Read-Log-Update

A Lightweight Synchronization Mechanism for Concurrent Programming

Alexander Matveev [*] Nir Shavit [*†] Pascal Felber [‡] Patrick Marlier [‡]

[*] MIT [†] Tel-Aviv University [‡] University of Neuchâtel

Abstract

This paper introduces *read-log-update* (RLU), a novel extension of the popular read-copy-update (RCU) synchronization mechanism that supports scalability of concurrent code by allowing unsynchronized sequences of reads to execute concurrently with updates. RLU overcomes the major limitations of RCU by allowing, for the first time, concurrency of reads with multiple writers, and providing automation that eliminates most of the programming difficulty associated with RCU programming. At the core of the RLU design is a logging and coordination mechanism inspired by software transactional memory algorithms. In a collection of micro-benchmarks in both the kernel and user space, we show that RLU both simplifies the code and matches or improves on the performance of RCU. As an example of its power, we show how it readily scales the performance of a real-world application, Kyoto Cabinet, a truly difficult concurrent programming feat to attempt in general, and in particular with classic RCU.

1. Introduction

Context. An important paradigm in concurrent data structure scalability is to support *read-only traversals*: sequences of reads that execute without any synchronization (and hence require no memory fences and generate no contention [20]). The gain from such unsynchronized traversals is significant because they account for a large fraction of operations in many data structures and applications [20, 34].

The popular *read-copy-update* (RCU) mechanism of McKenney and Slingwine [28] provides scalability by enabling this paradigm. It allows read-only traversals to proceed concurrently with updates by creating a copy of the data structure being modified. Readers access the unmodified data structure while updaters modify the copy. The key to RCU is that once modifications are complete, they are installed using a single pointer modification in a way that does not interfere with ongoing readers. To avoid synchronization, updaters wait until all pre-existing readers have finished their operations, and only then install the modified copies. This barrier-based mechanism allows for simple epoch-based reclamation [17] of the old copies, and the mechanism as a whole eliminates many of the atomic read-modify-write instructions, memory barriers, and cache misses that are so expensive on modern multicore systems.

RCU is supported in both user-space and in the kernel. It has been widely used in operating system programming (over 6'500 API calls in the Linux kernel as of 2013 [30]) and concurrent applications (as reported at http://urcu.so/, user-space RCU is notably used in a DNS server, a networking toolkit, and a distributed storage system).

Motivation. Despite its many benefits, RCU programming is not a panacea and its performance has some significant limitations. First, it is quite complex to use. It requires the programmer to implement dedicated code in order to duplicate every object it modifies, and ensure that the pointers inside the copies are properly set to point to the correct locations, before finally connecting this set of copies using a single atomic pointer assignment (or using lock-protected critical sections). This complexity is evidenced by the fact that there are not many RCU-enhanced data structures beyond simple linked-lists, and the few other RCU-ed data structures are innovative algorithms published in research papers [2, 5]. A classic example of this difficulty is the doubly linked list implementation in the Linux Kernel, in which threads are only allowed to traverse the list in the forward direction (the backward direction is unsafe and may return in inconsistent sequence of items) because of the limitations of the single pointer manipulation semantics [5, 26].

Second, RCU is optimized for a low number of writers. The RCU kernel implementation [25, 27] reduces contention and latency, but does not provide concurrency among writers.

Third, threads using RCU experience delays when waiting for readers to complete their operations. This makes

SOSP'15, October 4-7, 2015, Monterey, CA, USA.

http://dx.doi.org/10.1145/2815400.2815406

RCU potentially unfit for time-critical applications. Recent work by Arbel and Morrison [3] suggests how to reduce these delays by having the programmer provide RCU with predicates that define the access patterns to data structures.

Our objective is to propose an alternative mechanism that will be simpler for the programmer while matching or improving on the scalability obtainable using RCU.

Contributions. In this paper, we propose *read-log-update* (RLU), a novel extension of the RCU framework that supports read-only traversals concurrently with multiple updates, and does so in a semi-automated way. Unlike with RCU, adding support for concurrency to common sequential data structures is straightforward with RLU because it removes from the programmer the burden of handcrafting the concurrent copy management using only single pointer manipulations. RLU can be implemented in such a way that it remains API-compatible with RCU. Therefore, it can be used as a drop-in replacement in the large collection of RCU-based legacy code.

In a nutshell, RLU works as follows. We keep the overall RCU barrier mechanism with updaters waiting until all pre-existing readers have finished their operations, but replace the hand-crafted copy creation and installation with a clock-based logging mechanism inspired by the ones used in software transactional memory systems [1, 8, 33]. The biggest limitation of RCU is that it cannot support multiple updates to the data structure because it critically relies on a single atomic pointer swing. To overcome this limitation, in RLU we maintain an object-level write-log per thread, and record all modifications to objects in this log. The objects being modifiead are automatically copied into the write-log and manipulated there, so that the original structures remain untouched. These locations are locked so that at most one thread at a time will modify them.

The RLU system maintains a global clock [8, 33] that is read at the start of every operation, and used to decide which version of the data to use, the old one or the logged one. Each writer thread, after completing the modifications to the logged object copies, commits them by incrementing the clock to a new value, and waits for all readers that started before the clock increment (have a clock value lower than it) to complete. This modification of the global clock using a single operation has the effect of making all the logged changes take effect at the same time. In other words, if RCU uses a single read-modify-write to switch a pointer to one copy, in RLU the single read-modify-write of the clock switches multiple object copies at once.

The simple barrier that we just described, where a writer increments the counter and waits for all readers with smaller clock values, can be further improved by applying a deferral mechanism. Instead of every writer incrementing the clock, we have writers increment it only if they have an actual conflict with another writer or with their own prior writes. These conflicts are detectable via the locks placed on objects. Thus, writers typically complete without committing their modifications, and if conflict is detected, only then the clock is incremented. This deferred increment applies all the deferred modifications at once, with a much lower overhead since the number of accesses to the shared clock is lowered, and more importantly, the number of times a writer must wait for readers to complete is significantly reduced in comparison to RCU. In fact, this deferral achieves in an automated way almost the same improvement in delays as the RCU predicate approach of Arbel and Morrison [3].

We provide two versions of RLU that differ in the way writers interact. One version allows complete writer concurrency and uses object locks which writers attempt to acquire before writing. The other has the system serialize all writers and thus guarantee that they will succeed. The writers' operations in the serialized version can further be parallelized using hardware transactions with a software fallback [24, 36]. Our RLU implementation is available as open source from https://github.com/rlu-sync.

We conducted an in-depth performance study and comparison of RLU against kernel and user-space RCU implementations. Our findings show that RLU performs as well as RCU on simple structures like linked lists, and outperforms it on more complex structures like search trees. We show how RLU can be readily used to implement a doubly linked list with full semantics (of atomicity), allowing threads to move forward or backward in the list—this is a task that to date is unattainable with RCU since the doubly linked list in the Linux kernel restricts threads to only move forward or risk viewing an inconsistent sequence of items in the list [26]. We also show how RLU can be used to increase the scalability of a real-world application, Kyoto Cabinet, that would be difficult to apply RCU to because it requires to modify several data structures concurrently, a complex feat if one can only manipulate one pointer at a time. Replacing the use of reader-writer locks with RLU improves the performance by a factor of 3 with 16 hardware threads.

2. Background and Related Work

RLU owes much of its inspiration to the read-copy-update (RCU) algorithm, introduced by McKenney and Slingwine [28] as a solution to lock contention and synchronization overheads in read-intensive applications. Harris et al. [16] and Hart et al. [17] used RCU ideas for epoch-based explicit memory reclamation schemes. A formal semantics of RCU appears in [13, 15].

RCU minimizes synchronization overhead for sequences of reads traversing a data structure, at the price of making the code of writers more complex and slower. As noted in the introduction, the core idea of RCU is to duplicate an object each time it is modified and perform the modifications on the private copy. In this way, writers do not interfere with readers until they atomically update shared data structures, typically by "connecting" their private copies. To ensure consistency,

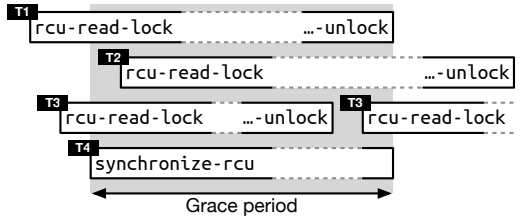

Figure 1. Principle of RCU: read-side critical sections and grace periods.

such updates must be done at a safe point when no reader can potentially hold a reference to the old data.

RCU readers delimit their read operations by calls to rcu_read_lock() and rcu_read_unlock(), which essentially define read-side critical sections. RCU-protected data structures are accessed in critical sections using rcu_dereference() and rcu_assign_pointer(), which ensure dependency-ordered loads and stores by adding memory barriers as necessary.[1] When they are not inside a critical section, readers are said to be in a *quiescent state*. A period of time during which every thread goes through at least one quiescent state is called a *grace period*. The key principle of RCU is that, if an updater removes an element from an RCU-protected shared data structure and waits for a grace period, there can be no reader still accessing that element. It is therefore safe to dispose of it. Waiting for a grace period can be achieved by calling synchronize_rcu(). The basic principle of read-side critical sections and grace periods is illustrated in Figure 1, with three reader threads (T_1, T_2, T_3) and one writer thread (T_4). As the grace period starts while T_1 and T_3 are in read-side critical sections, T_4 needs to wait until both other threads exit their critical section. In contrast, T_2 starts a critical section after the call to synchronize_rcu() and hence cannot hold a reference to old data. Therefore the grace period can end before the critical section completes (and similarly for the second critical section of T_3 started during the grace period).

An emblematic use case of RCU is for reclaiming memory in dynamic data structures. To illustrate how RCU helps in this case, consider a simple linked list with operations to add, remove, and search for integer values. The (simplified) code of the search() and remove() methods is shown in Listing 1. It is important to note that synchronization between writers is not managed by RCU, but must be implemented via other mechanisms such as locks. Another interesting observation is how similar RCU code is to a reader-writer lock in this simple example: read-side critical sections correspond to shared locking while writers acquire the lock in exclusive mode.

A sample run is illustrated in Figure 2, with thread T_1 searching for element c while thread T_2 concurrently removes element b. T_1 enters a read-side critical section while T_2 locates the element to remove. The corresponding node

[1] Note that on many architectures rcu_dereference() calls are replaced by simple loads and hence do not add any overhead.

```
1  int search(int v) {
2    node_t *n;
3    rcu_read_lock();
4    n = rcu_dereference(head->next);
5    while (n != NULL && n->value != v)
6      n = rcu_dereference(n->next);
7    rcu_read_unlock();
8    return n != NULL;
9  }

10 int remove(int v) {
11   node_t *n, *p, *s;
12   spin_lock(&writer_lock);
13   for (p = head, n = rcu_dereference(p->next);
14     n != NULL && n->value != v;
15     p = n, n = rcu_dereference(n->next));
16   if (n != NULL) {
17     s = rcu_dereference(n->next);
18     rcu_assign_pointer(p->next, s);
19     spin_unlock(&writer_lock);
20     synchronize_rcu();
21     kfree(n);
22     return 1;
23   }
24   spin_unlock(&writer_lock);
25   return 0;
26 }
```

Listing 1. RCU-based linked list.

is unlinked from the list by T_2 while T_1 traverses the list. T_2 cannot yet free the removed node, as it may still be accessed by other readers; hence it calls synchronize_rcu(). T_1 continues its traversal of the list while T_2 still waits for the end of the grace period. Finally T_1 exits the critical section and the grace period completes, which allows T_2 to free the removed node.

RCU has been supported in the Linux kernel since 2002 [29] and has been heavily used within the operating system. The kernel implementation is very efficient because, by running in kernel space and being tightly coupled to the scheduler, it can use a high-performance quiescent-state-based reclamation strategy wherein each thread periodically announces that it is in a quiescent state. The implementation also provides a zero-overhead implementation of rcu_read_lock() and rcu_read_unlock() operations. As a drawback, synchronization delays for writers can become unnecessarily long as they are tied to scheduling periods. Furthermore, the Linux kernel's RCU implementation is not suitable for general-purpose user-space applications.

User-space RCU [6] is another popular library implementing the RCU algorithms entirely in user space. It provides several variants of synchronization algorithms (using quiescent-state-based reclamation, signals, or memory barriers) offering different trade-offs in terms of read-/write-side overheads and usage constraints. User-space RCU is widely applicable for general-purpose code but in general does not perform as well as the kernel implementation.

Our implementation of RLU is based on the use of a global clock mechanism inspired by the one used in some software transactional memory systems [8, 33], which notably use lock-based designs to avoid some of the costs inherent to lock-free or obstruction-free algorithms [11, 22]. The global clock is a memory location that is updated by

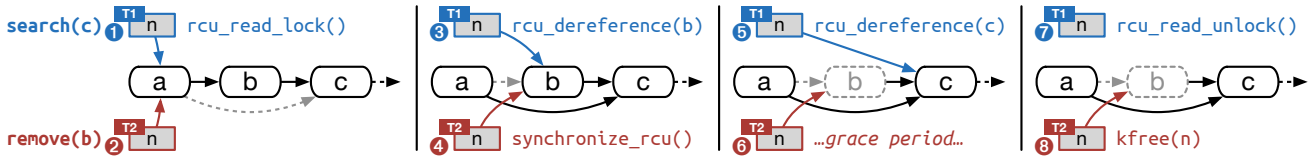

Figure 2. Concurrent search and removal with the RCU-based linked list.

threads when they wish to establish a synchronization point. All threads use the clock as a reference point, time-stamping their operations with this clock's value. The observation in [1, 8, 33] is that despite concurrent clock updates and multiple threads reading the global clock while it is being updated, the overall contention and bottlenecking it introduces is typically minimal.

3. The RLU Algorithm

In this section we describe the design and implementation of RLU.

3.1 Basic Idea

For simplicity of presentation, we first assume in this section that write operations execute serially, and later show various programming patterns that allow us to introduce concurrency among writers.

RLU provides support for multiple object updates in a single operation by combining the quiescence mechanism of RCU with a global clock and per thread object-level logs. All operations read the global clock when they start, and use this clock to dereference shared objects. In turn, a write operation logs each object it modifies in a per thread write-log: to modify an object, it first copies the object into the write-log, and then manipulates the object copy in this log. In this way, write modifications are hidden from concurrent reads, and to avoid conflicts with concurrent writes, each object is also locked before its first modification (and duplication). Then, to commit the new object copies, a write operation increments the global clock, which effectively splits operations into two sets: (1) *old operations* that started before the clock increment, and (2) *new operations* that start after the clock increment. The first set of operations will read the old object copies while the second set will read the new object copies of this writer. Therefore, in the next step, the writer waits for old operations to finish by executing the RCU-style quiescence loop, while new operations "steal" new object copies of this writer by accessing the per thread write-log of this writer. After the completion of old operations, no other operation may access the old object memory locations, so the writer can safely write back the new objects from the writer-log into the memory, overwriting the old objects. It can then release the locks.

Figure 3 depicts how RLU provides multiple object updates in one operation. In the figure, execution flows from top to bottom. Thread T_2 updates objects O_2 and O_3, whereas threads T_1 and T_3 only perform reads. Initially, the global clock is 22, and T_2 has an empty write-log and a local write-clock variable that holds ∞ (maximum 64-bit integer value). These per-thread write-clocks are used by the stealing mechanism to ensure correctness (details follow).

In the top figure, threads T_1 and T_2 start by reading the global clock and copying its value to their local clocks, and then proceed to reading objects. In this case, none of the objects is locked, so the reads are performed directly from the memory.

In the middle figure, T_2 locks and logs O_2 and O_3 before updating these objects. As a result, O_2 and O_3 are copied into the write-log of T_2, and all modifications are re-routed into the write-log. Meanwhile, T_1 reads O_2 and detects that this object is locked by T_2. T_1 must thus determine whether it needs to steal the new object copy. To that end, T_1 compares its local clock with the write clock of T_2, and only when the local clock is greater than or equal to the write-clock of T_2 does T_1 steal the new copy. This is not the case in the depicted scenario, hence T_1 reads the object directly from the memory.

In the bottom figure, T_2 starts the process of committing new objects. It first computes the next clock value, which is 23, and then installs this new value into the write-clock and global-clock (notice that the order here is critical). At this point, as we explained before, operations are split into "old" and "new" (before and after the clock increment), so T_2 waits for the old operations to finish. In this example, T_2 waits for T_1. Meanwhile, T_3 reads O_2 and classifies this operation as new by comparing its local clock with the write-clock of T_2; it therefore "steals" the new copy of O_2 from the write-log of T_2. In this way, new operations read only new object copies so that, after T_2 wait completes, no-one can read the old copies and it is safe to write back the new copies of O_2 and O_3 to memory.

3.2 Synchronizing Write Operations

The basic idea of RLU described above provides read-write synchronization for object accesses. It does not however ensure write-write synchronization, which must be managed by the programmer if needed (as with RCU). A simple way to synchronize writers is to execute them serially, without any concurrency. In this case, the benefit is simplicity of code and the concurrency that does exist between read-only and write operations. On the other hand, the drawback is a lack of scalability.

Another approach is to use fine-grained locks. In RLU, each object that a writer modifies is logged and locked by

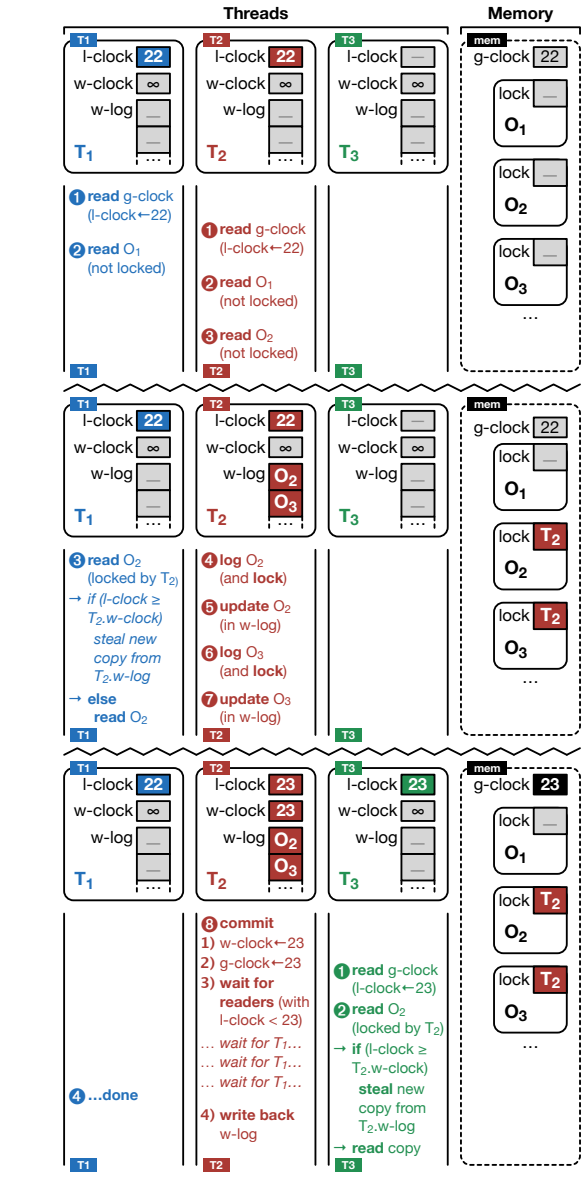

Figure 3. Basic Principle of RLU.

the RLU mechanism. Programmers can therefore use this locking process to coordinate write operations. For example, in a linked-list implementation, instead of grabbing a global lock for each writer, a programmer can use RLU to traverse the linked list, and then use the RLU mechanism to lock the target node and its predecessor. If the locking fails, then the operation restarts, otherwise the programmer can proceed and modify the target node (e.g., insertion or removal) and release the locks.

3.3 Fine-grained Locking Using RLU

Programmers can use RLU locks as a fine-grained locking mechanism, in the same way they use standard locks. However, RLU locks are much easier to use due to the fact that all object reads and writes execute inside "RLU protected" sections that are subject to the RCU-based quiescence mechanism of each writer. This means that when some thread reads or writes (locks and logs) objects, no other concurrent thread may overwrite any of these objects. With RLU, after the object lock has been acquired, no other action is necessary whereas, with standard locks, one needs to execute post-lock customized verifications to ensure that the state of the object is still the same as it was before locking.

3.4 RLU Metadata

Global. RLU maintains a global clock and a global array of threads. The global array is used by the quiescence mechanism to identify the currently active threads.

Thread. RLU maintains two write-logs, a run counter, and a local clock and write clock for each thread. The write-logs hold new object copies, the run counter indicates when the thread is active, and the local clock and write clock control the write-log stealing mechanism of threads. In addition, each object copy in the write-log has a header that includes: (1) a thread identifier, (2) a pointer to the actual object, (3) the object size, and (4) a special pointer value that indicates this is a copy (constant).

Object. RLU attaches a header for each object, which includes a single pointer that points to the copy of this object in a write-log. If this pointer is NULL, then there is no copy and the object is unlocked.

In our implementation, we attach a header to each object by hooking the malloc() call with a call to rlu_alloc() that allocates each object with the attached header. In addition, we also hook the free() call with rlu_free() to ensure proper deallocation of objects that include headers. Note that any allocator library can be used with RLU.

We use simple macros to access and modify the metadata headers of an object. First, we use get_copy(obj) to get *ptr-copy*: the value of the pointer (to copy) that resides in the header of *obj*. Then, we use this ptr-copy as a parameter in macros: (1) is_unlocked(ptr-copy) that checks if the object is free, (2) is_copy(ptr-copy) that checks if this object is a copy in a write-log, (3) get_actual(obj) that returns a pointer to the actual object in memory in case this is a copy in a write-log, and (4) get_thread_id(ptr-copy) that returns the identifier of the thread that currently locked this object.

We use 64-bit clocks and counters to avoid overflows and initialize all RLU metadata to zero. The only exception is write clocks of threads, that are initialized to ∞ (maximum 64-bit value).

3.5 RLU Pseudo-Code

Algorithm 1 presents the pseudo-code for the main functions of RLU. An RLU protected section starts by calling rlu_reader_lock() that registers the thread: it increments the run counter and initializes the local clock to the global clock. Then, during execution of the section, it dereferences each object by calling the rlu_dereference() function, which first

Algorithm 1 RLU pseudo-code: main functions

```
1:  function RLU_READER_LOCK(ctx)
2:      ctx.is-writer ← false
3:      ctx.run-cnt ← ctx.run-cnt +1                    ▷ Set active
4:      memory fence
5:      ctx.local-clock ← global-clock                  ▷ Record global clock

6:  function RLU_READER_UNLOCK(ctx)
7:      ctx.run-cnt ← ctx.run-cnt +1                    ▷ Set inactive
8:      if ctx.is-writer then
9:          RLU_COMMIT_WRITE_LOG(ctx)                   ▷ Write updates

10: function RLU_DEREFERENCE(ctx, obj)
11:     ptr-copy ← GET_COPY(obj)                        ▷ Get copy pointer
12:     if IS_UNLOCKED(ptr-copy) then                   ▷ Is free?
13:         return obj                                  ▷ Yes ⇒ return object
14:     if IS_COPY(ptr-copy) then                       ▷ Already a copy?
15:         return obj                                  ▷ Yes ⇒ return object
16:     thr-id ← GET_THREAD_ID(ptr-copy)
17:     if thr-id = ctx.thr-id then                     ▷ Locked by us?
18:         return ptr-copy                             ▷ Yes ⇒ return copy
19:     other-ctx ← GET_CTX(thr-id)                     ▷ No ⇒ check for steal
20:     if other-ctx.write-clock ≤ ctx.local-clock then
21:         return ptr-copy                             ▷ Stealing ⇒ return copy
22:     return obj                                      ▷ No stealing ⇒ return object

23: function RLU_TRY_LOCK(ctx, obj)
24:     ctx.is-writer ← true                            ▷ Write detected
25:     obj ← GET_ACTUAL(obj)                           ▷ Read actual object
26:     ptr-copy ← GET_COPY(obj)                        ▷ Get pointer to copy
27:     if ¬ IS_UNLOCKED(ptr-copy) then
28:         thr-id ← GET_THREAD_ID(ptr-copy)
29:         if thr-id = ctx.thr-id then                 ▷ Locked by us?
30:             return ptr-copy                         ▷ Yes ⇒ return copy
31:         RLU_ABORT(ctx)                              ▷ No ⇒ retry RLU section
32:     obj-header.thr-id ← ctx.thr-id                  ▷ Prepare write-log
33:     obj-header.obj ← obj
34:     obj-header.obj-size ← SIZEOF(obj)
35:     ptr-copy ← LOG_APPEND(ctx.write-log, obj-header)
36:     if ¬ TRY_LOCK(obj, ptr-copy) then               ▷ Try to install copy
37:         RLU_ABORT(ctx)                              ▷ Failed ⇒ retry RLU section
38:     LOG_APPEND(ctx.write-log, obj)                  ▷ Locked ⇒ copy object
39:     return ptr-copy

40: function RLU_CMP_OBJS(ctx, obj1, obj2)
41:     return GET_ACTUAL(obj1) = GET_ACTUAL(obj2)

42: function RLU_ASSIGN_PTR(ctx, handle, obj)
43:     ∗handle ← GET_ACTUAL(obj)

44: function RLU_COMMIT_WRITE_LOG(ctx)
45:     ctx.write-clock ← global-clock +1               ▷ Enable stealing
46:     FETCH_AND_ADD(global-clock, 1)                  ▷ Advance clock
47:     RLU_SYNCHRONIZE(ctx)                            ▷ Drain readers
48:     RLU_WRITEBACK_WRITE_LOG(ctx)                    ▷ Safe to write back
49:     RLU_UNLOCK_WRITE_LOG(ctx)
50:     ctx.write-clock ← ∞                             ▷ Disable stealing
51:     RLU_SWAP_WRITE_LOGS(ctx)                        ▷ Quiesce write-log

52: function RLU_SYNCHRONIZE(ctx)
53:     for thr-id ∈ active-threads do
54:         other ← GET_CTX(thr-id)
55:         ctx.sync_cnts[thr-id] ← other.run-cnt
56:     for thr-id ∈ active-threads do
57:         while true do                               ▷ Spin loop on thread
58:             if ctx.sync-cnts[thr-id] is even then
59:                 break                               ▷ Not active
60:             other ← GET_CTX(thr-id)
61:             if ctx.sync-cnts[thr-id] ≠ other.run-cnt then
62:                 break                               ▷ Progressed
63:             if ctx.write-clock ≤ other.local-clock then
64:                 break                               ▷ Started after me

65: function RLU_SWAP_WRITE_LOGS(ctx)
66:     ptr-write-log ← ctx.write-log-quiesce           ▷ Swap pointers
67:     ctx.write-log-quiesce ← ctx.write-log
68:     ctx.write-log ← ptr-write-log

69: function RLU_ABORT(ctx, obj)
70:     ctx.run-cnt ← ctx.run-cnt +1                    ▷ Set inactive
71:     if ctx.is-writer then
72:         RLU_UNLOCK_WRITE_LOG(ctx)                   ▷ Unlock
73:     RETRY                                           ▷ Specific retry code
```

checks whether the object is unlocked or a copy and, in that case, returns the object. Otherwise, the object is locked by some other thread, so the function checks whether it needs to steal the new copy from the other thread's write-log. For this purpose, the current thread checks if its local clock is greater than or equal to the write clock of the other thread and, if so, it steals the new copy. Notice, that the write-clock of a thread is initially ∞, so stealing from a thread is only possible when it updates the write-clock during the commit.

Next, the algorithm locks each object to be modified by calling rlu_try_lock(). First, this function sets a flag to indicate that this thread is a writer. Then, it checks if the object is already locked by some other thread, in which case it fails and retries. Otherwise, it starts the locking process that first prepares a write-log header for the object, and then installs a pointer to object copy by using compare-and-swap (CAS) instruction. If the locking succeeds, the thread copies the object to the write-log, and returns a pointer to the newly created copy. Note that the code also uses the rlu_cmp_objs() and rlu_assign_ptr() functions that hide the internal implementation of object duplication and manipulation.

An RLU protected section completes by calling rlu_reader_unlock() that first unregisters the thread by incrementing the run counter, and then checks if this thread is a writer. If so, it calls rlu_commit_write_log() that increments the global clock and sets the write clock of the thread to the new clock to enable write-log stealing. As we mentioned before, the increment of the global clock is the critical point at which all new object copies of the write-log become visible at once (atomically) to all concurrent RLU protected sections that start after the increment. As a result, in the next step, the function executes rlu_synchronize() in order to wait for the completion of the RLU protected sections that started before the increment of the global clock, i.e., that currently active (have odd run counter) and have a local clock smaller than the write clock of this thread. Thereafter, the function writes back the new object copies from the write-log to the actual memory, unlocks the objects, and sets the write-clock back to ∞ to disable stealing. Finally, It is important to notice that an additional quiescence call is necessary to clear the current write-log from threads that steal copies from this write-log, before the given thread can reuse this write-log once again. For this purpose, the function swaps the current write-log with a new write-log, and only after the next rlu_synchronize() call is completed, the current write-log is swapped back and reused.

3.6 RLU Correctness

The key for correctness is to ensure that *RLU protected sections always execute on a consistent memory view* (snapshot). In other words, an RLU protected section must be resistant to possible concurrent overwrites of objects that it currently reads. For this purpose, RLU sections sample the global clock on start. RLU writers modify object *copies* and commit these copies by updating the global clock.

More precisely, correctness is guaranteed by a combination of three key mechanisms in Algorithm 1.

1. On commit, an RLU writer first increments the global clock (line 46), which effectively "splits" all concurrent RLU sections into *old sections* that observed the old global clock and *new sections* that will observe the new clock. The check of the clock in the RLU dereference function (line 20) ensures that new sections can only read object copies of modified objects (via stealing), while old sections continue to read the actual objects in the memory. As a result, after old sections complete, no other thread can read the actual memory of modified objects and it is safe to overwrite this memory with the new object copies. Therefore, in the next steps, the RLU writer first executes the RLU synchronize call (line 47), which waits for old sections to complete, and only then write backs new object copies to the actual memory (line 48).
2. After the write-back of RLU writer, the write-log cannot be reused immediately since other RLU sections may be still reading from it (via stealing). Therefore, the RLU writer swaps the current write-log L_1 with a new write-log L_2. In this way, L_2 becomes the active write-log for the next writer, and only after this next writer arrives to the commit and completes its RLU synchronize call, it swaps back L_1 with L_2. This RLU synchronize call (line 46) effectively "splits" concurrent RLU sections into *old sections* that may be reading from the write-log L_1 and *new sections* that cannot be reading from the write-log L_1. Therefore, after this RLU synchronize call completes, L_1 cannot be read by any thread, so it is safe to swap it back and reuse. Notice that this holds since the RLU writer of $L1$ completes by disabling stealing from $L1$: it unlocks all modified objects of $L1$ and sets the write-clock back to ∞ (line 49).
3. Finally, to avoid write-write conflicts between writers, each RLU writer locks each object it wants to modify (line 36).

The combination of these tree mechanisms ensures that an RLU protected section that starts with a global clock value g will not be able to see any concurrent overwrite that was made for global clock value $g' > g$. As a result, the RLU protected section always executes on a consistent memory view that existed at global time g.

3.7 RLU Deferring

As shown in the pseudo-code, each RLU writer must execute one RLU synchronize call during the process of commit. The relative penalty of RLU synchronize calls depends on the specific workload, and our tests show that, usually, it becomes expensive when operations are short and fast. Therefore, we implement the RLU algorithm in a way that allows us to defer the RLU synchronize calls to as late a point as possible and only execute them when they are necessary. Note that a similar approach is used in flat combining [19] and OpLog [4]. However, they defer on the level of data-structure operations, while RLU defers on the level of individual object accesses.

RLU synchronize deferral works as follows. On commit, instead of incrementing the global clock and executing RLU synchronize, the RLU writer simply saves the current write-log and generates a new log for the next writer. In this way, RLU writers execute without blocking on RLU synchronize calls, while aggregating write-logs and locks of objects being modified. The RLU synchronize call is actually only necessary when a writer tries to lock an object that is already locked. Therefore, only in this case, the writer sends a "sync request" to the conflicting thread to force it to release its locks, by making the thread increment the global clock, execute RLU synchronize, write back, and unlock.

Deferring RLU synchronize calls and aggregating write-logs and locks over multiple write operations provides several advantages. First, it significantly reduces the amount of RLU synchronize calls, effectively limiting them to the number of actual write-write data conflicts that occur during

runtime execution. In addition, it significantly reduces the contention on the global clock, since this clock only gets updated after RLU synchronize calls, which now executes less often. Moreover, the aggregation of write-logs and the deferral of the global clock update defers the stealing process to a later time, which allows threads to read from memory without experiencing cache misses that would otherwise occur when systematically updating data in memory.

We note that the described deferral mechanism is sensitive to scheduling constraints. For example, a lagging thread may delay other threads that wait for a sync response from this thread. In our benchmarks we have not experienced such behavior, however, it is possible to avoid dependency on scheduling constraints by allowing a waiting thread to help the other thread: the former can simply execute the sync and write-back for the latter.

Also, we point out that these optimizations work when the code that executes outside of RLU protected sections can tolerate deferred updates. In practice, this requires to define specific sync points in the code where it is critical to see the most recent updates. In general, as we later explain, this optimization is more significant for high thread counts, such as when benchmarking the Citrus tree on an 80-way 4 socket machine (see Section 4). In all other benchmarks that execute on a 16-way processor, using deferral provides modest improvements over the simple RLU algorithm.

3.8 RLU Implementation

We implement Algorithm 1 and provide two flavors of RLU: (1) coarse-grained and (2) fine-grained. The coarse-grained flavor has no support for RLU deferral and it provides writer locks that programmers can use to serialize and coordinate writers. In this way, the coarse-grained RLU is simpler to use since all operations take an immediate effect, and they execute once and never abort. In contrast, the fine-grained flavor has no support for writer locks. Instead it uses per-object locks of RLU to coordinate writers and does provide support for RLU deferral. As a result, in fine-grained RLU, writers can execute concurrently while avoiding RLU synchronize calls.

Our current RLU implementation consists of approximately 1,000 lines of C code and is available as open source. Note that it does not support the same set of features as RCU, which is a more mature library. In particular callbacks, which allow programmers to register a function called once the grace period is over, are currently not supported. RCU also provides several implementations with different synchronization primitives and various optimizations.

This lack of customization may limit the ability to readily replace RCU by RLU in specific contexts such as in the kernel. For instance, RCU is closely coupled with the operating system scheduler to detect the end of a grace period based on context switches.

The current version of RLU can be used in the kernel but it requires special care while interacting with signals, synchronization primitives, or thread-specific features, in the same way as RCU. For example, RCU can suffer from deadlocks due to interaction of synchronize_rcu() and RCU read-side with mutexes. Furthermore, one should point out that approximately one third of RCU calls in the Linux kernel are performed using the RCU list API, which is supported on top of RLU, hence enabling seamless use of RLU in the kernel.

Finally, we note that a recently proposed passive locking scheme [23] can be used to eliminate memory barriers from RLU section start calls. Our preliminary results of RLU with this scheme show that it is beneficial for benchmarks that are read-dominated and have short and fast operations. Therefore, we plan to incorporate this feature in the next version of RLU.

4. Evaluation

In this section, we first evaluate the basic RLU algorithm on a set of user-space micro-benchmarks: a linked list, a hash table, and an RCU-based resizable hash table [35]. We compare an implementation using the basic RLU scheme, with the state-of-the-art concurrent designs of those data-structures based on the user-space RCU library [6] with the latest performance fix [2]. We use RLU locks to provide concurrency among write operations, yielding code that is as simple as sequential code; RCU can achieve this only by using a writer lock that serializes all writers and introduces severe overheads. We also study the costs of RLU object duplication and synchronize calls in pathological scenarios.

We then apply the more advanced RLU scheme with the deferral mechanism of synchronize calls to the state-of-the-art RCU-based Citrus tree [2], an enhancement of the Bonsai tree of Clements et al. [5]. The Citrus tree uses both RCU and fine-grained locks to deliver the best performing search tree to date [2, 5]. We show that the code of a Citrus tree based on RLU is significantly simpler and provides better scalability than the original version based on RCU and locks.

Next, we show an evaluation of RLU in the Linux kernel. We compare RLU to the kernel RCU implementation on a classic doubly linked list implementation, the most popular use of RCU in the kernel, as well as a single-linked list and a hash table. We show that RLU matches the performance of RCU while always being safe, that is, eliminating the restrictions on use imposed in the RCU implementation (in the kernel RCU doubly linked list, traversing the list forward and backwards is done in unsafe mode since it can lead to inconsistencies [26]). We also evaluate correctness of RLU using a subset of the kernel-based RCU torture test module.

Finally, to show the expressibility of RLU beyond RCU, we convert a real-world application, the popular in-memory Kyoto Cabinet Cache DB, to use RLU instead of using a single global reader-writer lock for thread coordination. We show that the new RLU-based code has almost linear scalability. It is unclear how one could convert Kyoto Cabinet,

```
int rlu_list_add(rlu_thread_data_t *self,
                 list_t *list, val_t val) {
  int result;
  node_t *prev, *next, *node;
  val_t v;

restart:
  rlu_reader_lock();
  prev = rlu_dereference(list->head);
  next = rlu_dereference(prev->next);
  while (next->val < val) {
    prev = next;
    next = rlu_dereference(prev->next);
  }
  result = (next->val != val);
  if (result) {
    if (!rlu_try_lock(self, &prev) ||
        !rlu_try_lock(self, &next)) {
      rlu_abort(self);
      goto restart;
    }
    node = rlu_new_node();
    node->val = val;
    rlu_assign_ptr(&(node->next), next);
    rlu_assign_ptr(&(prev->next), node);
  }
  rlu_reader_unlock();

  return result;
}
```

Listing 2. RLU list: add function.

which requires simultaneous manipulation of several concurrent data structures, to use RCU.

4.1 Linked List

In our first benchmark, we apply RLU to the linked list data-structure. We compare our RLU implementation to the state-of-the-art concurrent non-blocking Harris-Michael linked list (designed by Harris and improved by Michael) [16, 31]. The code for the Harris-Michael list is from synchrobench [14] and, since this implementation leaks memory (it has no support for memory reclamation), we generate an additional more practical version of the list that uses hazard pointers [32] to detect stale pointers to deleted nodes. We also compare our RLU list to an RCU implementation based on the user-space RCU library [2, 6]. In the RCU list, the simple implementation serializes RCU writers. Note that it may seem that one can combine RCU with fine-grained per node locks and make RCU writers concurrent. However, this is not the case, since nodes may change after the locking is complete. As a result, it requires special post-lock validations, ABA checks, and more, which complicates the solution and makes it similar to Harris-Michael list. We show that by using RLU locks, one can provide concurrency among RLU writers and maintain the same simplicity as that of RCU code with serial writers. Our evaluation is performed on a 16-way Intel Core i7-5960X 3GHz 8-core chip with two hardware hyperthreads per core, on Linux 3.13 x86_64 with GCC 4.8.2 C/C++ compiler.

In Figure 4 one can see throughput results for various linked lists with various mutation ratios. Specifically, the figure presents 2%, 20%, and 40% mutation ratios for each algorithm (insert:remove ratio is always 1:1):

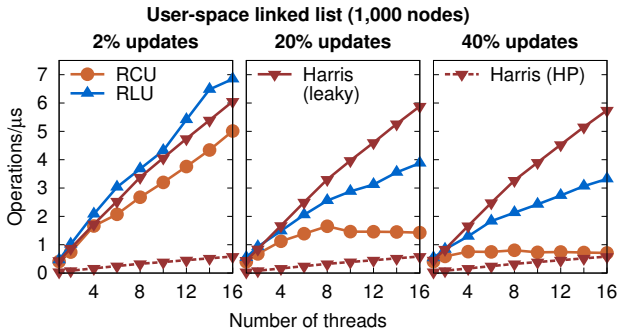

Figure 4. Throughput for linked lists with 2% (left), 20% (middle), and 40% (right) updates.

1. **Harris leaky:** The original list of Harris-Michael that leaks memory.
2. **Harris HP:** The more practical list of Harris-Michael with a fixed memory leak via the use of hazard pointers.
3. **RCU:** The RCU-based list that uses the user-space RCU library and executes serial writers.
4. **RLU:** The basic RLU, as described in Section 3, that uses RLU locks to concurrently execute RLU writers.
5. **RLU defer:** The more advanced RLU that defers RLU synchronize calls to the actual data conflicts between threads. The maximum defer limit is set to 10 write-sets.

In Figure 4, as expected the leaky Harris-Michael list provides the best overall performance across all concurrency levels. The more realistic HP Harris-Michael list with the leak fixed is much slower due to the overhead of hazard pointers that execute a memory fence on each object dereference. Next, the RCU-based list with writers executing serially has a significant overhead due to writer serialization. This is the cost RCU must pay to achieve a simple implementation, whereas by using RLU we achieve the same simplicity but a better concurrency that allows RLU to perform much better than RCU. Listing 2 shows the actual code for the list `add()` function that uses RLU. One can see that the implementation is simple: by using RLU to lock nodes, there is no need to program custom post-lock validations, ABA identifications, mark bits, tags and more, as is usually done for standard fine-grained locking schemes [2, 18, 21] (consider also the related example in Listing 3). Finally, in this execution, the difference between RLU and deferred RLU is not significant, so we plot only RLU.

4.2 Hash Table

Next, we construct a simple hash table data-structure that uses one linked list per bucket. For each key, it first hashes the key into a bucket, and then traverses the associated linked list using the specific implementations discussed above.

Figure 5 shows the results for various hash tables and mutation ratios. Here we base the RCU hash table implementation on per-bucket locks, so RCU writers that access differ-

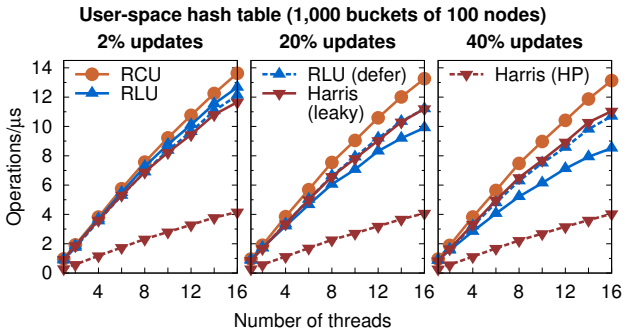

Figure 5. Throughput for hash tables with 2% (left), 20% (middle), and 40% (right) updates.

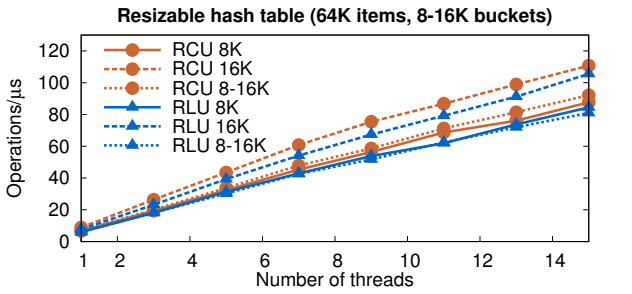

Figure 6. Throughput for the resizable hash table.

ent buckets can execute concurrently. As a result, RCU is highly effective and even outperforms (by 15%) the highly concurrent hash table design that uses Harris-Michael lists as buckets. The reason for this is simply the fact that RCU readers do less constant work than the readers of Harris-Michael (that execute mark bits checks and more). In addition, in this benchmark we show deferred RLU since it has a more significant effect here than in the linked-list. This is because the probability of getting an actual data conflict in a hash table is significantly lower than getting a conflict in a list, so the deferred RLU reduces the amount of synchronize calls by an order of magnitude as compared to the basic RLU. As one can see, the basic RLU incurs visible penalties for increasing mutation ratios. This is a result of RLU synchronize calls that have more effect when operations are short and fast. However, the deferred RLU eliminates these penalties and matches the performance of Harris-Michael. Note that hazard pointers are less expensive in this case, since operations are shorter and are more prone to generate a cache miss due to sparse memory accesses of hashing.

4.3 Resizable Hash Table

To further evaluate RLU on highly-efficient data structures, we implement an RCU-based resizable hash table of Triplett, McKenney, and Walpole [35]. In RCU, the table expand process first creates a new array of buckets that is linked to the old buckets. As a result, the new buckets are "zipped" and, in the next stage, the algorithm uses a "column-wise" iterative RCU process to unzip the buckets: it unzips each pair of buckets one step and, before moving to the next step, it executes the RCU synchronize call. The main reason for this column-wise design is the single pointer update limitation of RCU. Notice that this design exposes intermediate "unzip point" nodes to concurrent inserts and removes, which significantly complicates these operations in RCU.

We convert the RCU table to RLU that uses per bucket writer locks, and eliminate the column-wise design: each pair of buckets is fully unzipped in "one shot" unzip operation. As a result, in RLU, there is no need to handle intermediate "unzip point" nodes during inserts or removes, so they can can execute concurrently without any programming effort.

The authors of RCU-based resizable hash table provide source code that has no support for concurrent inserts or removes (only lookups). As a result, we use the same benchmark as in their original paper [35]: a table of 2^{16} items that constantly expands and shrinks between 2^{13} and 2^{14} buckets (resizing is done by a dedicated thread), while there are concurrent lookups.

Figure 6 presents results for RCU and RLU resizable hash tables. For both, it shows the 8K graph, which is the 2^{13} buckets table without resizes, the 16K graph, which is the 2^{14} table without resizes, and the 8K-16K, which is the table that constantly resizes between 2^{13} and 2^{14} buckets. As can be seen in the graphs, RLU provides throughput that is similar to RCU.

We also compared the total number of resizes and saw that the RLU resize is twice slower than the RCU resize due to the overheads of duplicating nodes in RLU. However, resizes are usually infrequent, so we would expect the latency of a resize to be less critical than the latency that it introduces into concurrent lookups.

4.4 Update-only Stress Test

In order to evaluate the main overheads of RLU compared to RCU, we execute an additional micro-benchmark that reproduces pathological cases that stress RLU object duplication and synchronize calls. The benchmark executes 100% updates on a 10,000 bucket hash table that has only one item in each bucket. As a result, RCU-based updates are quick: they simply hash into the bucket and update the single item of this bucket, whereas the RLU-based updates must also duplicate the single item of the bucket and then execute the RLU synchronize call.

Figure 7 presents results for this stress test. As can be seen, RLU is 2-5 times slower than RCU. Notice that, for a single-thread, RLU is already twice slower than RCU, which is a result of RLU object duplications (synchronize has no penalty for a single thread). Then, with increased concurrency, the RLU penalty increases due to RLU synchronize calls. However, by using RLU deferral, this penalty decreases to the level of the single-thread execution. This

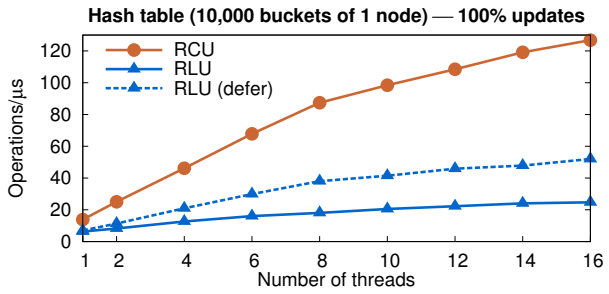

Figure 7. Throughput for the stress test on a hash table with 100% updates and a single item per bucket.

means that RLU deferral is effective in eliminating the penalty of RLU synchronize calls.

4.5 Citrus Search Tree

A recent paper by Arbel and Attiya [2] presents a new design of the Bonsai search tree of Clements et al. [5], which was initially proposed to provide a scalable implementation for address spaces in the kernel. The new design, called the *Citrus tree*, combines RCU and fine-grained locks to support concurrent write operations that traverse the search tree by using RCU protected sections. The results of this work are encouraging, and they show that scalable concurrency is possible using RCU.

The design of Citrus is however quite complex and it requires careful understanding of concurrency and rigorous proof procedures. Specifically, a write operation in Citrus first traverses the tree by using RCU protected read-side section, and then uses fine-grained locks to lock the target node (and possibly successor and parent nodes). Then, after node locking succeeds, it executes post-lock validations, makes node duplications, performs an RCU synchronize call, and manipulates object pointers. As a result, the first phase that traverses the tree is simple and efficient, while the second phase of locking, validation, and modification is manual, complex, and error-prone.

We use RLU to reimplement the Citrus tree, and our results show that the new code is much simpler: RLU completely automates the complex locking, validation, duplication, and pointer manipulation steps of the Citrus writer, which a programmer would have previously needed to manually design, code, and verify. Listings 3 and 4 present the code of Citrus `delete()` function for RCU and RLU (for clarity, some details are omitted). Notice, that the RCU implementation is based on mark bits, tags (to avoid ABA), post-lock custom validations, and manual RCU-style node duplication and installation. In contrast, the RLU implementation is straightforward: it just locks each node before writing to it, and then performs "sequential" reads and writes.

Figure 8 presents performance results for RCU and RLU Citrus trees. In this benchmark, we execute on an 80-way highly concurrent 4 socket Intel machine, in which each socket is an Intel Xeon E7-4870 2.4GHz 10-core chip with

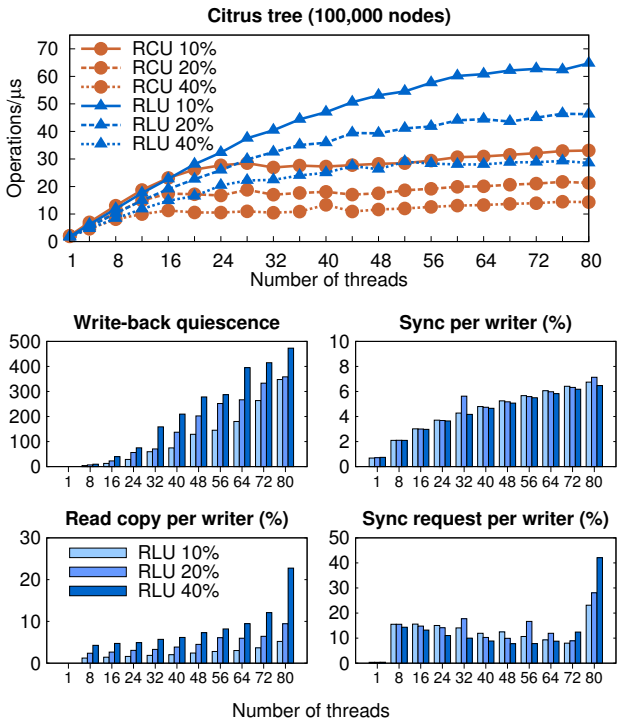

Figure 8. Throughput for the Citrus tree with RCU and RLU (top) and RLU statistics (bottom).

two hyperthreads per core. In addition, we apply the deferred RLU algorithm to reduce the synchronization calls of RLU and provide better scalability. We show results for 10%, 20%, and 40% mutation ratios for both RCU and RLU, and also provide some RLU statistics:

1. *RLU write-back quiescence:* The average number of iterations spent in the RLU synchronize waiting loop. This provides a rough estimate for the cost of RLU synchronize for each number of threads (each iteration includes one `cpu_relax()` call to reduce bus noise and contention).
2. *RLU sync ratio:* The probability for a write operation to execute the RLU synchronize call, write-back, and unlock. In other words, the sync ratio indicates the probability for an actual data conflict between threads, where a thread sends a sync request that forces another thread to synchronize and "flush" the new data to the memory.
3. *RLU read copy ratio:* The probability for an object read to steal a new copy from a write-log of another thread. This provides an approximate indication for how many read-write conflicts occur during benchmark execution.
4. *RLU sync request ratio:* The probability for a thread to find a node locked by other thread. Notice, that this number is higher than the actual RLU sync ratio, since multiple threads may find the same locked object and send multiple requests to the same thread to unlock the same object.

```
bool RCU_Citrus_delete(citrus_t *tree, int key) {
  node_t *pred, *curr, *succ, *parent, *next, *node;

  urcu_read_lock();
  pred = tree->root;
  curr = pred->child[0];

  ...Traverse the tree...

  urcu_read_unlock();
  pthread_mutex_lock(&(pred->lock));
  pthread_mutex_lock(&(curr->lock));

  // Check that pred and curr are still there
  if (!validate(pred, 0, curr, direction)) {
    ...Restart operation...
  }

  ...Handle case with 1 child, assume 2 children now...

  // Find successor and its parent
  parent = curr;
  succ = curr->child[1];
  next = succ->child[0];
  while (next != NULL) {
    parent = succ;
    succ = next;
    next = next->child[0];
  }
  pthread_mutex_lock(&(succ->lock));

  // Check that succ and its parent are still there
  if (validate(parent, 0, succ, succDirection) &&
      validate(succ, succ->tag[0], NULL, 0)) {
    curr->marked = true;

    // Create a new successor copy
    node = new_node(succ->key);
    node->child[0] = curr->child[0];
    node->child[1] = curr->child[1];
    pthread_mutex_lock(&(node->lock));

    // Install the new successor
    pred->child[direction] = node;

    // Ensures no reader is accessing the old successor
    urcu_synchronize();

    // Update tags/marks and redirect the old successor
    if (pred->child[direction] == NULL)
      pred->tag[direction]++;
    succ->marked = true;
    if (parent == curr) {
      node->child[1] = succ->child[1];
      if (node->child[1] == NULL)
        node->tag[1]++;
    }
  } else {
    parent->child[0] = succ->child[1];
    if (parent->child[1] == NULL)
      parent->tag[1]++;
  }

  ...Unlock all nodes...

  // Deallocate the removed node
  free(curr);

  return true;
}
```

Listing 3. RCU-based Citrus delete operation [2].

Performance results show that RLU Citrus matches RCU for low thread counts, and improves over RCU for high thread counts by a factor of 2. This improvement is due to the deferral process of synchronize calls, which allows RLU to execute expensive synchronize calls only on actual data conflicts, whereas the original Citrus must execute synchronize for each delete. As can be seen in RLU statistics, the reduction is almost an order of magnitude (10% sync and write-back ratio) relative to the basic RLU. It is important to note that Arbel and Morrison [3] proposed an RCU predicate primitive that allows them to reduce the cost of synchronize calls. However, defining an RCU predicate requires explicit programming and internal knowledge of the data-structure, in contrast to RLU that automates this process.

```
bool RLU_Citrus_delete(citrus_t *tree, int key) {
  node_t *pred, *curr, *succ, *parent, *next, *node;

  rlu_reader_lock();
  pred = (node_t *)rlu_dereference(tree->root);
  curr = (node_t *)rlu_dereference(pred->child[0]);

  ...Traverse the tree, assume 2 children now...

  // Find successor and its parent
  parent = curr;
  succ = (node_t *)rlu_dereference(curr->child[1]);
  next = (node_t *)rlu_dereference(succ->child[0]);
  while (next != NULL) {
    parent = succ;
    succ = next;
    next = (node_t *)rlu_dereference(next->child[0]);
  }

  // Lock nodes and manipulate pointers as in serial code
  if (parent == curr) {
    rlu_lock(&succ);
    rlu_assign_ptr(&(succ->child[0]), curr->child[0]);
  } else {
    rlu_lock(&parent);
    rlu_assign_ptr(&(parent->child[0]), succ->child[1]);
    rlu_lock(&succ);
    rlu_assign_ptr(&(succ->child[0]), curr->child[0]);
    rlu_assign_ptr(&(succ->child[1]), curr->child[1]);
  }
  rlu_lock(&pred);
  rlu_assign_ptr(&(pred->child[direction]), succ);

  // Deallocate the removed node
  rlu_free(curr);

  // Done with delete
  rlu_reader_unlock();

  return true;
}
```

Listing 4. RLU-based Citrus delete operation.

4.6 Kernel-space Tests

The kernel implementation of RCU differs from user-space RCU in a few key aspects. It notably leverages kernel features to guarantee non-preemption and scheduling of a task after a grace period. This makes RCU extremely efficient and have very low overhead. The rlu_reader_lock() can be as short as a compiler memory barrier with non-preemptible RCU. Thus, to compare the performance of the kernel implementation of RCU with RLU, we create a Linux kernel module along the same line as Triplett et al. with the RCU hash table [35].

One main use case of kernel RCU are for linked lists that are used throughout Linux, from the kernel to the drivers. We therefore first compare the kernel RCU implementation of this structure to our RLU version.

We implemented our RLU list by leveraging the same API as the RCU list (list_for_each_entry_rcu(), list_add_rcu(), list_del_rcu()) and replacing the RCU API with RLU calls. For benchmarking we inserted dummy nodes with ap-

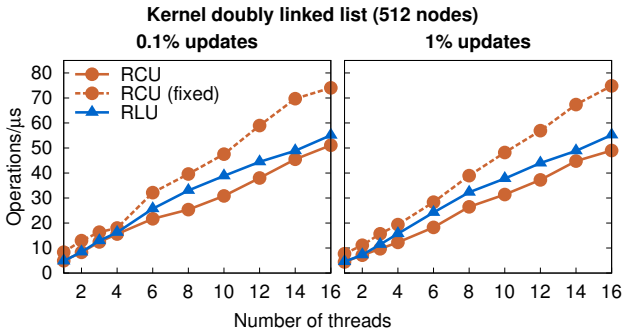

Figure 9. Throughput for kernel doubly linked lists (`list_*` APIs) with 0.1% (left) and 1% (right) updates.

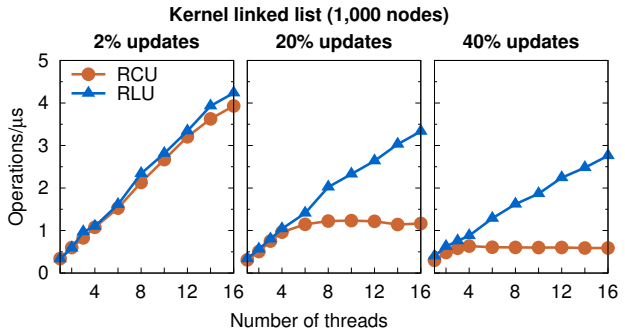

Figure 10. Throughput for linked lists running in the kernel with 2% (left), 20% (middle), and 40% (right) updates.

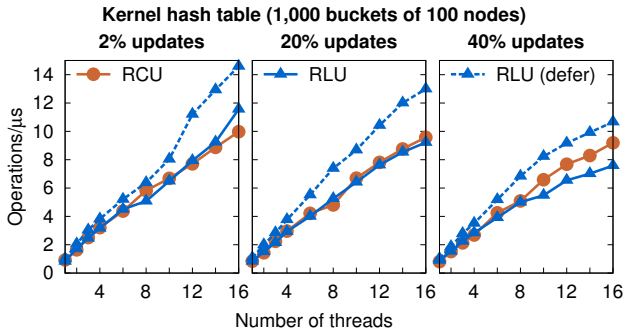

Figure 11. Throughput for hash tables running in the kernel with 2% (left), 20% (middle), and 40% (right) updates.

propriate padding to fit an entire cache line. We used the same test machine as for the user-space experiment (16-way Intel i7-5960X) with version 3.16 of the Linux kernel and non-preemptible RCU enabled, and we experimented with low update rates of 0.1% and 1% updates that represent the common case for using RCU-based synchronization in the kernel. We implemented a performance fix in the RCU list implementation (in list_entry_rcu()), which we have reported to the Linux kernel mailing list. Results with the fix are labeled as "RCU (fixed)" in the graphs.

We observe in Figure 9 that RCU has reduced overhead compared to RLU in read-mostly scenarios. However, the semantics provided by the two lists is different. RCU cannot add an element atomically in the doubly-linked list and it therefore by restricts all concurrent readers to only traverse forward. Traversing the list backwards is unsafe since it can lead to inconsistencies, so special care must be taken to avoid memory corruptions and system crash [26]. In contrast, RLU provides a consistent list at a reasonable cost.

We also conducted kernel tests with higher update rates of 2%, 20% and 40% on a single-linked list and a hash-table to match the userspace benchmarks and compare RLU against the kernel implementation of RCU. Note that these data structure are identical to those tested earlier in user space, but they use the kernel implementation of RCU instead of the user-space RCU library. Results are shown in Figure 10 and Figure 11. As expected, in the linked-list, increasing writers in RCU introduces a sequential bottleneck, while RLU writers proceed concurrently and allow RLU to scale. In the hash-table, RCU scales since it uses per bucket locks and RLU matches RCU. Note that the deferred RLU slightly outperforms RCU, which due to faster memory deallocations (and reuse) in RLU compared to RCU (that must wait for the kernel threads to context-switch).

4.7 Kernel-space Torture Tests

The kernel implementation of RCU comes with a module, named RCU torture, which tests the RCU implementation for possible bugs and correctness problems. It contains many tests that exercise the different implementations and operating modes of RCU. As RLU does not support all the features and variants of RCU, we only considered the basic RCU torture tests that check for consistency of the classic implementation of RCU and can be readily applied to RLU.

These basic consistency tests consist of 1 writer thread, n reader threads, and n fake writer threads. The writer thread gets an element from a private pool, shares it with other threads using a shared variable, then takes it back to the private pool using RCU mechanism (deferred free, synchronize, etc.). The reader threads continuously read the shared variable while the fake writers just invoke synchronize with random delays. All the threads perform consistency checks at different steps and with different delays.

We have successfully run this RLU torture test with deferred free and up to 15 readers and fake writers on our 16-way Intel i7-5960X machine. While our experiments only cover a subset of all the RCU torture tests, it still provides strong evidence of the correctness of our algorithm and its implementation. We plan to expand the list of tests as we add additional features and APIs to RLU.

4.8 Kyoto Cabinet Cache DB

We finally illustrate how to use RLU for an existing application with the popular in-memory database implementation Kyoto Cabinet Cache DB [12]. Kyoto Cache DB is written in C++ and its DBM implementation is relatively simple

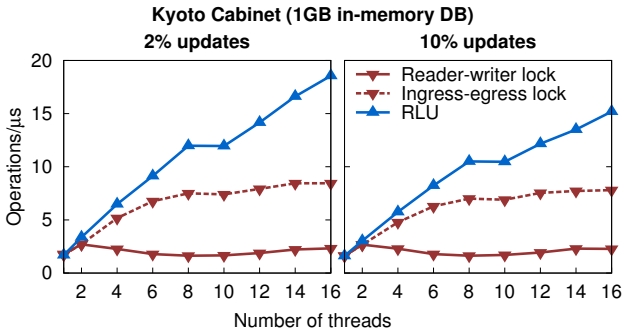

Figure 12. Throughput for the original and RLU versions of the Kyoto Cache DB.

and straightforward. Internally, Kyoto breaks the database into slots, where each slot is composed of buckets and each bucket is a search tree. As a result, to find a key, Kyoto first hashes the key into a slot, and then into a bucket in this slot. Then, it traverses the search tree that resides in the bucket and processes the record that includes the key and returns.

Database operations in Kyoto CacheDB are fast due to the double hashing procedures and use of search trees. However, Kyoto fails to scale with increasing numbers of threads, and in fact it usually collapses after 3-4 threads. Recent work by Dice et al. [9] observed a scalability bottleneck and indicated that the problem is the global reader-writer lock that Kyoto uses to synchronize database operations.

We conducted a performance analysis of Kyoto Cache DB and concur with [9] that the global reader-writer lock is indeed the problem. However, we also found that Kyoto performs an excessive amount of thread context switches due to the specific implementation of reader-writer spin locks in the Linux pthreads library. We therefore decided to first eliminate the context switches by replacing the reader-writer lock of Kyoto with an ingress-egress reader-writer lock implementation [7]. To the best of our knowledge, the ingress-egress reader-writer locks perform the best on Intel machines (ingress/enter counter and egress/exit counter for read-lock/read-unlock) [1]. We note that one could use hierarchical cohort-based reader-writer locks [10] in our benchmark to reduce the cache traffic in Kyoto, but this would not have a significant effect since the performance analysis reveals that the cache miss ratio is already low (4%-5%).

In addition to the global reader-writer lock, Kyoto also uses a lock per slot. As a result, each operation acquires the global reader-writer lock for a read or a write, depending on whether the actual operation is read-only or not, and then acquires the lock of the relevant slot. Based on this, we apply the RLU scheme to Kyoto Cache DB in a way that eliminates the need for the global reader-writer lock, and use the per slot locks to synchronize the RLU writers. A good benefit of this design is the fact that RLU writers are irrevocable and have no need to support abort or undo procedures. As a result, the conversion to RLU is simple and straightforward.

Figure 12 shows throughput results for the original, fixed (ingress-egress reader-writer lock), and RLU-based Kyoto Cache DB for 2% and 10% mutation ratios and 1GB DB. This benchmark runs on a 16-way Intel 8-core chip, where each thread randomly executes set(), add(), remove(), and get() DB operations.

In the performance graph one can see that the new RLU based Kyoto provides continued scalability where the original Kyoto fails to scale due to the global reader-writer lock (the slight drop of RLU from 8 to 10 threads is due to 8-core hyper-threading). Observe that the original Kyoto implementation fails to scale despite the fact that the amount of read-only operations is high, about 90-98%. Fixing this problem by replacing the global reader-writer lock with an ingress-egress lock eliminates the excess context switching and allows Kyoto to scale until 6-8 threads. Note that it is possible to combine the ingress-egress lock with a passive locking scheme [23] to avoid memory barriers on the read-side of the lock, but writers still cannot execute concurrently with readers and this approach introduces a sequential bottleneck and limits scalability.

We believe that if one would convert Kyoto to RCU by using the per slot locks for synchronization of writers (like we did), it would provide the same performance as with RLU. However, it is not clear how to even begin to convert those update operations to use RCU. Kyoto's update operation may modify multiple nodes in a search tree, multiple locations in the hash tables, and maybe some more locations in other helper data-structures. The result, we fear, will be a non-trivial design, which in the end will deliver performance similar to the one RLU provides quite readily today.

5. Conclusion

In summary, one can see that the increased parallelism hidden under the hood of the RLU system provides for a simple programming methodology that delivers performance similar or better than that obtainable with RCU, but at a significantly lower intellectual cost to the programmer. RLU is compatible with the RCU interface, and we hope that its combination of good performance and more expressive semantics will convince both kernel and user-space programmers to use it to parallelize real-world applications.

Acknowledgments

We would like to thank the anynymous reviewers for their constructive comments, as well as Haibo Chen for his help in preparing the final version of this paper. Support is gratefully acknowledged from the National Science Foundation under grants CCF-1217921, CCF-1301926, and IIS-1447786, the Department of Energy under grant ER26116/DE-SC0008923, the European Union under COST Action IC1001 (Euro-TM), and the Intel and Oracle corporations.

References

[1] Y. Afek, A. Matveev, and N. Shavit. Pessimistic software lock-elision. In *Proceedings of the 26th International Conference on Distributed Computing*, DISC'12, pages 297–311, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33650-8.

[2] M. Arbel and H. Attiya. Concurrent updates with RCU: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2944-6.

[3] M. Arbel and A. Morrison. Predicate RCU: An RCU for scalable concurrent updates. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 21–30, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7.

[4] S. Boyd-Wickizer. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels*. PhD thesis, Massachusetts Institute of Technology, 2013.

[5] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8.

[6] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012. ISSN 1045-9219.

[7] D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 284–293, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0079-7. . URL http://doi.acm.org/10.1145/1810479.1810531.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-44624-9, 978-3-540-44624-8.

[9] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 188–197, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2821-0.

[10] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: A general technique for designing NUMA locks. *ACM Trans. Parallel Comput.*, 1(2):13:1–13:42, Feb. 2015. ISSN 2329-4949.

[11] R. Ennals. Software transactional memory should not be obstruction-free. Technical report, Intel Research Cambridge, 2006. IRC-TR-06-052.

[12] FAL Labs. Kyoto cabinet: A straightforward implementation of DBM, 2011. URL http://fallabs.com/kyotocabinet/.

[13] A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 249–269, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37035-9. . URL http://dx.doi.org/10.1007/978-3-642-37036-6_15.

[14] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7.

[15] 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):221–236, 2008.

[16] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[17] T. E. Hart. Comparative performance of memory reclamation strategies for lock-free and concurrently-readable data structures. Master's thesis, University of Toronto, 2005.

[18] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, OPODIS'05, pages 3–16, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-36321-1, 978-3-540-36321-7. . URL http://dx.doi.org/10.1007/11795490_3.

[19] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0079-7. . URL http://doi.acm.org/10.1145/1810479.1810540.

[20] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, San Mateo, CA, 2008. ISBN 0-12-370591-6.

[21] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity*, SIROCCO'07, pages 124–138, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72918-1. URL http://dl.acm.org/citation.cfm?id=1760631.1760646.

[22] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 82–91, New York, NY, USA, 2006. ACM. ISBN 1-59593-578-9. . URL http://doi.acm.org/10.1145/1178597.1178611.

[23] R. Liu, H. Zhang, and H. Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 219–230, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-

10-2. URL http://dl.acm.org/citation.cfm?id=2643634.2643658.

[24] A. Matveev and N. Shavit. Reduced hardware NOrec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 59–71, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7.

[25] P. E. McKenney. Sleepable RCU, 2006. URL http://lwn.net/Articles/202847/.

[26] P. E. McKenney. What is RCU, fundamentally?, 2007. URL https://lwn.net/Articles/262464/.

[27] P. E. McKenney. Hierarchical RCU, 2008. URL http://lwn.net/Articles/305782/.

[28] P. E. McKenney and J. D. Slingwine. Read-Copy-Update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, Oct. 1998.

[29] P. E. McKenney and J. Walpole. Introducing technology into the linux kernel: A case study. *SIGOPS Oper. Syst. Rev.*, 42 (5):4–17, July 2008. ISSN 0163-5980.

[30] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole. RCU usage in the linux kernel: One decade later. Technical report, 2013.

[31] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM. ISBN 1-58113-529-7. . URL http://doi.acm.org/10.1145/564870.564881.

[32] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15 (6):491–504, June 2004. ISSN 1045-9219.

[33] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 284–298, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-44624-9, 978-3-540-44624-8.

[34] M. L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, San Mateo, CA, 2013. ISBN 1-60-845956-X.

[35] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2002181.2002192.

[36] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9.