section overhead.

Higher-performance implementations of software transactional memory have begun to abandon the high-overhead optimistic approach that adds fault-tolerance in favor of lock-based implementations with lower overhead [12, 16].

Much other research has sought out techniques to accelerate STM. For instance, hash-based STM implementations [68, 43, 21] hash the set of locations accessed by a transaction, providing fast but imprecise conflict detection. This approach reduces overhead, but introduces another form of false sharing, as otherwise unrelated transactions may conflict and roll back due to hash collisions. Other implementations [15] achieve better performance via explicit annotations on private and shared memory operations, but this eliminates much of the promised simplicity and transparency of STM.

Many proposals exist for hardware-assisted transactional memory (HTM), aiming to solve the performance problems of software implementations. Thus far the necessary support has failed to materialize on general-purpose hardware, with the death of Sun's planned "Rock" processor, but Intel's "Transactional Synchronization Extensions" (TSX) on Haswell and newer processors may change that. However, even with the necessary hardware support, few software projects could afford to tie themselves to the availability of HTM on the target platform, necessitating STM as a fallback mechanism for portability. STM would also remain critical to support complex transactions that exceed the fixed limits supported by a hardware implementation. Even if transaction-based software carefully remained within the limits of the hardware's capabilities, any composition of transactions could create a single transaction that requires software fallbacks to handle. Thus, even with a hypothetical HTM system, STM's performance problems would still severely hamper the adoption of transactional memory.

Many HTM implementations also introduce new forms of spurious failures, analogous to the false sharing of hash-based software transactions. Hardware transactions based on cache coherence can fail due to cache line conflicts in a non-fully-associative cache. Furthermore, checking for conflicts at a cache line granularity results in the classic false sharing problem when accessing different fields in the same cache line.

The apparent generality and simplicity of applying transactional memory remains its strongest attribute. However, that generality and simplicity

may simply reflect a lack of application to many real-world systems, and indeed transactional memory systems applied to real systems do grow in complexity due to special cases and unanticipated needs [53]. Even such simple additions as performing I/O or similar irreversible operations can lead to problems when rolling back transactions; attempted solutions such as "inevitable transactions" [72] or "irrevocable transactions" [81] introduce additional complexity, overhead, and scalability problems. Preserving generality will continue to require further extensions, with corresponding erosion of simplicity and scalability.

Even the disjoint-access parallelism that provides TM's primary performance benefits itself incurs a severe cost. Attiya, Hillel, and Milani [5] proved that any transactional memory system providing disjoint-access parallelism cannot avoid having readers perform store operations to shared memory. In fact, they showed that a transaction loading t items has a strict lower bound of $\Omega(t)$ required store operations in its implementation. Eliminating these store operations requires giving up disjoint-access parallelism, the primary motivation for TM over simple coarse-grained locking.

With or without its automatic fine-grained disjoint-access parallelism, transactional memory fails to offer any solution for concurrent, non-disjoint accesses. Some implementations of transactional memory may offer concurrent readers, but no implementations allow readers to run concurrently with writers; either the readers will force the writer to roll back or vice versa. Thus, like mutual exclusion and NBS, transactional memory devolves to mu-

tual exclusion in the face of non-disjoint accesses.

2.4 Partitioning

As communication costs grow between CPUs, the lessons of distributed systems become more relevant. Modern shared-memory multiprocessor systems internally look like distributed systems with explicit communication, but provide the illusion of a coherent shared memory through cache coherence protocols.

In an effort to embrace this model, some shared-memory algorithms designate a single thread as the owner of certain resources, and force all other threads to access those resources by sending a message to the owning thread. If these resources prove amenable to strict partitioning, and threads mostly access their own resources, this results in an embarrassingly parallel algorithm with excellent scalability. However, in the common case where resources do not partition well, this merely replaces the negative scalability caused by contention with zero scalability caused by running all operations on a single processor.

2.5 Hazard Pointers

Hazard pointers [58, 3] provide one means of surpassing disjoint-access parallelism. To maintain reader safety with concurrent writers, readers must store pointers they wish to dereference in their local hazard pointers, and

writers must check all hazard pointers before performing an operation that might cause such a dereference to fail. This approach does allow readers to run concurrently with writers, but it forces readers to perform writes, and furthermore forces readers to perform expensive memory barrier operations to preserve the ordering between the hazard pointer manipulations and the code they protect.[1]

2.6 Read-Copy Update

An ideal technique for concurrent algorithm design should go beyond disjoint-access parallelism. Scalable readers should run concurrently with writers, and in particular, neither should block the other from proceeding. This concurrency should apply even for concurrent accesses to the same location.

Deferred destruction techniques such as Read-Copy Update (RCU) [47, 23, 56] allow algorithms to surpass the limited concurrency of disjoint-access parallelism. These techniques provide a lightweight means for readers to delimit their critical regions, using only inexpensive, non-synchronizing, CPU-local operations. RCU-based writers then have an operation to wait for all current readers to finish.

Section 2.6.1 discusses the RCU primitives in detail. Section 2.6.2 demonstrates how to construct RCU-based data structures using these primitives. Section 2.6.3 compares RCU to other concurrent programming techniques,

[1] Some hazard pointer implementations exist that avoid expensive operations in readers; these implementations incorporate techniques from deferred destruction methods such as RCU.

and discusses the key problems hampering widespread adoption of RCU. Section 2.6.4 discusses implementation considerations for the RCU primitives.

2.6.1 RCU Primitives

RCU readers execute concurrently, both with each other and with writers, and thus readers can potentially observe writers in progress [47]. (Other concurrent programming models prevent readers from viewing intermediate memory states via locking or conflict detection.) The methodologies of RCU-based concurrent programming, as originally presented by McKenney [47], 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 must avoid exposing inconsistent intermediate states from writers.

RCU writers typically 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: 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. Current implementations of RCU provide a publish function

named `rcu_assign_pointer`; this function executes a hardware store barrier before assigning the pointer, which ensures that readers that see the published pointer will see the preceding initialization of the memory accessed through that pointer. Readers may also require compiler directives to prevent certain aggressive optimizations across the pointer dereference; RCU wraps those directives into a *read* primitive, commonly provided as a function `rcu_dereference`.[2]

These primitives 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 [47]. 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* (typically invoked via a function named `synchronize_rcu`), which blocks until all readers that started before the barrier have completed [47]. 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*

[2]On certain obsolete architectures, such as the DEC Alpha, readers must also use a memory barrier to prevent reordering by the CPU [13].

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 the memory a waiting writer wants to reclaim, 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 approach allows readers to avoid the synchronization overhead of tracking individual references to shared data, making those readers more efficient and scalable; instead, readers delimit themselves with lightweight CPU-local operations, typically called `rcu_read_lock` and `rcu_read_unlock` [47].

Writers often use wait-for-readers solely for safe memory reclamation. Because memory reclamation operations can safely occur concurrently and need not occur immediately (given sufficient memory), RCU implementations also provide an asynchronous wait-for-readers callback, `call_rcu` [47].

Chapter 5 documents the translation from the relativistic causal ordering model to hardware ordering models, which uses the wait-for-readers (`synchronize_rcu`) barrier, and hardware load and store barriers based on those used in `rcu_dereference` and `rcu_assign_pointer`. The use of load and store barriers in relativistic causal ordering differs from the RCU primitives in two key ways. First, a load barrier only translates to `rcu_dereference` in the case of a dependent load. Independent loads require hardware load barriers, which RCU does not provide a primitive for; RCU algorithms rely on existing portable memory-barrier interfaces in the occasional cases where they need hardware load barriers, and relativistic causal ordering

uses these same portable interfaces. Second, RCU's `rcu_assign_pointer` primitive incorporates both a store barrier and a subsequent store operation; the definition and translation of relativistic causal ordering uses explicit store barrier steps between store operations, to allow for the possibility of partially ordered store operations in which the store barrier does not naturally attach to any particular store operation. Nonetheless, the concrete implementation of a relativistic causal algorithm can choose to use `rcu_assign_pointer` to group a store operation with its preceding store barrier, rather than invoking a portable store-barrier abstraction directly.

2.6.2 RCU Data Structures

The simplest RCU data structures follow the basic pattern introduced in RCU-based linked lists [47]. RCU linked-list traversal uses the standard linked-list traversal algorithm, but adds reader delineation and `rcu_dereference`:

1. Call `rcu_read_lock` to start an RCU read-side critical section.
2. Walk the nodes from the head pointer to the terminating null, calling `rcu_dereference` to dereference each pointer.
3. Call `rcu_read_unlock` to end the RCU read-side critical section.

To insert a node, first create and initialize it, including its pointers into the data structure, and then modify an existing link from the data structure to point to it:

1. Allocate memory for a new list node.

2. Initialize the contents of the new node.

3. Acquire a lock, or otherwise synchronize with other writers. (This does not exclude readers, nor do readers exclude writers, as the readers do not acquire any lock.)

4. Traverse the list to find the insertion point.

5. Initialize the next pointer of the new node to point to the node which will follow it in the list.

6. Use `rcu_assign_pointer` to publish the new node by pointing the previous node's next pointer (or the head pointer if inserting at the beginning) to that node. The `rcu_assign_pointer` primitive ensures that readers cannot observe the publication of the node in this step without observing the initialization of the node.

7. Release the lock, or otherwise synchronize with other writers.

See figure 2.1 for an example of the RCU linked-list insert operation.

To remove a node, modify the link from the data structure to that node to route around it, then wait for all current readers to finish, and then reclaim the memory of the node:

1. Acquire a lock, or otherwise synchronize with other writers. (Again, this does not exclude readers, nor do readers exclude writers.)

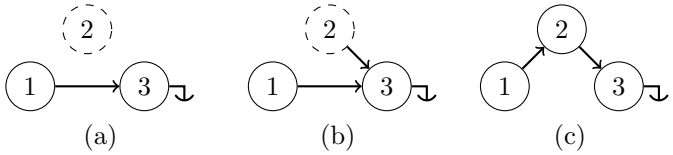

Figure 2.1: Insertion into a relativistic linked list. 2.1a shows the initial state of the list. Dashed nodes exist only in writer-private memory, unreachable by readers. The writer wants to insert node 2. The writer first initializes node 2's next pointer to point to node 3, resulting in 2.1b. The writer can then publish point node 1's next pointer to 2 using `rcu_assign_pointer`, and readers can then immediately begin observing the new node, as shown in 2.1c.

2. Set the next pointer of the previous node (or the head pointer if removing the first node) to point to the same place as the next pointer of the node to remove. Note that this store operation does not use `rcu_assign_pointer`, because it has no prior store operation to order with.

3. Release the lock, or otherwise synchronize with other writers.

4. Use `synchronize_rcu` to wait for all current readers to finish.

5. Reclaim the memory associated with the node.

See figure 2.2 for an example of the RCU linked-list remove operation.

2.6.3 RCU Discussion

In a data structure based on mutual exclusion, writers and readers must both wait for each other when accessing the same data; disjoint-access parallelism

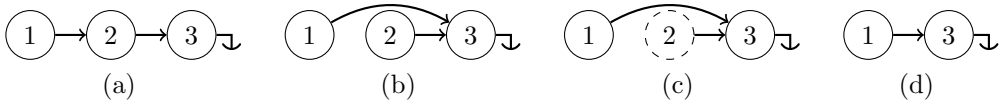

Figure 2.2: Removal from a relativistic linked list. 2.2a shows the initial state of the list. The writer wants to remove node 2. The writer first sets node 1's next pointer to node 3, removing node 2 from the list for all future readers; this results in the state shown in figure 2.2b. The writer then uses `synchronize_rcu` to wait for existing readers to finish, as shown in figure 2.2c. Dashed nodes exist only in writer-private memory, unreachable by readers. Once no readers can hold references to node 2, the writer can safely reclaim it.

permits concurrent access to disjoint parts of the structure, but access to the same data requires serialization. In a data structure based on RCU, readers can always proceed without waiting; readers run concurrently with writers, and vice versa. All waiting occurs in writers; writers wait not for an absence of readers, but for the completion of readers that would otherwise observe store operations in an unsafe order. A writer need not wait for readers unless it needs to complete a disruptive operation such as reclaiming memory. Furthermore, the writer can often defer such disruptive operations until readers have finished, while still proceeding with the non-disruptive operations it intended.

In sharp contrast to locking, non-blocking synchronization, and transactional memory, RCU readers perform no expensive synchronization operations whatsoever: no locks, no atomic operations, and no compare-and-swap. RCU readers typically incur little to no overhead even compared to concurrency-unsafe single-threaded implementations; furthermore, by avoid-

ing expensive synchronization, RCU readers avoid the need for communication between threads, allowing wait-free operation and excellent scalability. Implementations of RCU-based algorithms exist for various data structures, including linked lists [47], hash tables [77, 76, 78], radix trees [63], and balanced trees [34]. Each of these implementations demonstrates large scalability improvements over alternative implementations based on locking.

However, each new data structure implemented using RCU has required significant engineering effort to invent. Worse, even with the existing body of implemented data structures, algorithms for new data structures prove no easier to implement. No general construction techniques have arisen to make the process more mechanical. Furthermore, no strong mental model exists for reasoning about the correctness of such algorithms beyond intuition and individual argumentation, both notoriously prone to error or omission.

In the past, RCU adoption has also suffered greatly from the lack of a generally available implementation outside those in the Linux kernel. This gave RCU the appearance of a specialized technique without more general application. Furthermore, the first implementations of RCU relied on optimizations specific to the internals of an operating system kernel, such as the ability of kernel code to mark itself temporarily non-preemptible; this allowed the RCU implementation to work in terms of CPUs rather than threads. While this helped minimize RCU's read-side overhead, it also lent further credence to the impression of non-portability.

However, recent developments have lifted these technical limitations. Math-

ieu Desnoyers provided several scalable implementations of RCU for user processes, as part of the Userspace RCU (liburcu) project [10]. While some of these implementations take advantage of Linux-specific features for performance, the most straightforward implementations provide portability to other software platforms. This positions RCU as a broadly applicable technique for scalable algorithm design, limited primarily by two remaining issues, both related: no general technique to construct algorithms for arbitrary data structures, and no ordering model to allow straightforward reasoning about the correctness of those algorithms.

2.6.4 RCU Implementation Considerations

This section summarizes the practical implementations of the RCU wait-for-readers primitive (`synchronize_rcu`) and the corresponding read-side delineation (`rcu_read_lock` and `rcu_read_unlock`). While the semantics of these fundamental primitives of RCU have remained consistent throughout its lifetime, the implementations of RCU in current use remain under active development, with frequent optimizations and algorithmic improvements. Thus, this section serves primarily as an overview of practical implementations, rather than a full account of the details required to implement them. For those details, see published papers on RCU [47, 56, 54, 23, 10, 20, 49], as well as the RCU implementations in Linux and liburcu.

All RCU implementations operate by noting *quiescent states*: points in time during which no readers currently run on a particular CPU or thread [47,

23, 20, 49].[3] The implementation of `synchronize_rcu` waits for a quiescent state to occur on all CPUs or threads (depending on implementation); because a quiescent state indicates the absence of a reader, once a quiescent state has occurred on all CPUs or threads, any previously running readers must have completed.

An implementation of RCU without regard for reader synchronization costs could simply track the quiescence of readers individually. For example, `rcu_read_lock` and `rcu_read_unlock` could set a flag in shared memory which indicates the presence of a reader on the reader's current CPU or thread, and `synchronize_rcu` could wait for all flags to clear [20]. To prevent writer starvation, `synchronize_rcu` could swap between two sets of reader flags, and readers would always clear the flag they originally set [20]; this allows `synchronize_rcu` to wait on existing readers but not new readers. However, maintaining these shared flags consistently would require global synchronization between readers and writers, imposing large synchronization costs on readers.

To avoid these synchronization costs, RCU implementations typically amortize any costs associated with tracking a quiescent state over all the readers which have run since the last quiescent state. In particular, the total cost of tracking quiescent states does not increase with the addition of readers. Thus, individual RCU readers can avoid synchronization overhead

[3]Kernel implementations of RCU can use optimized implementations based on CPUs, making use of the kernel's control over the scheduling of threads on CPUs; userspace implementations typically operate on threads.

and the corresponding scalability limitations, increasing read throughput by increasing the latency of `synchronize_rcu`. In the case of RCU-based memory reclamation, amortization causes writers to retain memory longer before reclaiming it, increasing read throughput at the expense of higher memory usage.

The earliest and simplest implementations of RCU in Linux, "Classic RCU", take advantage of the ability of kernel code to temporarily disable preemptive scheduling on the current CPU. Thus, a CPU running in the scheduler cannot have a reader active, making the scheduler a quiescent state. In these implementations, `synchronize_rcu` simply blocks until all CPUs have passed through the scheduler.

The classic RCU implementation does not work well on low-latency or real-time systems, due to its implementation via disabled preemption [20]. Such systems use *preemptible* RCU implementations, which track quiescent states by having `rcu_read_lock` and `rcu_read_unlock` manipulate private, CPU-local counters [20]. Preemptible RCU amortizes the synchronization cost of communicating those CPU-local counters to writers, rather than doing so immediately after each reader completes. Preemptible RCU implementations also track RCU readers which get preempted separately from their tracking of quiescent states, as these preempted readers must complete before `synchronize_rcu` can return. The addition of preemption also introduces the possibility of priority inversion between readers and writers; thus, preemptible RCU implementations include priority boosting for preempted

RCU readers.

As the number of CPUs in a system grows, `synchronize_rcu` has to do more work to track quiescent states on every CPU, introducing scalability limitations in writers and quiescent states on highly parallel systems (in the hundreds of CPUs). To overcome these scalability limitations, *hierarchical* RCU implementations [49] organize CPUs into trees, with a word-sized bitmask at each level of the tree tracking the quiescence of all the nodes below. This tree structure decreases contention between CPUs communicating their quiescence, limiting the synchronization required for such communication to small groups of CPUs rather than all CPUs on the system.

To support scaling in the other direction, *tiny* RCU [50] optimizes for single-CPU systems. Because RCU readers cannot block, and `synchronize_rcu` blocks until readers have finished, a call to `synchronize_rcu` itself indicates a quiescent state on the sole CPU, and thus `synchronize_rcu` can return immediately, with only a compiler barrier to preserve ordering.

Chapter 3

Manual Construction of Scalable Concurrent Data Structures

Hash tables enjoy widespread use in many applications and operating systems, due to their O(1) average time for many operations [9, 36]. These users of hash tables have increasingly become concurrent, to adapt to concurrent hardware. Thus, hash table implementations should support concurrent operation, and should ideally scale linearly with additional CPUs. In particular, many hash table applications involve far more lookups than modifications [28]; such applications require fast, scalable lookups.

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

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

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

This hash-table move operation serves not only as a novel contribution in itself, but also as an example of the manual process for constructing scalable algorithms and reasoning about their correctness. While this algorithm works correctly, and serves the intended purpose, it required complex reasoning both to generate and to verify. This complex reasoning motivates the relativistic causal ordering model presented in chapter 4.

Section 3.1 documents the semantics the move operation must satisfy, and the semantics of hash-table lookups that make the move operation possible. Section 3.2 provides the full algorithm for the new move operation, including step-by-step diagrams of the hash table structure. Section 3.3 outlines the methodology for performance analysis, and section 3.4 presents the results of this analysis. Section 3.5 summarizes the conclusions.

3.1 Semantics

Section 3.1.1 lists required properties of the move operation. Section 3.1.2 documents some standard properties of hash-table lookups that support the

new move operation.

3.1.1 Properties of Hash-Table Moves

The move operation changes the key associated with an entry, and moves the entry to the hash bucket corresponding to the new key. A move operation must allow concurrent lookups, and must maintain the consistency properties expected by those lookups:

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

"Subsequent lookup ordered after the first" means either a lookup running in the same thread as the first but later in program order, or a lookup equivalently ordered after the first via some appropriate synchronization.

The first two properties originally arose through reasoning about the use of concurrent hash tables for directory entry lookups in an operating system kernel, and the observable effects this could have for userspace programs. The first property guarantees that, during a move operation, once a program has observed the file in its new location, it cannot subsequently observe the file still present in its old location. The second property guarantees that once

a program has observed the absence of the file in the old location, the file must appear in its new location.

Note that an implementation of the move operation composed of a linked-list insert and remove operation (as specified in section 2.6.2) cannot satisfy these consistency properties. Performing an insert followed by a remove would violate the first property by allowing the node to temporarily appear under both the old and the new keys. Performing a remove followed by an insert would violate the second property by allowing the node to temporarily appear under neither key.

3.1.2 Properties of Hash-Table Lookups

The new hash table move operation relies on two fundamental properties of a hash table lookup.

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

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

for a different key. Thus, multiple nodes with the same key can safely appear in a given hash bucket. Note that this requirement means that the hash table cannot safely hold multiple distinct entries with the same key, such as in the implementation of a multimap.

The first two possible semantics violations from section 3.1.1 (entries appearing in neither bucket or appearing in both buckets) occur when the writer does not simultaneously remove the node from the old bucket and add it to the new bucket with the new key. Most modern architectures do not feature memory-to-memory swaps, simultaneous store operations to multiple locations, or hardware transactional memory, so the writer cannot simultaneously and atomically change more than one pointer or key. Those architectures that do, or software systems such as software transactional memory that simulate such capabilities, incur a high cost for such an operation [7, 66, 80, 11]. Furthermore, even an atomic memory-to-memory swap operation or similar multi-store operation would not by itself prevent readers from seeing zero or two copies of the moved entry, without also involving the readers in some form of transactional system or otherwise handling the interleaving of load and store operations.

3.2 Algorithm

Section 3.2.1 describes the fundamental step in the new move algorithm. Section 3.2.2 outlines the hash-table lookup operation. Section 3.2.3 walks through the new move algorithm step-by-step. Section 3.2.4 discusses the

correctness of this algorithm in terms of the required semantics from section 3.1.1.

3.2.1 Atomic Rename via Cross-Linking

The new hash-table node move operation builds on a single fundamental insight: if the writer can make the moving node appear in both buckets simultaneously, it can in one operation remove the node from the old bucket and add it to the new bucket, by atomically changing the key. Before the change, searches in the old bucket using the old key will find the node, and searches in the new bucket using the new key will always skip over it; after the change, searches in the old bucket with the old key will always skip over the node, and searches in the new bucket with the new key will find it. This approach satisfies the key semantics for the move operation.

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

To safely resolve the cross-link, the algorithm makes use of a deferred destruction technique such as Read-Copy Update (RCU); specifically, the algorithm requires the `synchronize_rcu` or `call_rcu` primitives documented

in section 2.6.1. Deferred destruction removes one source of conflicts between readers and writers, by separating memory reclamation from writers and deferring that reclamation until readers have finished. Writers can thus focus on maintaining higher-level semantics such as those in section 3.1.1, rather than on preventing readers from crashing.

3.2.2 Hash-Table Lookup

The lookup operation consists of a standard uniprocessor hash-table lookup, except that it makes use of the appropriate RCU primitives to support deferred destruction and enforce correct memory ordering:

1. Hash the given key to determine the corresponding hash bucket.

2. Call `rcu_read_lock` to start an RCU read-side critical section.

3. Traverse the linked list in that hash bucket, comparing the given key to the key in each node. Each pointer dereference in this traversal must use `rcu_dereference`.[1]

[1] On architectures such as DEC Alpha that do not automatically guarantee memory ordering for dependent reads [13], `rcu_dereference` includes an appropriate barrier to order such reads, such as `smp_read_barrier_depends` on Linux. However, almost all current multiprocessor architectures provide dependent read ordering by default, and thus on these modern architectures `rcu_dereference` does not include a hardware memory barrier.

Aggressive compiler optimizations, particularly those regarding local caches of global data, can also lead to problems in this step [6]. This may necessitate compile-time barriers to locally prevent such optimizations, and `rcu_dereference` includes any such compile-time barriers.

3.1 If a node has the given key, proceed with the computation that required the node.

3.2 If the traversal reaches the end of the list without finding a node with the given key, the node does not exist in the table.

4. Call `rcu_read_unlock` to end the RCU read-side critical section.

Note that all of these steps allow implementations of this lookup algorithm to avoid expensive synchronization operations such as locks or atomic operations. As discussed earlier in section 3.2.1, the uses of RCU primitives do not require expensive synhronization operations. The list traversal in step 3 relies on the property that reads and writes to word-sized word-aligned locations such as pointers will occur atomically, retrieving either the old or the new value but not a mix of the two; this property holds on all current architectures.

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

3.2.3 Hash-Table Move

Figure 3.1 shows a sample configuration of a hash table, used to illustrate the move algorithm. The move algorithm operates by moving the node to the end of its bucket, cross-linking the two buckets to both contain that node, changing the key, and safely resolving the cross-link. The following steps walk through the move algorithm on the hash table shown in figure 3.1:

1. Perform the appropriate synchronization to modify hash buckets a and b. For instance, obtain the locks for hash buckets a and b, in hash bucket order to avoid deadlocks. Note that this step only exists to synchronize with other concurrent writes, not with lookups.

2. Make a copy of the target node n_2; call the copy n_2'.

3. Set $n_2'.next$ to NULL.

4. Set $n_3.next$ to n_2' using `rcu_assign_pointer`; `rcu_assign_pointer` ensures that the new value of $n_2'.next$ will become visible to other processors before n_2' does.

5. Remove n_2 from a by pointing $n_1.next$ to n_3 using `rcu_assign_pointer`. a now has the target node n_2' at the end. `rcu_assign_pointer` ensures that n_2' will become visible to other processors before n_2 disappears.

6. Point the tail of bucket b ($n_5.next$) to the new target node (n_2'). Both hash bucket chains now include n_2'. Figure 3.2 shows the state of the

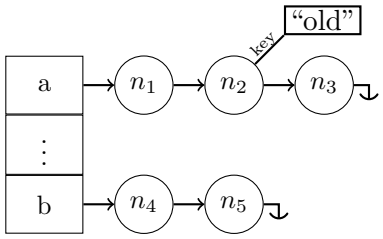

Figure 3.1: Initial hash table configuration used to illustrate move algorithm. $n_1.key$, $n_2.key$, and $n_3.key$ hash to a. $n_4.key$ and $n_5.key$ hash to b. The move operation will change $n_2.key$ from "old" to "new". "new" hashes to b.

hash table after this step.

7. Atomically change $n_2'.key$ to "new" using `rcu_assign_pointer`. `rcu_assign_pointer` ensures that the removal of n_2 and the cross-linking will become visible before $n_2'.key$ changes.

8. Point $n_3.next$ to null using `rcu_assign_pointer`, un-cross-linking the chains. `rcu_assign_pointer` ensures that the change to $n_2'.key$ will become visible before n_2' disappears from bucket a. Figure 3.3 shows the state of the hash table after this step.

9. Release the write-side synchronization for hash buckets a and b.

10. Using `call_rcu`, asynchronously remove the original n_2 and the old key "old" after all current readers have finished. (Alternatively, this step may occur synchronously using `synchronize_rcu`.)

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

3.2.4 Discussion

These operations meet the required semantics described in section 3.1.1. First, "If a lookup finds the item under the new key, a subsequent lookup

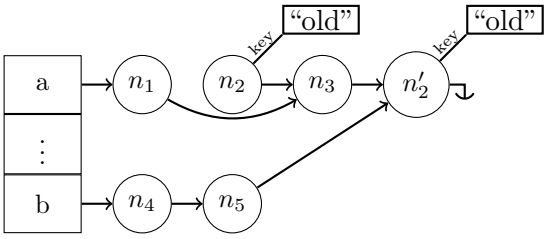

Figure 3.2: State of the hash table after cross-linking hash chains in step 6 of the relativistic hash table move algorithm.

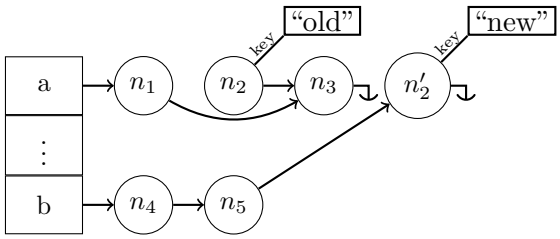

Figure 3.3: State of the hash table after un-cross-linking hash chains in step 8 of the relativistic hash table move algorithm.

ordered after the first cannot find the item under the old key." Suppose a reader finds the item under the new key. It must find n'_2, because $n_2.key$ never changes. The writer writes the new key in step 7, so the reader must observe the result of this step. To subsequently find an item under the old key, the reader must find n_2, because n'_2 no longer has the old key. To find n_2, the reader must not see the change to $n_1.next$ in step 5 removing it. However, the use of `rcu_assign_pointer` in step 7 ensures that a reader cannot see the result of step 7 and not step 5.

Second, "If a concurrent lookup does not find the item under the old key, a subsequent lookup ordered after the first must find the item under the new key." Suppose a reader does not find the item under the old key. It must not

see n_2, and it must not see n_2' before its key changes. Because it does not see n_2, it must see the result of step 5. Because it does not see n_2', it must either see the result of step 7 or not see the result of step 4. Because the reader saw the result of step 5, the use of `rcu_assign_pointer` in step 5 ensures that the reader must see the result of step 4, and therefore the reader must see the result of step 7. However, if the reader sees the result of step 7, it will find n_2' with the new key on a subsequent lookup.

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

3.3 Performance Analysis Methodology

The lookup algorithm requires no synchronization instructions, and runs wait-free, even when running concurrently with a move operation. Thus, it should allow significantly more lookup throughput than a lock-based lookup operation. The move algorithm performs four `rcu_assign_pointer` oper-

ations, a memory allocation, and either a `call_rcu` or `synchronize_rcu` operation, as well as various additional non-synchronizing operations. This should result in lower move throughput than a lock-based move operation. Testing these hypotheses required a new benchmark framework for concurrent hash tables.

The new lookup and move operations make extensive use of RCU. The Linux kernel contains several mature and widely used implementations of RCU, as well as implementations of the standard forms of mutual exclusion, and a fast concurrent memory allocator. Thus, a Linux kernel module provided the most practical and straightforward target for a benchmark.

The `rcuhashbash` benchmark module implemented for this chapter consists of two components: a set of concurrent hash table implementations with a common interface, and a test harness that invokes the hash table operations and tracks statistics. `rcuhashbash` includes the following hash table implementations:

- The move and lookup algorithms presented in this chapter, as described in sections 3.2.2 and 3.2.3. This implementation uses per-bucket spinlocks to synchronize with other writers. The lookup operation contained no synchronization operations of any kind.

- Multiple variants of mutual exclusion: whole-table spinlocks, whole-table reader-writer locks, per-bucket spinlocks, and per-bucket reader-writer locks.

- The RCU-based algorithm currently used for the Linux directory entry cache (dcache) [55, 41]. This algorithm uses an RCU-based linked list to allow concurrent insertions and deletions without disrupting readers. However, the lookup operation uses an optimistic *sequence lock* [37] to detect concurrent moves, and retries a lookup if it raced with a move; this sequence lock entails some expensive synchronization operations.

`rcuhashbash` begins by constructing a hash table of a specified size, and loading it with integers from 0 to a specified maximum. (These integers effectively serve as both keys and values.) The experiments in this chapter used a hash table with 1024 buckets and 4096 entries. `rcuhashbash` then spawns a specified number of threads at startup. Each thread goes into a continuous loop, randomly choosing to lookup or move based on a specified reader/writer ratio. The move operation randomly chooses an old key and a new key from the range of 0 to twice the maximum initial value ($[0, 8191]$ for this experiment); it then attempts to move the item with the old key to the item with the new key. The lookup operation randomly chooses a key from the same range and performs a lookup. The lookup and the move operation each increment a thread-local count of the number of operations completed.

The machine used for testing had 16 IBM POWER6 processors at 4.7GHz, each with two cores of two logical threads each, for a total of 64 hardware-supported threads (henceforth referred to simply as "CPUs"). This machine ran the Linux 2.6.28 kernel, compiled for the 64-bit powerpc architecture, using the "classic" RCU implementation [54] and no preemption. To observe

scalability, the benchmark ran each hash table implementation on 1, 2, 4, 8, 16, 32, and 64 CPUs. To obtain enough samples for statistical analysis, the benchmark ran each implementation 10 times, for 30 seconds each time. To observe the effect of a varying read to write ratio, each implementation ran with the read to write ratio set to 999999:1, 999:1, and 1:1.

3.4 Performance Analysis

Section 3.4.1 gives the performance results for the hash-table lookup operation, and section 3.4.2 gives the results for the move operation. Section 3.4.3 summarizes these results.

3.4.1 Hash Lookup Performance

Figures 3.4, 3.5, and 3.6 show the average number of lookups in 30 seconds for each hash table implementation as the number of CPUs used increases; the three figures depict the three decreasing read to write ratios.

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

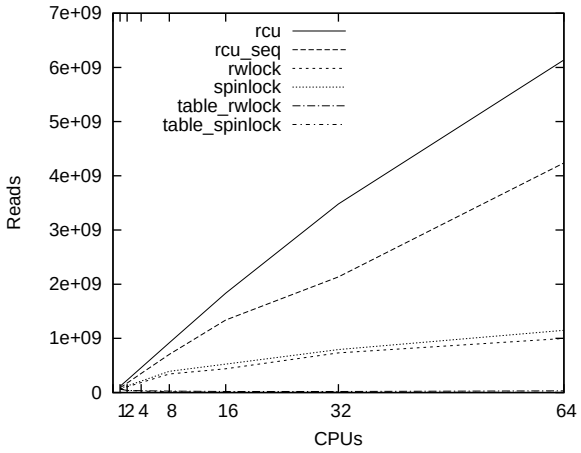

Figure 3.4: Average lookups in 30 seconds by number of CPUs with 999999:1 read:write ratio

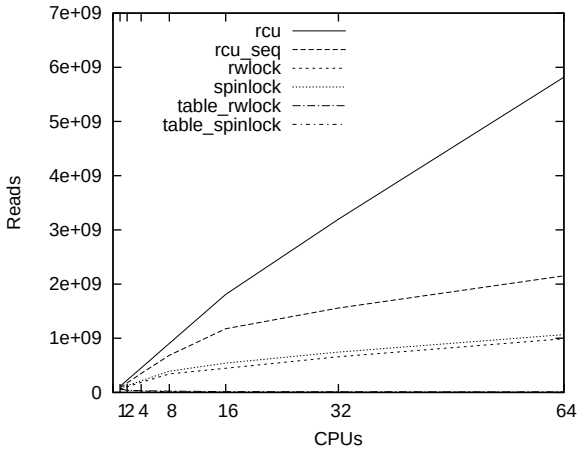

Figure 3.5: Average lookups in 30 seconds by number of CPUs with 999:1 read:write ratio

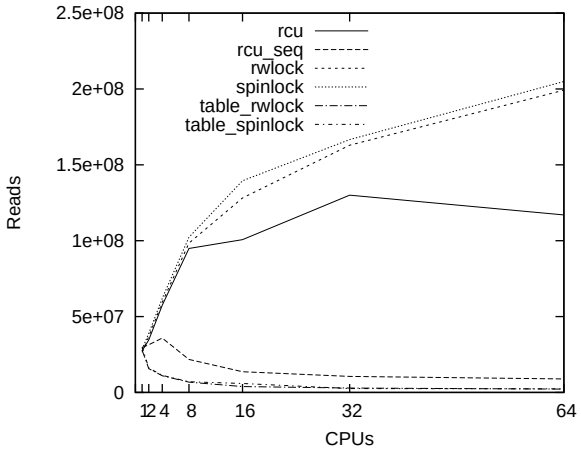

Figure 3.6: Average lookups in 30 seconds by number of CPUs with 1:1 read:write ratio

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

At all three read to write ratios, per-bucket spinlocks outperform per-bucket reader-writer locks, even on the full 64 CPUs. Reader-writer locks have a higher critical section overhead than ordinary spinlocks, and for small critical sections this overhead nullifies the benefits of concurrent readers.

3.4.2 Hash Move Performance

Figures 3.7, 3.8, and 3.9 show the average number of moves in 30 seconds for each hash table implementation as the number of CPUs used increases; the three figures again depict the three decreasing read to write ratios: 999999:1, 999:1, and 1:1.

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

For the balanced 1:1 read to write ratio, per-bucket mutual exclusion takes a healthy lead, with spinlocks still winning over reader-writer locks. However, even for this workload, the proposed algorithm scales far better than the sequence-lock-based algorithm used in the Linux kernel.

3.4.3 Performance Summary

The proposed hash table move algorithm provides marked performance and scalability advantages compared to the current state of the art used in the

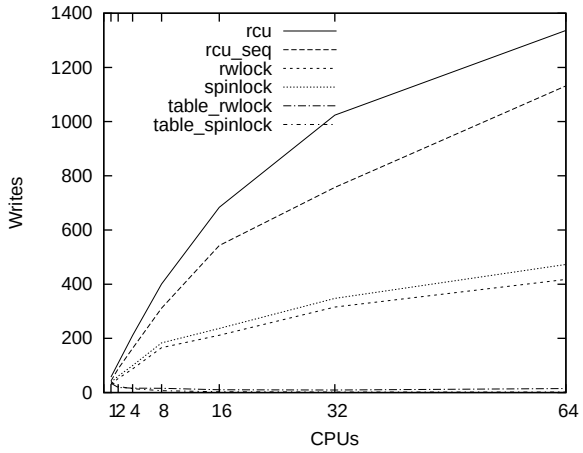

Figure 3.7: Average moves in 30 seconds by number of CPUs with 999999:1 read:write ratio

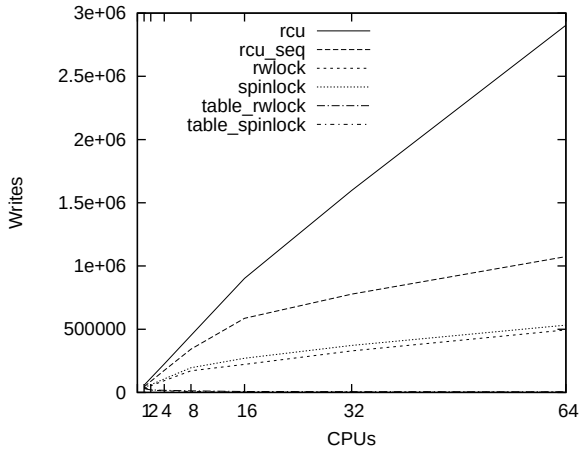

Figure 3.8: Average moves in 30 seconds by number of CPUs with 999:1 read:write ratio

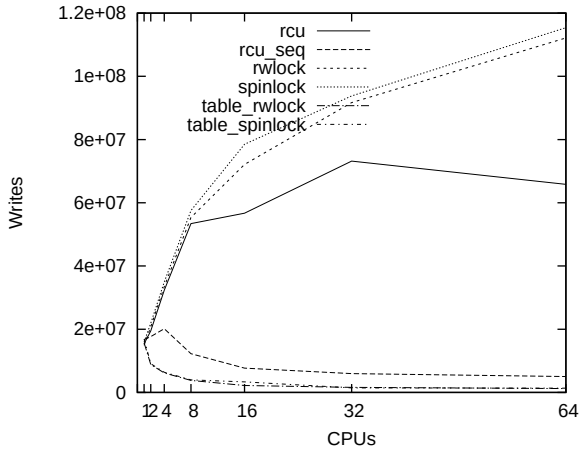

Figure 3.9: Average moves in 30 seconds by number of CPUs with 1:1 read:write ratio

Linux kernel. It proves the clear winner for read-biased workloads, and degrades gracefully for balanced workloads.

3.5 Summary

I have presented novel algorithms for a concurrent hash table, supporting a semantically atomic move operation based on cross-linking hash chains. Benchmarks of the proposed concurrent hash table implementation demonstrated a 6x scalability improvement for lookups versus fine-grained locking, and a 1.5x improvement versus the current state of the art in Linux. Read-biased workloads provided the highest scalability and performance, but the algorithm remained competitive even for balanced workloads.

The hash table move operation allows fast scalable lookups by making it possible for those lookups to avoid all expensive synchronization operations.

This operation removes conflicts between readers and writers by making updates appear semantically consistent after every visible step, and by deferring destruction to maintain referential integrity.

Furthermore, the lookup algorithm involves no helping, rollback, or retry code. In addition to improving performance, this gives lookups a deterministic response time, making them more suitable for use in code requiring real-time response. This deterministic behavior assumes an appropriate real-time implementation of deferred destruction, such as a real-time RCU implementation [20]. Similarly, the move algorithm does not wait on concurrent readers; memory reclamation must wait until readers have completed, but this can occur asynchronously as long as the system has sufficient memory.

This approach provided an example of the broader class of concurrent programming techniques and data structures based on relativistic programming, which share the common theme of allowing additional parallelism by permitting concurrent access to shared data without a critical section. These techniques use a copy-based update strategy to allow readers and writers to run concurrently without conflicts, avoiding many of the non-scalable costs of inter-processor communication, cache coherence, and synchronization.

However, allowing one version of a structure to be read concurrently with updates to a different version of the same structure may permit weaker memory-ordering behavior than normally expected by readers. For example, a thread may walk a linked list concurrently with a sequence of insertions, and observe a set of items that does not correspond to any state the list

passed through as a result of those insertions: it may see items inserted later in time (from the perspective of the thread performing the insertions) without seeing items inserted earlier.

Algorithms designed to work with relativistic programming must either tolerate this weakened memory ordering, or take steps to ensure that the data structure appears to change directly from one semantically consistent state to another without inconsistent intermediate states. The hash-table move operation described in this chapter implements the latter approach to preserve the semantics described in section 3.1.1.

However, the algorithm documented in this chapter required complex data-structure-specific reasoning to argue for its correctness. The development of relativistic algorithms for other use cases currently requires the same level of complex data-structure-specific reasoning. To ease the adoption of relativistic programming techniques, chapter 4 presents the relativistic causal ordering model, a generalized memory model supporting the construction of algorithms for a broad class of data structures without the need for data-structure-specific reasoning.

Chapter 4

The Relativistic Causal Ordering Model

Section 4.1 outlines the design constraints and assumptions supporting the relativistic causal ordering model, including the minimal functionality expected from the underlying shared-memory system. Section 4.2 provides terminology supporting the definition of the relativistic causal ordering property. Section 4.3 defines the relativistic causal ordering property itself. Section 4.4 defines abstract versions of the barrier operations used to enforce the relativistic causal ordering property on weakly ordered hardware models. Section 4.5 shows how to construct algorithms based on relativistic causal ordering, and provides specific rules for placing the abstract barrier operations to enforce this ordering. Section 4.6 demonstrates the relativistic causal ordering property by using it to reconstruct the RCU linked-list algorithms from section 2.6.2.

4.1 Design Constraints and Assumptions

Memory consistency models play a central role in reasoning about any concurrent program or programming methodology based on shared memory. Adve and Boehm [1] showed that the choice of memory model determines fundamental properties of a concurrent programming environment, and makes a fundamental tradeoff between scalability and ease of programming. Concurrent hardware and software can scale most effectively with less enforcement of memory ordering, while the simplest implementation techniques rely on stricter ordering models [1, 4, 30].

The relativistic causal ordering model acknowledges that real hardware systems do not necessarily provide strict memory ordering models by default. While such systems typically provide all the necessary primitives (such as memory barriers) to enforce strict ordering models, those primitives require explicit use in concurrent algorithms to enforce the desired ordering model. As previously discussed, those primitives also result in many of the scalability limitations that plague concurrent algorithms; thus, the relativistic model minimizes their use. Furthermore, since the manual placement of those primitives makes the construction of scalable algorithms complex, the relativistic model systematizes the placement of these barriers, allowing the construction of scalable concurrent algorithms without regard for the underlying hardware model.

The design of the relativistic causal ordering model applies the end-to-end

principle [67], by specifying the minimal behavior expected from the underlying system, and then specifying consistency properties as features provided by the relativistic causal ordering model rather than as demands placed on the underlying system. The barrier insertion rules defined in section 4.5, together with the translation to hardware memory models defined in chapter 5, separate the mechanical process of enforcing relativistic causal ordering from the task of designing algorithms relying on relativistic causal ordering. This allows relativistic algorithms to assume the consistency properties guaranteed by the relativistic ordering model without requiring that the underlying system provide those properties for all programs.

The relativistic ordering model assumes a shared memory, with common addressing of that memory among threads. This memory need not consist of a single central resource accessed by all threads; it may consist of arbitrary local caches or memories, with various caching policies, coherence mechanisms, and consistency properties.

The underlying shared memory must provide a form of *eventual consistency*: eventually all stored values should propagate to become accessible for all potentially interested load operations [74, 79]. Furthermore, while eventual consistency only requires that stores propagate once writers stop performing new stores, real hardware propagates stores continuously even while writers continue to perform new stores, and the relativistic model (like most synchronization techniques) assumes this stronger form of eventual consistency. Multiple stores to the same address must still result in a single value

eventually propagating to all readers, though determining the winning store requires synchronization between writers. Limiting the duration of "eventually", or controlling the order of load and store operations, requires explicit constraints; the ordering model will provide the necessary constraints and their semantics.

4.2 Terminology

Given an address in memory, a thread may *load* a value from that address, or *store* a value to that address. The value returned by a load must precisely match a value stored by a previous store to the same address, and not an arbitrary value or a bitwise combination of previously stored values. While synchronization algorithms without this requirement do exist, notably Lamport's bakery algorithm for mutual exclusion [38], supporting joint-access parallelism without this property would prove far more difficult, and existing shared-memory systems all guarantee this property, at least for word-sized, word-aligned memory operations.

At the data-structure level, high-level *read* and *write* operations consist of delineated sequences of underlying memory loads and stores, respectively. Each read or write operation has a well-defined beginning and end, and corresponds to a high-level semantic operation provided by the data structure. For instance, linked lists provide lookups (a read operation), insertion (a write), and removal (a write). The hash table defined in chapter 3 additionally provides a node-move operation (a write). At any given time, a thread

will either act as a *reader*, performing a read operation, or a *writer*, performing a write operation; a thread can perform a series of both read or write operations, but will only take on one or the other role at any given time.

Note that while a write operation frequently includes both loads and stores, readers cannot observe the load operations performed by writers; thus, the analysis of concurrent reads and writes can treat writes as consisting entirely of store operations. Similarly, a read may perform store operations to private memory, but does not perform store operations on the shared data structure; thus, from a writer's perspective a read consists entirely of load operations.

Readers run concurrently with each other and with writers; they require no synchronization other than that provided by the implementation of the relativistic causal ordering model. Writers run concurrently with readers, but must synchronize with each other using some other mechanism, such as mutual exclusion.

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

4.3 The Relativistic Causal Ordering Property

The relativistic causal ordering model enforces a causality property: a reader that loads two different addresses will observe results consistent with the order of stores to those addresses. This causality property allows readers to rely on the natural local orderings of memory operations within writers, and vice versa.

Given a writer performing two store operations ordered by its desired local partial order, and a reader performing load operations on the locations of both of those writes, the relativistic causal ordering property guarantees that if the reader loads the value stored by the later store, it will load the value stored by the earlier store. Note that this property holds regardless of the order of the load operations within the reader.

The proscribed case would allow the reader to observe a result inconsistent with the writer's desired order. By avoiding this case, the relativistic ordering model ensures that the reader observes a state of memory consistent with a single point in the writer's order.

This property avoids the need to analyze complex interactions between multiple load and store operations. Instead, a reader appears to execute atomically between two ordered store operations of the writer: the reader can observe some prefix of the stores performed by the writer, and cannot observe the remaining stores. While writers do not run atomically, and readers can observe intermediate states, writers can safely assume that their store

operations become visible to readers in order, and need not consider out-of-order stores. The writer thus need only consider the state of memory (and the corresponding semantics of data structures in memory) after each store operation it performs, rather than every possible interleaving of load and store operations.

In the case where the writer enforces program order rather than some looser partial order, this makes the number of states to consider linear in the number of store operations, rather than exponential as in the case without this ordering property.

The relativistic ordering model provides a stronger consistency model than PRAM consistency [42], as it enforces the ordering of stores from a single writer. If causally-related writers enforce ordering between their causally-related stores just as they do between stores within the same writer, then relativistic causal ordering also provides a stronger consistency model than causal consistency [2]. Both PRAM consistency and causal consistency only require that a series of loads performed by a reader observe an ordered series of stores in a manner that does not regress: each load can observe strictly more writes than the previous load. The relativistic ordering model eliminates the interleaving entirely, requiring all loads within a reader to observe the same prefix of stores performed by a writer. This creates a form of transaction for the reader, making its entire series of loads appear atomic with respect to individual store operations.

4.4 Abstract Barrier Operations

Enforcing the relativistic causal ordering property on systems with weaker ordering properties requires explicit ordering constraints in readers and writers, expressed in the form of barrier operations. These barrier operations promote certain local ordering properties within a thread into global ordering properties across threads. Because barrier operations vary between hardware architectures, this section provides a set of abstract barriers with well-defined ordering properties. Section 4.5 shows how to insert these abstract barriers in readers and writers to enforce relativistic causal ordering; chapter 5 then shows how to translate these abstract barriers to concrete implementations suitable for running on hardware.

Relativistic causal ordering assumes three types of barriers: load barriers, store barriers, and wait-for-readers barriers.

A *load barrier* enforces a causal property between preceding loads and subsequent loads. Given a reader performing two loads separated by a load barrier, if the first load observes the result of a given store, the second load will not fail to observe any store ordered before the given store. (See section 5.1 and the definition of load barriers on real hardware for why this constraint need not imply an expensive hardware barrier.)

A *store barrier* enforces a causal property between preceding stores and subsequent stores. Given a writer performing two stores separated by a store barrier, if a reader performs loads on the locations of both of those stores,

and the earlier load observes the later store, the later load will observe the earlier store.

Notice that the definitions of load and store barriers interact. A load barrier creates a causal chain between loads and the partial order of stores as defined by writers; a store barrier creates a causal chain between stores and the order of loads as defined by readers.

A *wait-for-readers* barrier enforces ordering between store operations and an entire reader. Given a writer performing two stores separated by a wait-for-readers barrier, if a reader performs a load at the same address as the earlier store and fails to observe the result of that store, a later load in the same reader will fail to observe the results of any later store in the same writer.

The definitions of the abstract load and store barriers derive from the portable barrier abstractions used in Linux and their use in RCU-based algorithms. The definition of wait-for-readers comes directly from RCU itself [47].

4.5 Constructing Relativistic Algorithms

Building on the definitions of the underlying shared-memory model (section 4.1), the relativistic causal ordering property (section 4.3), and the barrier operations used to enforce this ordering property (section 4.4), this section presents the generalized construction technique for scalable concurrent data structure algorithms.

The relativistic ordering model allows separating the construction of con-

current algorithms into three components: the construction of reader and writer algorithms based on the desired memory model, the placement of barriers to properly enforce that memory model on real hardware, and the synchronization between writers.

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

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

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

Because relativistic readers define a total order on their loads, rather than a partial order, translating relativistic read algorithms to real systems requires inserting a load barrier between every pair of load operations in a reader. (Again, see section 5.1 for why this requirement need not imply an expensive hardware load barrier.)

To determine where to insert barriers in a relativistic writer, consider any pair of store operations in a writer, which may potentially require the insertion of some form of barrier separating them. Per the relativistic causal ordering property (section 4.3), the translation must prevent the case where a reader loads from the locations of both stores, observes the later store, but fails to observe the earlier store. This leads to three independent cases:

1. If no reader exists that loads from the locations of both stores, then no reader may violate the relativistic causal ordering property. Thus, the writer need not execute any barrier operation between the two store operations.

2. If readers exist that load from the locations of both stores, but all such readers load from those locations in the opposite order that the writer stores to them, the writer need only execute a store barrier between the two store operations. Per the definition of a store barrier (section 4.4), if the earlier load observes the later store, the later load will observe the earlier store. Thus, a store barrier ensures that the reader cannot observe the later store without observing the earlier store, preserving

the relativistic causal ordering property.

3. If any reader exists that loads from the locations of both stores in the same order that the writer stores to them, a store barrier does not suffice. However, a wait-for-readers barrier will ensure the desired ordering. Per the definition of a wait-for-readers barrier (section 4.4), if the earlier load fails to observe the earlier store, no load operation in the same reader, including the later load, can observe the later store. Thus, a wait-for-readers barrier ensures that the reader cannot observe the later store but not the earlier store, perserving the relativistic causal ordering property.

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

The traversal order of a reader need not define a total order on the el-

ements of a data structure. A relativistic data structure may have readers traversing any individual pair of memory locations in both directions. In such cases, one reader necessitates a store barrier between stores to those two locations, while the other reader necessitates a wait-for-readers barrier. However, a wait-for-readers barrier supersedes the need for a store barrier; thus, writers simply need to wait for readers between any pair of stores to those two locations, regardless of the order of those stores.

Note that if a single reader loads from the same location multiple times, none of the ordering barriers can prevent that reader from potentially seeing different values for each such load. That limits the applicability of this construction technique to data structures with acyclic read traversals. In practice, this does not significantly restrict the set of data structures usable with relativistic causal ordering; note in particular that it does not prohibit cyclic accesses used by writers only, such as previous pointers in a doubly-linked list or parent pointers in a tree. Shared cyclic data structures prove difficult to handle via fine-grained locking as well. Section 7.1 suggests some approaches for handling cyclic data structures and traversals.

Relativistic causal ordering does not define a specific method of synchronization between writers. In the simplest case, writers may synchronize via coarse-grained locking; if writes occur sufficiently infrequently compared to reads, this may suffice. To improve concurrency, writers may opt for fine-grained locking instead, partitioning the data structure into independent pieces and applying a lock to each. Typically, these pieces will remain inde-

pendent for readers as well, making any ordering concerns between writers moot; however, if a single reader may potentially load from locations stored to simultaneously by multiple writers, and the ordering of those store operations matters, the writers must arrange an appropriate barrier between their store operations.

As an alternative to fine-grained locking, writers may use a software transactional memory (STM) system to provide write concurrency. Howard [33] demonstrated a system combining relativistic readers with STM writers, such that relativistic readers can exist outside the transactional system while still preserving the isolation guarantees of the transactional system for writers. This preserves the low-overhead scalable performance of the readers while also allowing for scalable writers within the limitations of the STM system.

As previously suggested in the definition of the relativistic causal ordering property, writers may specify a partial order for their store operations rather than a total order, when the relative order of some store operations does not actually matter to readers. For instance, a writer initializing a structure before publishing it need not order individual store operations in the initialization, as long as all such store operations become visible before the store operation publishing the structure. The rules for the placement of barriers still hold, but requirements to place barriers between an ordered pair of store operations no longer specify an exact location in program order. Instead, the writer need only place barriers to satisfy all the ordering constraints between store operations. Note again that a wait-for-readers barrier between two

store operations supersedes the need for a store barrier between those same two store operations. Finding a correct barrier placement will always prove trivial; future work will present an algorithm for the optimization of finding a minimal barrier placement for partially ordered writes.

4.6 Reconstructing a Scalable Linked List

As a simple demonstration of the relativistic causal ordering model, this section reconstructs the algorithms for a scalable linked list. Previous work has extensively explored the construction of scalable linked lists using RCU [47, 56], and section 2.6.2 documents the known algorithms for RCU linked lists. The construction technique documented in section 4.5 should result in the same algorithms.

Linked lists most commonly provide three operations: a read operation to traverse the list (reconstructed in section 4.6.1), a write operation to insert a new node (reconstructed in section 4.6.2), and a write operation to remove an existing node (reconstructed in section 4.6.3).

4.6.1 Linked-List Lookup

Linked-list reads traverse the list from the head to the end via the next pointers of each node; reads may stop after encountering a node meeting some criteria. Per the construction technique, the read algorithm requires a load barrier between each pair of load operations. (As shown in section 5.1, these load barriers do not actually translate to expensive hardware load barriers.)

This barrier placement matches the established algorithm for RCU linked-list reads.

4.6.2 Linked-List Insert

Inserting a new node involves the following store operations:

1. Initialize the contents of the new node.
2. Initialize the next pointer of the new node.
3. Publish the new node by pointing some existing node's next pointer (or the head pointer) to that node.

The first two steps need not occur in any particular order with respect to readers; however, both of those steps must occur before the third, to ensure that readers cannot read uninitialized memory. Thus, the writer's partial order puts steps 1 and 2 before step 3. Since readers necessarily load the pointer stored to in step 3 before loading the node stored to in steps 1 and 2, the causally related store operations occur in reverse traversal order. Thus, this algorithm requires a store barrier between steps 1 and 3, and between steps 2 and 3. Since the order of steps 1 and 2 doesn't matter, the same barrier can serve both purposes. The following algorithm implements the required barrier placement:

1. Initialize the contents of the new node.
2. Initialize the next pointer of the new node.

- Execute a store barrier.

3. Publish the new node by pointing some existing node's next pointer (or the head pointer) to that node.

This implementation agrees with the established algorithm for RCU linked-list insertion.

4.6.3 Linked-List Remove

Removing an existing node involves the following store operations:

1. Set the next pointer of the previous node (or the head pointer if removing the first node) to point to the same place as the next pointer of the node to remove.

2. Reclaim the memory associated with the node. (This effectively represents a store operation to the contents of the node, after which readers must not access the contents of the node at all.)

These store operations occur in traversal order; thus, writers must execute a wait-for-readers barrier after making the node inaccessible but before reclaiming the memory, resulting in the following implementation:

1. Set the next pointer of the previous node (or the head pointer if removing the first node) to point to the same place as the next pointer of the node to remove.

- Wait for readers.

2. Reclaim the memory associated with the node.

This implementation agrees with the established algorithm for RCU linked-list removal.

4.7 Summary

Chapter 3 provided an example of the manual construction of a scalable concurrent data structure, and the complex reasoning required to construct and verify that data structure demonstrated the need for a memory ordering model. This chapter presented the relativistic causal ordering model, including the relativistic causal ordering property itself (section 4.3), and the algorithm construction and barrier placement rules that enforce this ordering property (section 4.5). Section 4.6 demonstrated that applying relativistic causal ordering to a linked list recreates the same algorithms established in previous publications on RCU linked lists. The next chapter provides concrete implementations of the abstract barrier operations to enforce relativistic causal ordering on real hardware. Chapter 6 uses relativistic causal ordering to construct a more complex data structure: a resizable hash table.

Chapter 5

Translating to Hardware Ordering Models

Chapter 4 defined the relativistic causal ordering property, and defined an abstract set of barriers usable to enforce those properties on top of the underlying shared-addressing system. Implementing the relativistic causal ordering model and algorithms using it on real hardware requires implementing the abstract barrier operations in terms of those available on that hardware. This chapter presents an implementation of relativistic causal ordering on real shared-memory hardware.

A typical shared-memory multiprocessor system includes multiple layers of caching between memory and each processor in an effort to mitigate the memory wall [29]. Such systems implement differing coherence protocols to maintain some consistency property between processors, though none implement the relativistic causal ordering property specified in chapter 4. Caching layers, as well as other properties of an architecture, potentially allow reordering of memory accesses in violation of the relativistic causal ordering property. In particular, store operations may remain in a writer's store buffer or cache before reaching memory, and load operations may return

data from a reader's cache without refreshing that cache from memory. All such caching may have different durations for different memory addresses, leading to potential reordering of store operations as observed by readers.

In this model, a load or store barrier effectively forces preceding operations to interact with memory before subsequent operations. (In practice, a barrier may instead force an operation to become visible to other processors by way of a cache coherence protocol without necessarily forcing it to reach memory.)

In addition to the reordering possible in hardware, compilers and language runtimes may also reorder operations, either directly or by providing another layer of caching behavior [6, 29]. For instance, a compiler may load a value from memory into a processor register, and reuse that register for a subsequent load from the same address without reloading it from memory, unless explicitly instructed to do otherwise.

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

5.1 Translating Load Barriers

A load barrier forces any load operations following the barrier to obtain values from memory at least as up to date as those returned by load operations prior to the barrier, with respect to each writer. A load barrier does not, however, interact with the caching layers of processors other than the one executing the barrier. This matches the semantics of various common load barrier instructions provided by current processors, as well as portable abstractions such as the `smp_rmb` function in the Linux kernel, the `cmm_smp_rmb` function provided by the userspace RCU implementation, or `atomic_thread_fence(memory_order_acquire)` in the C11 and C++11 standards.

In the most general case, a reader performing a series of unrelated independent reads would need to execute a load barrier operation between each pair of reads. However, in practice, readers typically perform a series of related loads to traverse some data structure in shared memory. In particular, readers often perform *dependent load* operations[1], in which subsequent loads depend on the results of previous loads. For instance, loading a pointer and subsequently dereferencing that pointer constitutes a dependent load. All current processors used in shared-memory multiprocessor systems automatically preserve the ordering of dependent loads without any explicit barrier. Thus, a reader performing a data structure traversal that consists entirely of

[1] Also commonly referred to as *dependent read* operations when not making a distinction between the terms "load" and "read".

dependent loads need not execute any expensive hardware memory barrier instructions.

As a notable exception, the DEC Alpha processor did not order dependent loads; concurrent algorithms that require portability to Alpha must use an explicit barrier between dependent reads [13]. The Linux kernel provides an `smp_read_barrier_depends` operation with precisely these semantics, which uses an appropriate barrier on Alpha and compiles to nothing on all other architectures. The Linux implementation of RCU also provides a function (`rcu_dereference`) for the common case of dereferencing a pointer as a dependent load, and this function includes the necessary barrier implicitly.

5.2 Translating Store Barriers

For writers, a store barrier follows the same model of bypassing caching layers between the writer and memory. Executing a store barrier does not necessarily guarantee that preceding store operations have reached memory; however, it does guarantee that subsequent store operations will not reach memory any sooner than preceding store operations. This matches the semantics of common store barrier instructions provided by current processors, as well as portable abstractions such as the `smp_wmb` function in the Linux kernel, the `cmm_smp_wmb` function in the userspace RCU implementation, or `atomic_thread_fence(memory_order_release)` in the C11 and C++11 standards.

5.3 Translating Wait-For-Readers

Implementing the wait-for-readers barrier requires a mechanism that allows writers to wait for current readers to complete, ideally without incurring synchronization costs in readers. Read-Copy Update (RCU) [47, 23, 56] provides an implementation of this barrier.

As documented in section 2.6.1, RCU provides a `synchronize_rcu` operation which waits for all currently running readers to finish. Using this operation requires delimiting the start and end of each read operation using the lightweight, CPU-local operations `rcu_read_lock` and `rcu_read_unlock`.

`synchronize_rcu` provides the necessary semantics for the wait-for-readers barrier: if a writer executes a `synchronize_rcu` between two store operations, any reader that fails to observe the earlier store must complete before `synchronize_rcu` completes, and therefore cannot observe the store operations after the `synchronize_rcu`. RCU preserves the scalability of readers by not adding any expensive synchronization primitives to those readers; RCU's read-side delineation incurs little to no cost.

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

Chapter 6

Relativistic Concurrent Hash-Table Resize

Section 4.6 provided a simple example of applying the relativistic causal ordering model to the construction of a relativistic linked list. This chapter presents a more complex example: constructing a concurrent hash-table resize algorithm based on relativistic causal ordering.

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 minimal complexity: simply acquire the appropriate locks to exclude concurrent reads and writes, then move items to a new table. However, relativistic

algorithms *cannot exclude readers*; doing so would require expensive read-side synchronization, severely limiting scalability and performance. Thus, any relativistic hash-table resize algorithm must cope with concurrent reads while resizing.

The algorithms presented in this chapter support resizing an RCU-based hash table without blocking or slowing concurrent lookups. Because lookups can occur at any time, the relativistic hash table must remain in a consistent state at all times, and must never allow a lookup to spuriously miss an entry due to a concurrent resize operation. Furthermore, the resize algorithm avoids copying the individual hash-table nodes, allowing readers to maintain persistent references to table entries.

6.1 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.

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.

Section 6.1.1 provides the hash-table lookup operation. Section 6.1.2 provides the hash-table shrink operation. Section 6.1.3 provides the hash-table expand operation. Section 6.1.4 provides variations of the shrink and expand operations that can resize a hash-table in-place rather than creating a new table and later reclaiming the old.

6.1.1 Hash-Table Lookup

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

1. Load the hash-table pointer.
2. Load the number of buckets and the pointer to the array of buckets.
3. Hash the desired key, modulo the number of buckets.
4. Load the head pointer of 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]

Making this algorithm work as a relativistic reader requires wrapping the algorithm in RCU read-side delineation (`rcu_read_lock` and `rcu_read_unlock`) and adding load barriers between each pair of loads. Since this algorithm performs entirely dependent loads, those load barriers do not require expensive hardware barriers on modern architectures.

Lookups will traverse the hash table concurrently with other operations, including resizes. A resize must not disrupt concurrent lookups, such as by causing a lookup to fail to find a node. Thus, each hash chain must always contain all those items that hash to the corresponding bucket. Most prior hash table resize algorithms ensure that a hash chain contains *exactly* those items. This relativistic resize algorithm loosens this constraint, by also allowing hash chains to ephemerally contain items that hash to *different* buckets. Let the term *imprecise* refer to such hash chains, since they include all items that hash to that bucket but may include others as well. Readers must tolerate imprecise hash chains, but this does not require any changes to the standard lookup algorithm, which already ignores any element whose key does not match.

[1] 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.

6.1.2 Hash-Table Shrink

Because resizing occurs by integral factors, when shrinking the table, each bucket of the new table will contain all entries from multiple buckets of the old table; together with the notion of imprecise hash chains, this allows the straightforward construction of a relativistic algorithm to shrink a hash table:

1. Allocate the new, smaller table.
2. Link each bucket in the new table to the first bucket in the old table containing entries that 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. Reclaim the old hash table.

All of the store operations in steps 1 through 4 must occur to make the hash table valid before the publication of the table in step 5. Because readers load the table reference before loading anything else, these store operations occur in reverse traversal order, so the writer must execute a store barrier between steps 4 and 5.

The publication of the new hash table in step 5 must occur before the reclamation of the old hash table in step 6. Because readers load the table reference before loading the contents of the table, this pair of stores occurs in traversal order, so the writer must wait for readers between steps 5 and 6.

Finally, the shrink algorithm must include appropriate write-side synchronization to prevent conflicts between concurrent resize attempts, or between a resize and any other write to the hash table.

With barriers and write-side synchronization included, the complete algorithm for the relativistic hash shrink writer becomes:

1. Synchronize with other writers, such as by acquiring a lock. (This does not exclude readers, nor do readers exclude writers.)
2. Allocate the new, smaller table.
3. Link each bucket in the new table to the first bucket in the old table containing entries that hash to the new bucket.
4. 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.
5. Set the table size.

- Execute a store barrier.

6. Publish the new, valid hash table.

7. Release any locks or other write-side synchronization acquired in step 1.

- Wait for readers.

8. Reclaim the old hash table.

Figure 6.1 provides an example of the relativistic hash shrink algorithm.

6.1.3 Hash-Table Expand

Because resizing occurs by integral factors, when growing the table, each bucket of the new table will contain entries from at most one bucket of the old table. This constraint allows the construction of a relativistic algorithm to expand a hash table, albeit not quite as straightforwardly. The relativistic causal ordering property guarantees that store operations will become visible in order, but the hash table must remain semantically valid after every store operation. The concept of imprecise hash chains broadens the definition of "semantically valid" enough to allow for an incremental resize operation. All of these constraints together provide enough guidance to construct the expand algorithm.

As before, the algorithm will begin by allocating the new table, and will conclude by reclaiming the old table. However, in the shrink case, the new table did not become valid until the rest of the algorithm had completed, and thus publication occurred at the end. The reverse holds true for expansion: the expanded table will need to break many links present in the old table, to partition nodes into buckets, which will quickly make the old table invalid.

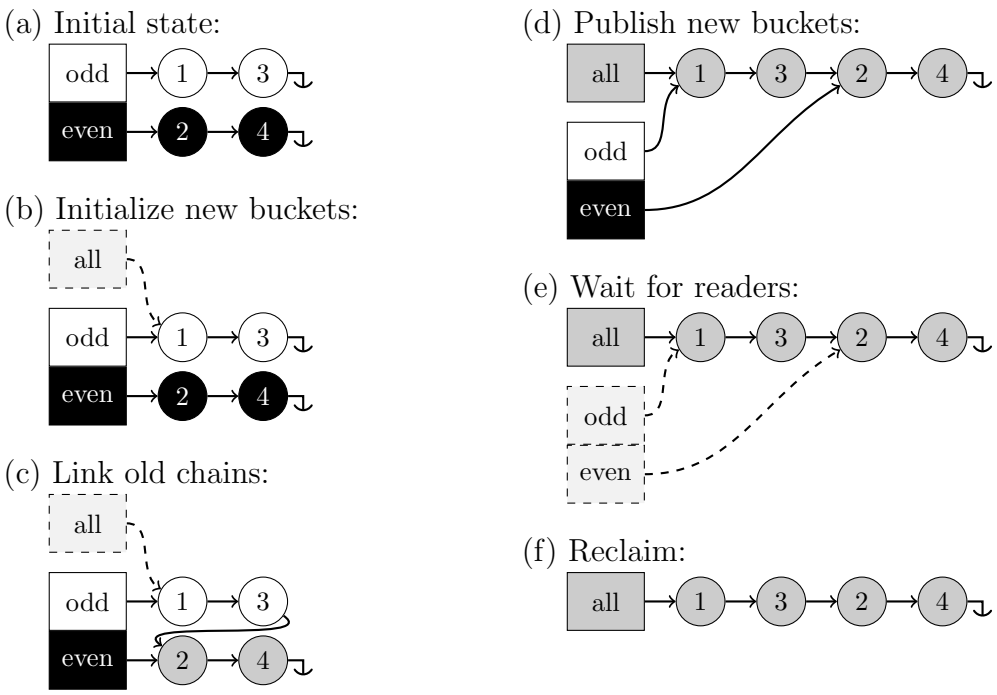

Figure 6.1: Shrinking a relativistic hash table. White nodes indicate reachability by odd readers, black nodes by even readers, and gray nodes by both odd and even readers. (a) The initial state has two buckets, one for odd numbers and one for even numbers. (b) The resizer allocates a new one-bucket 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. (d) The resizer publishes the new table. (e) After waiting for readers, (f) the resizer can free the old table.

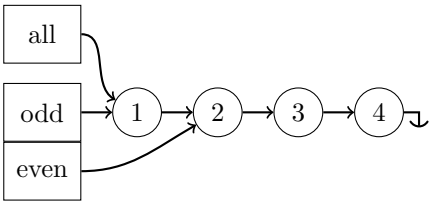

Figure 6.2: Example of a hash table with imprecise "zipped" buckets. The "all" bucket represents a smaller hash table, with all nodes in a single bucket. The "odd" and "even" buckets represent an expanded hash table. Each bucket contains all the nodes a reader expects to find, as well as some the reader will safely ignore.

Thus, before invalidating the old table, readers must begin using the new table, which must therefore start out in a semantically valid state. Imprecise hash chains allow the construction of a semantically valid hash table with an expanded set of hash buckets, but whose hash chains remain those of the smaller table, with multiple hash buckets referencing each chain; figure 6.2 gives an example of an expanded hash table with imprecise "zipped" buckets. The remainder of the algorithm must therefore "unzip" the buckets, partitioning the nodes into the desired target buckets, all while keeping the entire hash table semantically valid.

Finally, note that unzipping a hash chain requires a series of ordered store operations in that chain, starting at the beginning and proceeding towards the end. Readers will access the locations of multiple such stores while traversing a hash bucket, in the same order that they occur, necessitating wait-for-readers barriers between each pair of stores. However, no one reader will access the nodes from multiple zipped buckets, so the order of stores in different buckets does not matter. Thus, the writer's partial order contains

an independent ordered series of stores for each bucket. Executing the unzip operations for each bucket in parallel allows them to share the same wait-for-readers barriers, minimizing the number of these barriers and making the expansion algorithm more efficient.

The following relativistic hash-table expansion algorithm implements all of the aforementioned constraints:

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, constructing a valid "zipped" hash table.
3. Set the table size.
4. Publish the new table pointer.
5. For each non-empty bucket in the old table (each of which contains items from multiple buckets of the new table):
    - 5.1 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.
    - 5.2 Find the subsequent node that does hash to the same bucket as node p, or NULL if no such node exists.
    - 5.3 Set p's next pointer to that subsequent node pointer, bypassing the nodes that do not hash to p's bucket.

6. If any changes occurred in this pass, repeat from step 5.

7. Reclaim the old hash table.

Steps 1 through 3 initialize the new table, which must occur before publishing the new table in step 4; because readers load the table pointer before any contents of the table, these stores occur in reverse traversal order, necessitating a store barrier between steps 3 and 4.

The publication of the new table in step 4 must occur before the loop unzipping the hash chains starting in step 5, because unzipping makes the old table invalid. These store operations occur in traversal order, so the writer must wait for readers between steps 4 and 5.

Each iteration of the unzipping loop in steps 5 through 6 must complete before the subsequent iteration. The unzipping process modifies hash chains in traversal order, so the writer must wait for readers after each iteration of the loop.

Finally, the expansion algorithm must include appropriate write-side synchronization to prevent conflicts between concurrent resize attempts, or between a resize and any other write to the hash table.

With the barriers and write-side synchronization in place, the finished relativistic hash-table expansion algorithm looks as follows:

1. Synchronize with other writers, such as by acquiring a lock. (This does not exclude readers, nor do readers exclude writers.)

2. Allocate the new, larger table.

3. 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, constructing a valid "zipped" hash table.

4. Set the table size.

- Execute a store barrier.

5. Publish the new table pointer.

- Wait for readers.

6. For each non-empty 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 that 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 that do not hash to p's bucket.

- Wait for readers.

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

8. Release any locks or other write-side synchronization acquired in step 1.

9. Reclaim the old hash table.

For an example of the expansion algorithm, see figure 6.3.

Note that the expansion algorithm executes several wait-for-readers barriers while holding write-side locks or other synchronization. Assuming the use of mutual exclusion, this will exclude all write operations for a prolonged period while waiting on readers. However, unlike many hash-table resize algorithms, this algorithm permits readers to run concurrently with a resize operation, and does not impact the throughput of concurrent readers.

6.1.4 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, some adaptation to the resize algorithms would allow them to perform resizes in place instead, given a memory allocator that can resize existing allocations without moving them. 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.

Shrinking a hash table in place requires adapting the previous shrink algorithm to avoid disrupting unfinished readers. 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. Thus, the shrink algorithm need not take a separate step to publish the new table;

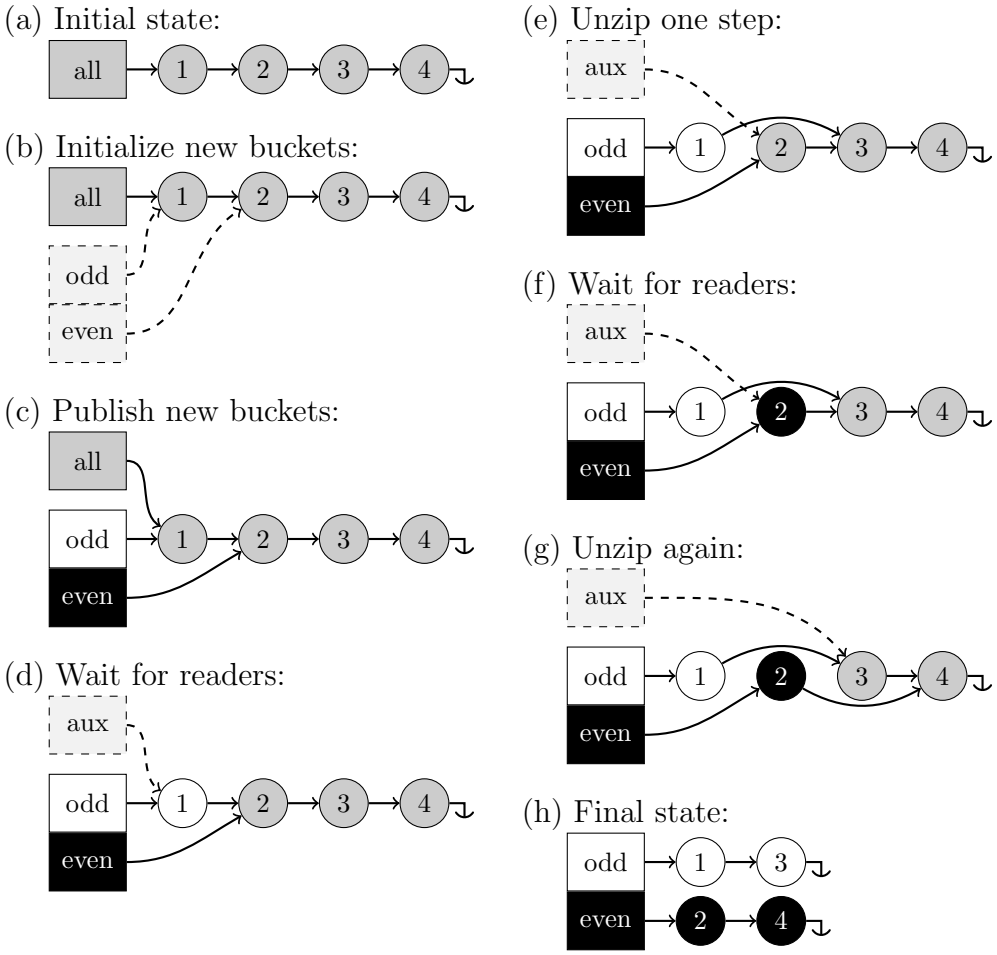

Figure 6.3: Growing a relativistic hash table. Colors as in figure 6.1. (a) The initial state contains one bucket. (b) The resizer allocates a new two-bucket table and points each bucket to the first item with a matching hash; this produces valid imprecise hash chains. (c) The resizer can now publish 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.

instead, setting the size of the table effectively publishes the new table. Since readers load the table size before loading the buckets, the shrink algorithm must wait for readers after concatenating the buckets and before setting the table size:

1. Synchronize with other writers, such as by acquiring a lock. (This does not exclude readers, nor do readers exclude writers.)
2. As before, concatenate all the buckets containing entries that hash to the same bucket in the smaller table.

- Wait for readers.

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

- Wait for readers.

4. Shrink the table's memory allocation.
5. Release any locks or other write-side synchronization acquired in step 1.

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

6.2 Comparisons with Other Algorithms

Existing RCU-based concurrent hash tables in the Linux kernel, such as the directory-entry cache (dcache) [55, 41], do not support resizing; they allocate a fixed-size table at boot time based on system heuristics such as available memory. Prior attempts to build resizable RCU hash tables have arisen from the limitations of these fixed-size RCU hash tables in the Linux kernel.

Nick Piggin's "Dynamic Dynamic Data Structures" (DDDS) [64] 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 hash tables; relativistic hash tables do neither. Section 6.3.1 discusses DDDS further.

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 [83]. 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 [28, 18, 57, 8] have proposed resizable concurrent hash tables. Unlike relativistic hash tables, these algorithms require expensive synchronization operations in readers, 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 [28]; all of these require expensive synchronization, and some require retries. Gao, Groote, and Hesselink proposed a lock-free hash table using closed hashing [18]; their approach relies on atomic operations and on helping concurrent operations complete, both of which introduce synchronization overhead.

Maged Michael implemented a lock-free hash table based on compare and swap (CAS) [57], 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 load barriers), but must retry on any concurrent modification. To support safe memory reclamation, Michael uses hazard pointers [58], 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 [23].

The relativistic hash table presented here uses 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. To decrease the cost of these copies, closed hash table buckets can store pointers to large nodes rather than the nodes themselves, but this introduces an additional indirection, eliminating the primary benefit of using a closed table. 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 [8]; 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 load barrier but no other synchronization. Writers require a CAS but not a store 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; such reasoning reduces the number of ordering relationships to enforce between store operations. However, this state-based reasoning remains entirely specific to the hash table design presented, with no generalizations offered to apply to other data structures. 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 [71, 28]. 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 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.

As documented in chapter 3, I previously developed a relativistic algorithm for moving a hash-table entry from one bucket to another atomically [76, 77]. 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. This prevents readers from maintaining persistent references to entries, which breaks real-world applications such as the Linux dcache; dcache readers frequently maintain persistent references to dcache entries via reference count.

The synthetic `rcuhashbash-resize` benchmarking framework documented in section 6.3.2 consists of a Linux kernel module, using the implementation of RCU in Linux. 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 [10]. The real-world benchmarks based on the memcached key-value storage engine (documented in section 6.3.3), use liburcu to implement a modified memcached storage engine.

6.3 Benchmark Methodology

This section presents the methods and implementation details used in benchmarking the hash-table resize algorithm. Section 6.3.1 outlines the other resizable hash-table algorithms included in the benchmarks for comparison. Section 6.3.2 documents the `rcuhashbash-resize` microbenchmark, and section 6.3.3 documents the real-world benchmarks on memcached.

Section 6.3.4 provides a collected summary of the hypotheses from sections 6.3.1 through 6.3.3; the corresponding section 6.4.3, after the benchmark results, reviews all of these hypotheses and details how the benchmark results confirm these hypotheses.

6.3.1 Other Benchmarked Algorithms

I evaluated relativistic hash tables both through microbenchmarks on the data structure operations themselves, and through real-world benchmarks on an adapted version of the memcached key-value storage engine. The microbenchmarks directly compare the 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, I 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. This implementation will not scale well, but it represents the best-known method based on mutual exclusion, and I included it to provide a baseline for comparison.

For a more competitive comparison, I turned to Nick Piggin's "Dynamic Dynamic Data Structures" (DDDS) [64]. 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 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 load 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 theoretically lead to a livelock, though in practice resizes do not occur frequently enough for a livelock to arise.

DDDS should 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 load barrier in the sequence counter, and the potential retries with a concurrent resize. Thus, relativistic hash tables should outperform DDDS significantly when running a concurrent resize, and slightly even without a concurrent resize.

For a real-world benchmark, I chose memcached, a key-value storage engine widely used on servers 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, memcached's default engine should 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.

I implemented a new relativistic 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 relativistic storage engine, I chose instead to copy the value out of a key-value pair while still within a relativistic 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 that would require mutating the store, and falls back to the slow path in those cases.

With the new relativistic storage engine, memcached should 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. Because I added wait-for-readers barriers to the SET handling, SET will become marginally slower, but the scalability of SET requests should not change; I believe this

tradeoff will prove acceptable in exchange for making GET requests scalable.

6.3.2 Microbenchmark: rcuhashbash-resize

To compare the performance and scalability of our algorithms to the alternatives, I created a test harness and benchmarking framework for resizable hash-table implementations. I 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, I created the 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: the 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, I implemented the in-place variation of the shrink algorithm and the copying implementation of the 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. I consider continuous resizing a harsh benchmark, but one which a scalable concurrent implementation should handle reasonably. Furthermore, I can perform separate benchmark runs to evaluate the cost of the lookup in the absence of resizes.

The microbenchmark runs in this chapter all used a hash table with 2^{16} entries. For each of the three implementations, I benchmarked three cases: no resizing and 2^{13} buckets, no resizing and 2^{14} buckets, and continuous resizing between 2^{13} and 2^{14} buckets. Lookups should take less time in a table with more buckets. For the relativistic resize algorithm, with its minimal impact on lookup performance, the number of lookups with a concurrent resizer should fall between the no-resize cases with the smaller and larger tables. The other two resize algorithms should have significantly more impact on concurrent lookups, causing the lookups performance to decrease relative to lookups with fixed-size tables.

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

The 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, I ran each benchmark with 1, 2, 4, 8, and 16 concurrent reader threads, with and without an additional resize thread. In all cases, I ran fewer threads than the hardware supported, thus minimizing the need to pass through the scheduler and allowing free CPUs to soak up any unremov-

able OS background noise. (However, performance may behave somewhat less than linearly when passing 12 threads, as that matches the number of hardware cores.)

All benchmark runs occurred on a Linux 2.6.37 kernel, targeting the x86-64 architecture. I used the default configuration (`make defconfig`), with the hierarchical RCU implementation, and no involuntary preemption.

6.3.3 Real-World Benchmarks: Memcached

As a client-server program, memcached required a separate benchmarking program. At the recommendation of memcached developers, I used mc-benchmark, developed by Salvatore Sanfilippo. To minimize the impact of network overhead, I 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, I 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, I ran from 1 to 12 mc-benchmark processes, each of which simulated

4 clients.

To run the memcached server and mc-benchmark client, and to collect statistics on the request rate, I used a benchmark script supplied by 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.

6.3.4 Summary of Hypotheses

This section provides a summary of the hypotheses from the previous sections 6.3.1, 6.3.2, and 6.3.3; section 6.4.3 provides a corresponding summary of how the benchmark results confirm these hypotheses.

1. The resize implementation using reader-writer locking will not scale well, if at all.

2. Relativistic hash-table lookups should outperform DDDS significantly when running concurrent resizes, and slightly even without a concurrent resize.

3. Memcached's default engine should hit a hard scalability limit, beyond

which it will not scale to more requests regardless of available resources.

4. With the new relativistic storage engine, memcached should 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.

5. With the new relativistic storage engine, memcached SET requests will become marginally slower, but the scalability of SET requests should not change.

6. For the relativistic resize algorithm, with its minimal impact on lookup performance, the number of lookups with a concurrent resizer should fall between the no-resize cases with the smaller and larger tables.

7. The other two resize algorithms should have significantly more impact on concurrent lookups, causing the lookups performance to decrease relative to lookups with fixed-size tables.

8. Performance may behave somewhat less than linearly when passing 12 threads, as that matches the number of hardware cores.

6.4 Benchmark Results

6.4.1 Microbenchmark Results

To evaluate baseline reader performance in the absence of resizes, I first compare lookups per second for all the implementations with a fixed table size of

2^{13} buckets; figure 6.4 shows this comparison. As predicted, the relativistic hash table and DDDS remain very competitive when not concurrently resizing, though as the number of concurrent readers increases, the relativistic 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.

I 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 6.5 compares the lookups per second for the relativistic implementation and DDDS in the face of concurrent resizes. (I omit rwlock from this figure, because it would vanish against the horizontal axis; with 16 CPUs, relativistic hash tables provide 125 times the lookup rate of rwlock.) With a resizer running, the relativistic table's 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. The relativistic lookup rate scales linearly with reader threads, demonstrating a clear scalability advantage over DDDS.

To more precisely evaluate the impact of resizing on lookup performance for each implementation, I compare the lookups per second when resizing

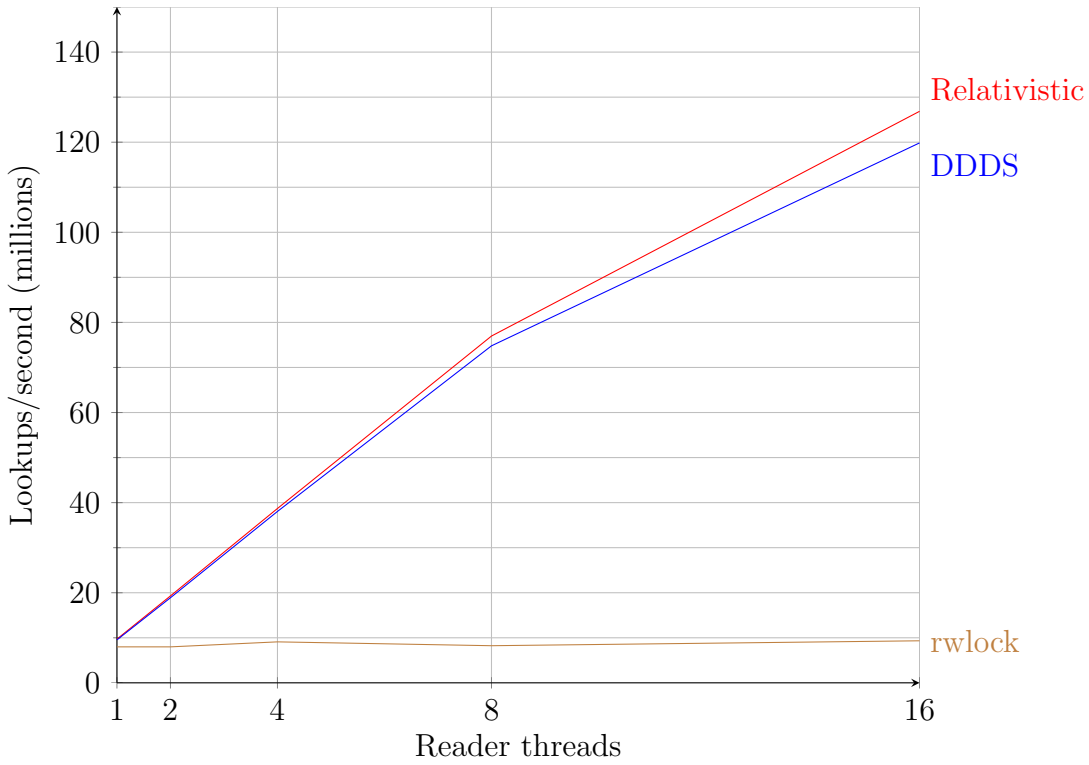

Figure 6.4: Lookups/second by number of reader threads for each of the three implementations, with a fixed hash-table size of 2^{13} buckets, and no concurrent resizes.

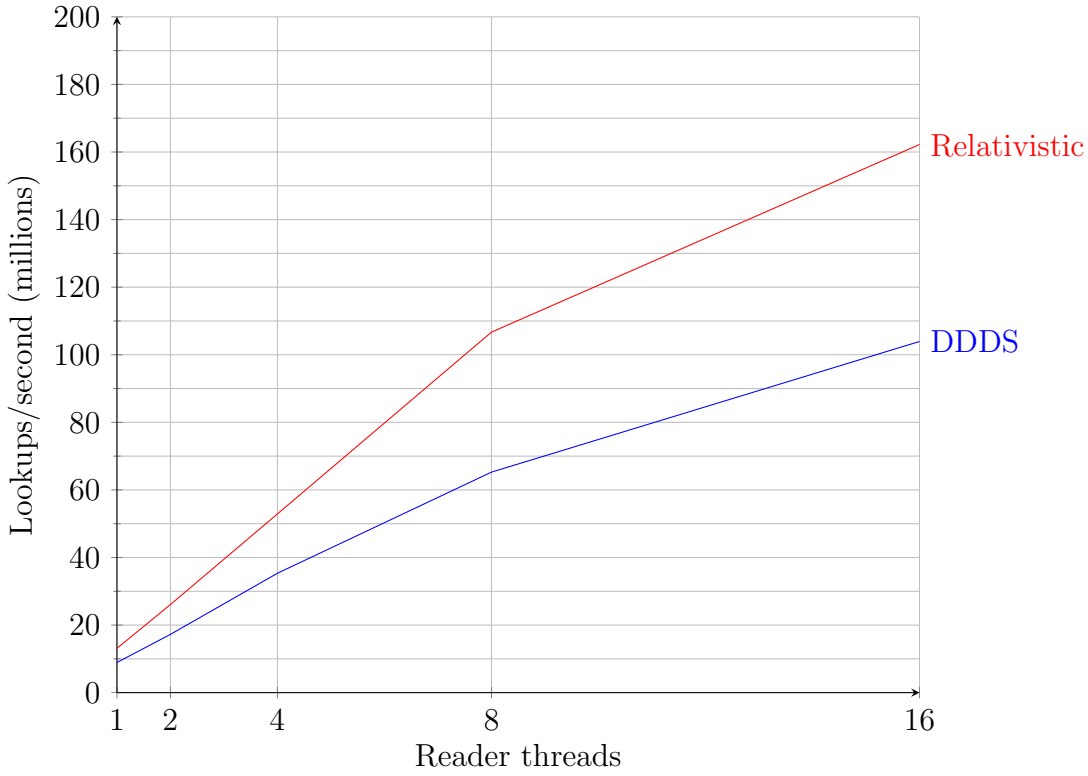

Figure 6.5: Lookups/second by number of reader threads for the relativistic hash table implementation versus DDDS, with a concurrent resize thread continuously resizing the hash-table between 2^{13} (8k) and 2^{14} (16k) buckets. rwlock omitted as it vanishes against the horizontal axis.

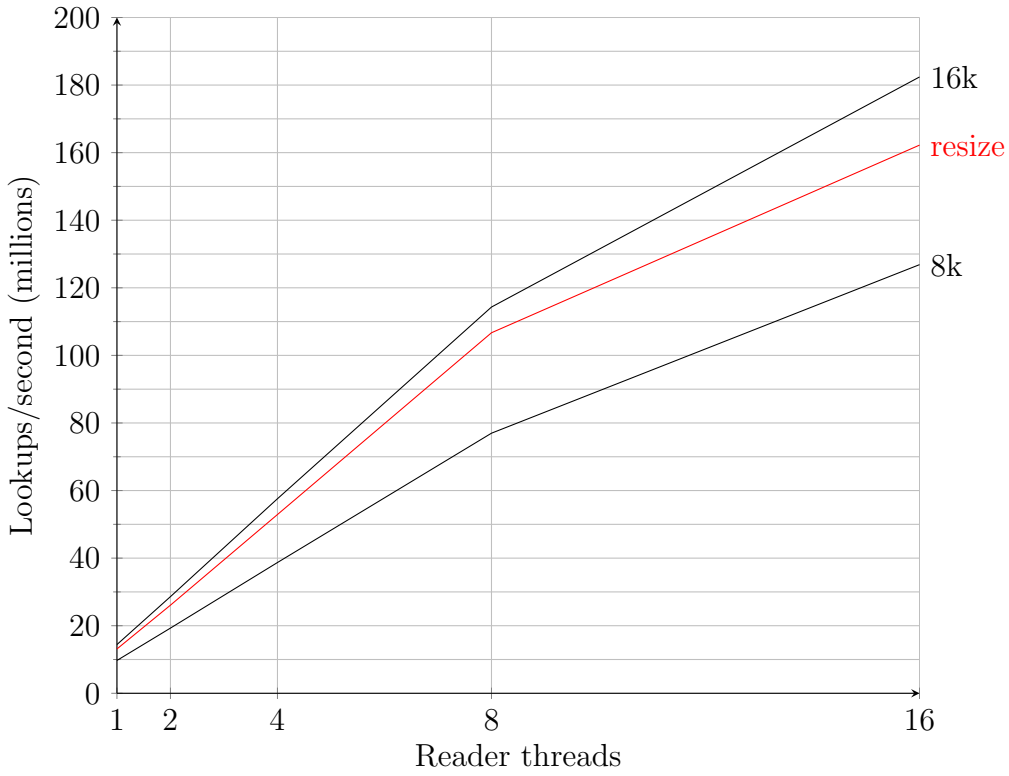

Figure 6.6: Lookups/second by number of reader threads for the relativistic hash-table resize algorithms, showing the impact of resizing on relativistic hash-table performance. "8k" and "16k" indicate fixed hash-table sizes in buckets; "resize" indicates continuous resize between the two sizes.

to the no-resize cases for the larger and smaller table size. Figure 6.6 shows the results of this comparison for the relativistic 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 the relativistic resize algorithms add little to no overhead to concurrent lookups.

Figure 6.7 shows the same comparison for the DDDS resize algorithm. In this case, the lookup rate with a resizer running falls below the lower

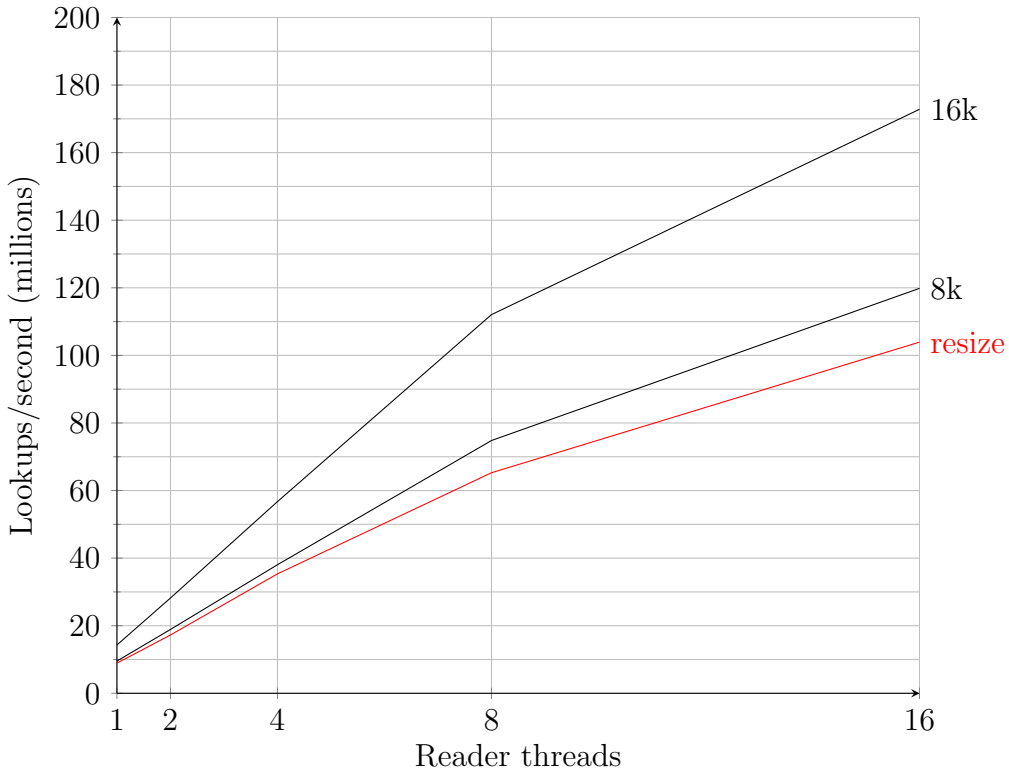

Figure 6.7: Lookups/second by number of reader threads for the DDDS resize algorithm, showing the impact of resizing on DDDS hash-table performance. "8k" and "16k" indicate fixed hash-table sizes in buckets; "resize" indicates continuous resize between the two sizes.

bound of the smaller hash table. This suggests that the DDDS resizer adds significant overhead to concurrent lookups, as predicted.

Finally, figure 6.8 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.

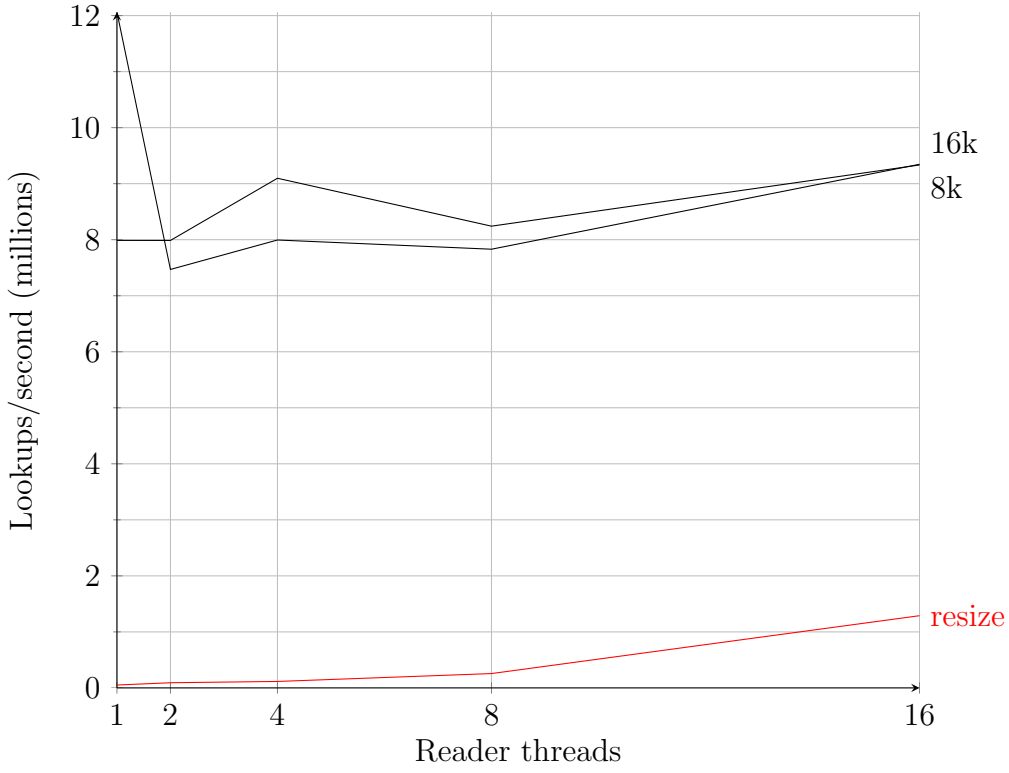

Figure 6.8: Lookups/second by number of reader threads for the rwlock-based implementation, showing the impact of resizing on rwlock-based hash-table performance. "8k" and "16k" indicate fixed hash-table sizes in buckets; "resize" indicates continuous resize between the two sizes.

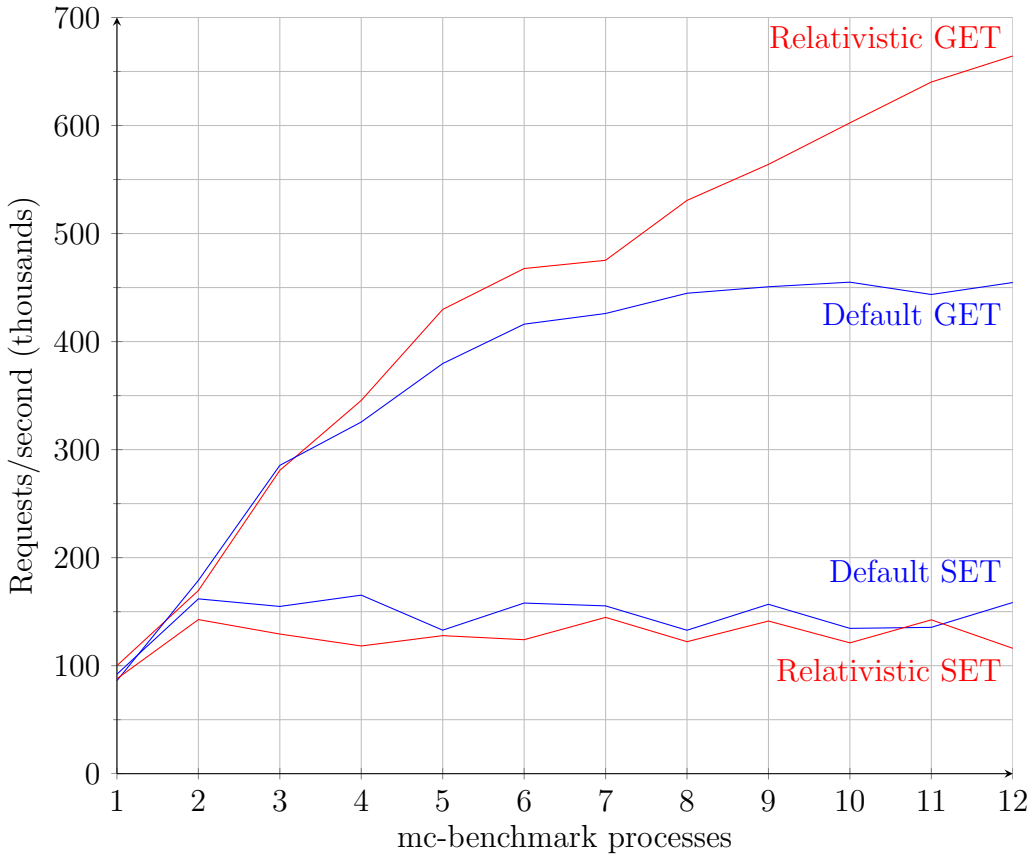

Figure 6.9: GET and SET operations per second by number of mc-benchmark processes for the default memcached storage engine and the new relativistic storage engine. Each mc-benchmark process simulated 4 clients to saturate the CPU.

6.4.2 Memcached Results

Figure 6.9 shows the results of my benchmarks on memcached. 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 relativistic 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 relativistic engine services 46% more GET requests per second than the default engine.

As expected, SET requests do not scale in either engine. In the relativistic 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.

I 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.

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

6.4.3 Benchmark Summary

The relativistic resizable hash table provides linearly scalable lookup performance in both microbenchmarks and real-world benchmarks. In microbenchmarks, the relativistic implementation surpassed DDDS by a widening mar-

gin of up to 56% with 16 reader threads; both implementations vastly dwarfed reader-writer locks, with the relativistic implementation providing a 125x improvement with 16 readers. Furthermore, the relativistic 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 relativistic storage 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.

Reviewing the hypotheses from section 6.3.4:

1. The resize implementation using reader-writer locking will not scale well, if at all. Confirmed by experiment: figure 6.4 shows that reader-writer locking did not scale at all even in a read-only workload.

2. Relativistic hash-table lookups should outperform DDDS significantly when running concurrent resizes, and slightly even without a concurrent resize. Confirmed by experiment: figure 6.4 shows that relativistic hash-table lookups slightly outperformed DDDS in the absence of concurrent resizes, and figure 6.5 shows that relativistic hash-table lookups significantly outperformed DDDS when running concurrent resizes.

3. Memcached's default engine should hit a hard scalability limit, beyond which it will not scale to more requests regardless of available resources.

Confirmed by experiment: the data in figure 6.9 shows a clear ceiling on GET request throughput with the default engine.

4. With the new relativistic storage engine, memcached should 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. Confirmed by experiment: the data in figure 6.9 shows that GET request throughput with the relativistic storage engine continued to scale up to the limits of the machine.

5. With the new relativistic storage engine, memcached SET requests will become marginally slower, but the scalability of SET requests should not change. Confirmed via experiment: the data in figure 6.9 shows SET request throughput for the relativistic engine slightly lower than that of the default engine, but this difference did not grow as concurrency increased.

6. For the relativistic resize algorithm, with its minimal impact on lookup performance, the number of lookups with a concurrent resizer should fall between the no-resize cases with the smaller and larger tables. Confirmed via experiment: figure 6.6 shows that with continuous concurrent resizing, relativistic lookup performance exceeded that of the smaller fixed-size hash table, while not reaching the performance of the larger fixed-size table.

7. The other two resize algorithms should have significantly more impact

on concurrent lookups, causing the lookups performance to decrease relative to lookups with fixed-size tables. Confirmed via experiment: figure 6.6 shows that continuous resizes caused DDDS lookup performance to drop below that of the smaller fixed-size hash table, and figure 6.6 shows that continous resizes caused rwlock-based lookups to drop far below that of the smaller fixed-size hash table.

8. Performance may behave somewhat less than linearly when passing 12 threads, as that matches the number of hardware cores. Confirmed via experiment: figures 6.4 and 6.5 shows lookup performance roughly doubling as the number of threads doubles up to 8 threads, but then only increasing by about 50% from 8 to 16 threads.

6.5 Summary

This chapter presented an algorithm for resizing a hash table while supporting concurrent, scalable readers. This algorithm serves as both a novel contribution of this dissertation and as a demonstration of constructing scalable concurrent data structures using the relativistic causal ordering model presented in chapter 4. Unlike the node-move algorithm given in chapter 3, this resize algorithm did not require complex, data-structure specific reasoning to construct or verify. The rules given in section 4.5 made the placement of barriers a highly mechanical process. Nonetheless, this resize algorithm provided the linear read scalability expected from an RCU-based algorithm.

Chapter 7

Conclusions

The performance of applications on modern hardware depends most critically on the ability to scale to as many processors as the hardware can make available. However, the most widely used techniques for concurrent programming, namely mutual exclusion and transactional memory, do not provide the necessary scalability; in particular, they do not allow concurrent access to the same data (joint-access parallelism), only concurrent access to separate data (disjoint-access parallelism). Techniques such as Read-Copy Update allow the construction of data structures with highly scalable read algorithms that can run concurrently with writers, which prove ideal for the common case of read-mostly workloads. However, such techniques necessitate complex, data-structure-specific reasoning to produce correct concurrent data structures (as demonstrated in chapter 3), and in particular, these techniques force programmers to contend directly with arbitrary reordering of memory operations. Mutual exclusion, by contrast, provides a well-known set of rules supporting the general construction of arbitrary data structures, which has contributed to its widespread adoption despite severe scalability limitations.

I have presented the *relativistic causal ordering* model (chapter 4), which supports the straightforward construction of a wide variety of scalable concurrent data structures. This model allows write algorithms to assume that readers will observe causally ordered store operations in order, rather than reasoning about arbitrary interleavings or reorderings of load and store operations. Writers need only ensure that they preserve the semantic validity of the data structure after each store operation.

To support the relativistic causal ordering model on weakly ordered shared-memory systems, I defined an abstract set of barrier operations that readers and writers could insert between pairs of memory operations to enforce the ordering of those operations, along with the precise rules for the application of these barriers. Summarizing the rules for barrier usage:

- Between a pair of loads in a reader, use a load barrier.
- Between a pair of stores in a writer:
  - If no single reader reads both stores, use no barrier.
  - If readers read both stores but only in the reverse order of the stores, use a store barrier.
  - If any reader reads both stores in the same order as the stores, use a wait-for-readers barrier.

Chapter 5 provides translations of the abstract barriers to implementations on real hardware. A load barrier between a dependent load and the load

it depends on (such as between the load of a pointer and a load through that pointer) requires no hardware barrier instruction on any modern hardware. Any other load barrier requires a hardware load barrier instruction. A store barrier requires a hardware store barrier instruction. Read-Copy Update provides a scalable implementation of a wait-for-readers barrier in the form of `synchronize_rcu`, which requires the use of the lightweight CPU-local operations `rcu_read_lock` and `rcu_read_unlock` to delineate the start and end of readers. Finally, note that all of the abstract barriers must include any necessary directives for compilers or language runtimes to ensure that they perform no more reordering than the underlying hardware.

I have demonstrated the effectiveness of the relativistic causal ordering model by using it to derive a resize algorithm for scalable concurrent hash tables (chapter 6). In place of the complex, one-off reasoning typically found in RCU-based data structure implementations, this derivation followed only the obvious series of steps necessitated by the requirements of the ordering model and the data structure semantics. The resulting resizable hash table demonstrated the same degree of scalability expected of RCU-based data structures.

The relativistic causal ordering model, together with the translations to real hardware, enables the construction of scalable concurrent algorithms for arbitrary acyclic data structures. This model provides a viable competitor to the generic construction rules of lock-based concurrent programming, allowing the implementers of scalable concurrent data structures to move beyond

the painful choice between the simplicity of locking and the scalability of techniques such as RCU.

7.1 Future Work

The relativistic causal ordering model focuses exclusively on read-side concurrency, requiring writers to synchronize with each other via other means, such as mutual exclusion or transactional memory. This allows readers to scale linearly, but does not help writers do the same. Existing work on scalable writers using RCU has focused primarily on partitioning, both via disjoint-access parallelism (such as fine-grained locking) and via CPU-local storage; however, this does not eliminate all sources of synchronization overhead except in special cases. General-purpose techniques for scalable writers remain the highest priority for future work.

Many writers begin by traversing the data structure to locate the context for their write. Making those initial loads relativistic would significantly reduce writer conflicts, increasing concurrency for partitionable data structures. However, since writers rely on that context to make their modifications, notably by attaching new items to existing parts of a data structure, the writers may need stronger consistency guarantees for their loads.

Structures such as queues and stacks primarily provide write operations, and their layout in memory implies a strict ordering. However, semantically such structures need not always require strict ordering. For instance, a process scheduling queue or UDP packet queue may receive new insertions in

any order, yet those queues enforce a serialized order after each insertion. A more scalable queue implementation might choose to preserve only causal ordering relationships, and otherwise leave item ordering non-deterministic until dequeued.

Chapter 4 presents the relativistic causal ordering property and barrier placement rules via informal supporting arguments, not formal proofs. The application of formal verification methods would allow the construction of a formal proof. This would add further precision to the barrier placement rules and data-structure semantics, and potentially aid in future expansion of this methodology.

The barrier translations specified in chapter 5 would also benefit from formal verification via hardware architectures with formal memory model specifications, such as x86-TSO [70].

The relativistic causal ordering model assumes that no readers perform cyclic traversals. Such traversals would allow readers to see changes in progress, breaking the illusion of reader atomicity. This requirement prohibits a small but significant set of algorithms, most notably those involving general graph traversal; such algorithms also prove exceedingly difficult for fine-grained locking. Some research exists suggesting approaches for the use of Read-Copy Update on cyclic data structures, including graphs, with additional reader overhead required to maintain consistency [51]. Adaptation of this research could extend the relativistic causal ordering property to cyclic data structures.

Alternatively, many cyclic read algorithms could cope with the semantics of non-repeatable loads, which would allow those readers to achieve scalability. Future work should explore cyclic traversals further, defining precise semantics for the ordering of loads and stores in such traversals, and proposing rules for the construction of readers that can cope with the non-atomic ordering semantics.

Bibliography

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

[2] Mustaque Ahamad, Gil Neiger, James Burns, Prince Kohli, and Phillip Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9:37–49, 1995.

[3] Andrei Alexandrescu and Maged Michael. Lock-free data structures with hazard pointers. *C/C++ Users Journal*, November 2004. Available from: `http://ddj.com/184401890`.

[4] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated. In *Proceedings of the ACM POPL'11*, 2011.

[5] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In

SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, pages 69–78, 2009.

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

[7] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[8] Cliff Click. A Lock-Free Hash Table. In *JavaOne Conference*, 2007.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 11: Hash Tables. MIT Press, second edition, 2001.

[10] Mathieu Desnoyers. *Low-Impact Operating System Tracing*. PhD thesis, École Polytechnique de Montréal, December 2009. Available from: `http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf`.

[11] Dave Dice, Yossi Lev, Mark Moir, and Dan Nussbaum. Early experience with a commericial hardware transactional memory implementation. In *Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, Washing-

ton, DC, USA, March 2009. Available from: http://research.sun.com/scalable/pubs/ASPLOS2009-RockHTM.pdf.

[12] Dave Dice and Nir Shavit. TLRW: return of the read-write lock. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

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

[14] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[15] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why stm can be more than a research toy. *Communications of the ACM*, 54:70–77, April 2011.

[16] Robert Ennals. Software transactional memory should not be obstruction-free. Technical report, Intel Research, 2006. Available from: http://berkeley.intel-research.net/rennals/pubs/052RobEnnals.pdf.

[17] Jason Fried and David Heinemeier Hansson. *Rework*. Crown Business, 2010.

[18] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 18(1), July 2005.

[19] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the second USENIX symposium on Operating Systems Design and Implementation*, pages 123–136, 1996.

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

[21] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Notices*, 38(11):388–402, 2003.

[22] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2001.

[23] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[24] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA04: Symposium on Parallelism in Algorithms and Architectures*, pages 206–215, 2004.

[25] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[26] Maurice Herlihy. The transactional manifesto: Software engineering and non-blocking synchronization. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 280–280, 2005.

[27] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[28] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*, chapter 11: Concurrent Hashing. Morgan Kaufmann Publishers, 2008.

[29] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[30] Maurice Herlihy, Nir Shavit, and Orli Waarts. Linearizable counting networks. *Distributed Computing*, 9:193–203, 1996.

[31] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[32] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

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

[34] Philip W. Howard and Jonathan Walpole. Relativistic red-black trees. Technical Report 1006, Portland State University, 2011. `http://www.cs.pdx.edu/pdfs/tr1006.pdf`.

[35] Wilson C. Hsieh and William E. Weihl. Scalable reader-writer locks for parallel systems. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 216–230, March 1992.

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

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

[38] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17:453–455, August 1974.

[39] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[40] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23:105–117, February 1980.

[41] Hanna Linder, Dipankar Sarma, and Maneesh Soni. Scalability of the directory entry cache. In *Ottawa Linux Symposium*, pages 289–300, June 2002.

[42] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Princeton University, 1988.

[43] Virendra J. Marathe and Michael L. Scott. A qualitative survey of modern software transactional memory systems. Technical report, Department of Computer Science, University of Rochester, June 2004.

[44] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT: the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[45] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. *ACM Operating Systems Review*, 26, April 1992.

[46] Sally A. McKee. Reflections on the memory wall. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, 2004.

[47] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available from: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf`.

[48] Paul E. McKenney. Using promela and spin to verify parallel algorithms. *Linux Weekly News*, August 2007. Available from: `https://lwn.net/Articles/243851/`.

[49] Paul E. McKenney. Hierarchical RCU. *Linux Weekly News*, November 2008. Available from: `https://lwn.net/Articles/305782/`.

[50] Paul E. McKenney. RCU: The Bloatwatch Edition. *Linux Weekly News*, March 2009. Available from: `https://lwn.net/Articles/323929/`.

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

[52] Paul E. McKenney. Lockdep-RCU. *Linux Weekly News*, February 2010. Available from: `https://lwn.net/Articles/371986/`.

[53] Paul E. McKenney, Maged M. Michael, Josh Triplett, and Jonathan Walpole. Why the grass may not be greener on the other side: A comparison of locking vs. transactional memory. *ACM Operating Systems Review*, 44(3), July 2010.

[54] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002. Available from: `http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz`.

[55] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling dcache with RCU. *Linux Journal*, 2004(117), 2004.

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

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

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

[59] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

[60] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51:1–26, 1998.

[61] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[62] Gordon E. Moore. Moore's law at 40. *Understanding Moore's law: four decades of innovation*, 2006.

[63] Nick Piggin. Lockless radix-tree, March 2006. Available from: `http://kernel.org/pub/linux/kernel/people/npiggin/patches/lockless/2.6.16-rc5/radix-intro.pdf`.

[64] Nick Piggin. 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.

[65] Relativistic programming wiki. Available from: `http://wiki.cs.pdx.edu/rp/`.

[66] Chistopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bhandari, and Emmett Witchel. TxLinux: Using and managing hardware transactional memory in an operating system. In *SOSP'07: Twenty-First ACM Symposium on Operating Systems*

Principles, October 2007. Available from: `http://www.sosp2007.org/papers/sosp056-rossbach.pdf`.

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

[68] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–133, 2007.

[69] William N. Scherer III and Michael L. Scott. Nonblocking concurrent data structures with condition synchronization. In *Proceedings of the International Symposium on Distributed Computing (DISC*, pages 174–187, 2004.

[70] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53:89–97, July 2010.

[71] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53:379–405, May 2006.

[72] Michael F. Spear, Maged M. Michael, and Michael L. Scott. Inevitability mechanisms for software transactional memory. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[73] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.

[74] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, SOSP 1995, pages 172–182, 1995.

[75] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[76] Josh Triplett. Scalable Concurrent Hash Tables via Relativistic Programming. In *Linux Plumbers Conference 2009*, Portland, OR, September 2009.

[77] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Scalable Concurrent Hash Tables via Relativistic Programming. *ACM Operating Systems Review*, 44(3), July 2010.

[78] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Pro-*

ceedings of the 2011 USENIX Annual Technical Conference, USENIX ATC '11, 2011.

[79] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52:40–44, January 2009.

[80] Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, New York, NY, USA, February 2008. Available from: `http://www.cs.wisc.edu/multifacet/papers/transact08_txlock.pdf`.

[81] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, 2008.

[82] William A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

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

[84] Dieter Zöbel. The deadlock problem: a classifying bibliography. *ACM Operating Systems Review*, 17:6–15, October 1983.