A Critical RCU Safety Property is... Ease of Use!

Paul E. McKenney
IBM Linux Technology Center
Hillsboro, OR, USA
paulmckrcu@gmail.com

ABSTRACT

Some might argue that read-copy update (RCU) is too low-level to be targeted by hackers, but the advent of Row Hammer [19] demonstrated the naïveté of such views. After all, if black-hat hackers are ready, willing, and able to exploit hardware bugs such as Row Hammer, they are assuredly ready, willing, and able to exploit bugs in RCU. Nor is it any longer the case that RCU's involvement in exploitable Linux-kernel bugs is strictly theoretical. However, this bug involved not RCU's correctness, but rather its ease of use. Nevertheless, it was a real bug that really needed fixing. This paper describes this bug and the road to its eventual fix.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; • **Software and its engineering** → **Multithreading**; • **General and reference** → *Validation*;

KEYWORDS

Operating systems, Security: Exploitable bugs, Concurrency control, Read-copy update

ACM Reference Format:
Paul E. McKenney. 2019. A Critical RCU Safety Property is... Ease of Use!. In *Proceedings of ACM International Systems and Storage Conference (SYSTOR'19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3319647.3325836

1 INTRODUCTION

RCU is a technique that often replaces reader-writer locking, due to RCU's read-side primitives being both wait-free and an order of magnitude faster than uncontended locking. However, it turns out that even the combination of wait-freedom, extreme scalability, and excellent performance do not constitute an excuse to completely ignore ease-of-use issues. This paper presents a brief overview of RCU, the ease-of-use issue in RCU that led to the exploit, a couple of non-fixes, the ultimate fix (thus far, anyway), further consequences, and lessons (re)learned.

SYSTOR'19, June 2019, Haifa, Israel

ACM ISBN 978-1-4503-6749-3…$15.00
https://doi.org/10.1145/3319647.3325836

2 A BRIEF OVERVIEW OF RCU

Although RCU has enjoyed heavy use by the Linux kernel and a number of other projects for quite some time, detailed awareness of its uses and implementation is not yet mainstream. Therefore, Section 2.1 summarizes RCU's conceptual properties and cites formalizations, Section 2.2 presents a trivial use case, Section 2.3 presents an equally trivial implementation, and finally Section 2.4 summarizes the three flavors of Linux-kernel RCU. Readers who have used or implemented RCU may wish to skip ahead to Section 2.4 on page 5, and readers fluent in Linux-kernel RCU may wish to skip further ahead to Section 3, also on page 5.

2.1 Relevant RCU Properties

RCU [27, 33, 36] permits updaters to wait for pre-existing *RCU read-side critical sections*, and is intended for situations in which such critical sections are executed much more frequently than the corresponding update-side waits. Each RCU read-side critical section begins with `rcu_read_lock()` and ends with `rcu_read_unlock()`, and any region of code not in a critical section represents a *quiescent state*. Nested critical sections are flattened, so that a nested set of critical sections acts as one large critical section.

RCU updaters are typically split into phases. The first phase changes reader-visible state, for example, removing a data element. The second phase waits for pre-existing readers, relying on the fact that new readers cannot see the old state, for example, being unable to gain a reference to a just-removed element. The third and final phase carries out reader-unsafe actions such as freeing a just-removed element. These actions are now safe because all remaining readers are aware of the new reader-visible state, for example, being guaranteed not to hold a reference to the just-removed element. The wait phase invokes `synchronize_rcu()`, which waits for a *grace period* that starts at the time of invocation

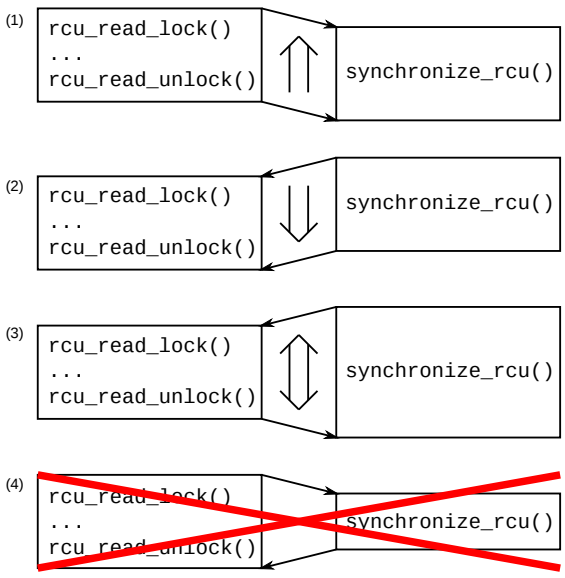

Figure 1: RCU Execution Constraints

and ends after all pre-existing readers have completed. During a grace period, all CPUs and tasks must visit a quiescent state at least once, hence any CPU or task failing to ever visit a quiescent state will prevent any future `synchronize_rcu()` invocations from returning. Any period of time containing a grace period is itself a grace period.

RCU differs from reader-writer locking in that `synchronize_rcu()` need only wait for pre-existing RCU read-side critical sections; it need neither exclude nor wait for critical sections that start execution after the call to `synchronize_rcu()`. This relationship between RCU readers and updaters is illustrated by the four scenarios shown in Figure 1, the first three of which are permitted and the last of which is forbidden. Each box on the left-hand side of this figure represents an RCU read-side critical section, and each box on the right-hand side of this figure represents a grace period, that is, the call to `synchronize_rcu()` happens at the top of the box and the return at the bottom, so that within each scenario, time progresses from top to bottom. The first scenario shows that if a critical section ends before the end of a given grace period, that critical section may start before that grace period starts. The second scenario shows that if a critical section starts after the beginning of a given grace period, that critical section may end after that grace period ends. The third scenario shows that a grace period may completely overlap a given critical section. The fourth and final scenario shows that a critical section absolutely must not complete overlap any grace period. Taken together, these scenarios mean that although critical sections must affect the execution of grace periods, the reverse need not be the case. This asymmetry permits `rcu_read_lock()` and `rcu_read_unlock()` to be exceedingly lightweight and scalable [4].

Formal semantics for RCU are also available [1, 10, 11].

```
1 struct foo {
2   int data;
3 } *gptr = NULL;
4
5 int lookup(int *dp)
6 {
7   struct foo *p;
8
9   rcu_read_lock();
10  p = rcu_dereference(gptr);
11  if (!p) {
12    rcu_read_unlock();
13    return 0;
14  }
15  *dp = p->data;
16  rcu_read_unlock();
17  return 1;
18 }
19
20 int insert(int newdata)
21 {
22  struct foo *p;
23
24  p = kmalloc(sizeof(*p), GFP_KERNEL);
25  BUG_ON(!p);
26  p->data = newdata;
27  if (cmpxchg(&gptr, NULL, p)) {
28    kfree(p);
29    return 0;
30  }
31  return 1;
32 }
33
34 int remove(void)
35 {
36  struct foo *p;
37
38  p = xchg(&gptr, NULL);
39  synchronize_rcu();
40  kfree(p);
41  return !!p;
42 }
```

Figure 2: RCU Linked Insertion and Removal

2.2 Trivial RCU Use Case

With the addition of `rcu_dereference()`,[1] the RCU primitives described in the preceeding section enable the linked-list use case that represents the most common application of RCU, a trivial instance of which is shown in Figure 2.[2] Lines 1-3 of this figure show a global pointer `gptr` that is initially `NULL`, but that can reference a `struct foo` which contains a single integer `->data`. This trivial data structure has the following three access functions:

- `lookup()` on lines 5-18, which returns 0 if `gptr` is `NULL`, and otherwise returns 1 and stores the current `->data`

[1] Similar to a C11 `memory_order_consume` load, but in the common case compiling to a single load instruction.

[2] Despite being trivial, it is used in production [40].

through the argument dp. Figures 3 and 4 illustrate `lookup()`.

- `insert()` on lines 20-32, which returns 0 if `gptr` is not `NULL`, and otherwise returns 1 and causes `gptr` to reference a `struct foo` containing the specified value. Given multiple concurrent `insert()` invocations, at most one of them will return 1. Figure 3 illustrates `insert()`.
- `remove()` on lines 34-42, which returns 0 if `gptr` is `NULL`, and otherwise returns 1, sets `gptr` to `NULL`, and safely frees the structure previously referenced by `gptr`. Given multiple concurrent `remove()` invocations, at most one of them will return 1. Figure 4 illustrates `remove()`.

The `lookup()` function uses an RCU read-side critical section starting with `rcu_read_lock()` on line 9 and ending with `rcu_read_unlock()` on either line 12 or 16. Line 10 uses `rcu_dereference()` to fetch `gptr`, guaranteeing to do so using a single load instruction and also guaranteeing that any dereferences of the fetched pointer will happen after the load. In theory, this line is equivalent to `p = gptr`, but in practice both the compiler and CPU could produce wildly inappropriate results [30, Sections 4.3.4.1 and 15.4.1]. If line 11 sees that `gptr` was `NULL`, line 12 ends the critical section and line 13 returns a failure indication. Otherwise, line 15 fetches the `->data` field, line 16 ends the critical section, and line 17 returns a success indication. A key point is that all of the `lookup()` function's `gptr`-related accesses are within its RCU read-side critical section.

Given the trivial nature of this linked data structure, the `insert()` function can synchronize updates using a single fully ordered atomic compare-and-swap operation, which in the Linux kernel is `cmpxchg()`. Initially, the state is as shown in step (1) in Figure 3: `gptr` is `NULL`. Line 24 allocates the new structure and line 25 checks it, normally resulting in the state in step (2) in the figure. The color of the new structure is green, indicating that readers cannot yet obtain a reference to it. Line 26 initializes the new structure, resulting in the state shown in step (3) of the figure. Then line 27 uses `cmpxchg()` to atomically check whether `gptr` is `NULL`, and if so update it to instead reference the newly allocated structure. If this `cmpxchg()` fails, line 28 frees the newly allocated structure and line 29 returns a failure indication. Otherwise, the state is as shown in step (4) of the figure, with the new object colored red to indicate that readers can now reference it, and finally line 31 returns a success indication. Either way, the atomic nature of the `cmpxchg()` ensures that at most one of a set of concurrently executing `insert()` invocations can succeed, as required.

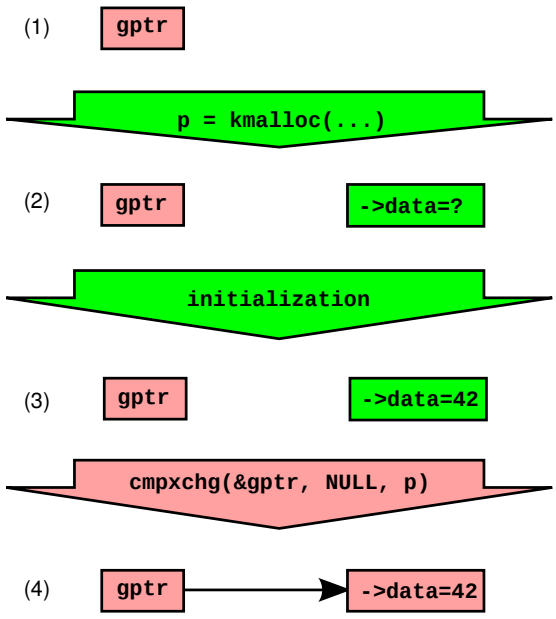

Figure 3: RCU Linked-Structure Insertion

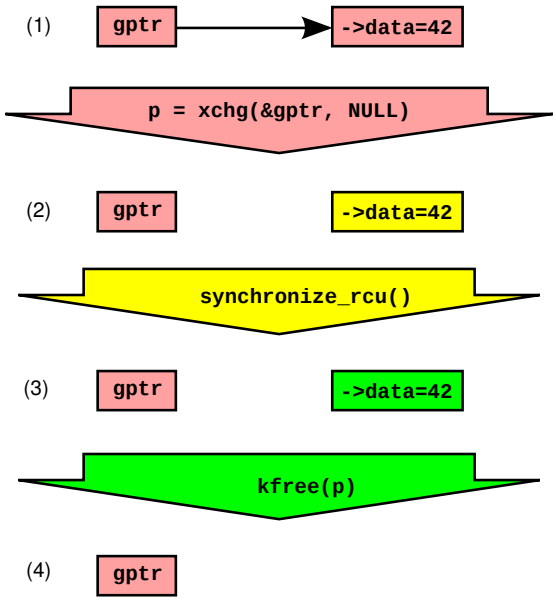

Figure 4: RCU Linked-Structure Removal

The `remove()` function's synchronization is even simpler, using a fully ordered atomic exchange operation [10, Section V.C], which in the Linux kernel is `xchg()`. Initially, the state is as shown in step (1) of Figure 4, where `gptr` references an element with `->data` equal to 42, which is colored

red due to the fact that any number of readers might be referencing it. Line 38 uses `xchg()` to atomically store `NULL` into `gptr` and return the previous value of this pointer, resulting in the state shown step (2) of the figure. At this point, new readers have no way to gain a reference to the old `struct foo` instance, but old readers might still retain references, hence the yellow color.

Note that all potential read-side references to this newly removed instance are held within the RCU read-side critical section spanning lines 9-16 of Figure 2. Furthermore, any such read-side critical section must have executed the `rcu_dereference()` on line 10 before the `xchg()` on line 38 executed and thus before the `synchronize_rcu()` on line 39 executed. Therefore, when `synchronize_rcu()` returns, there can no longer be any instances of `lookup()` retaining references to the just-removed `struct foo` instance. As a result, once execution reaches line 40, the `remove()` function's local variable `p` is the only remaining reference to the newly removed instance of `struct foo`. The state is thus as shown in step (3) of Figure 4, with the newly removed structure colored green to indicate that there can no longer be any readers referencing it. Therefore, line 40 can safely invoke `kfree()`,[3] so that the state is as shown in step (4) of Figure 4. Finally line 41 uses a double-not operation to return zero if `p` was `NULL` and one otherwise. Note that the Linux kernel carefully conceals the nature of its allocation and free functions from the compiler, thus avoiding the consequences of the undefined behavior that results from accessing pointers to already-freed objects [35]. In other environments, it might be wise to compute `!!p` before invoking `kfree()`.

Note that in step (2) of Figure 4, different readers might disagree as to whether or not `gptr` referenced the old `struct foo`. A large number of algorithms have no problem with this, especially those representing real-world state: After all, in-memory state will not immediately reflect a real-world state change. Nevertheless, algorithms requiring global read-side agreement can be accommodated straightforwardly [3].

We have seen a straightforward RCU use case that allows proper synchronization among readers and updaters where the updaters synchronize with each other via atomic read-modify-write operations. More complex linked structures usually require that updaters use higher-level synchronization primitives, with locking being a popular choice. For lock-based updates, `rcu_assign_pointer()` is the update-side counterpart to `rcu_dereference()`, for example, `rcu_assign_pointer(gptr, p)` assigns the value `p` to `gptr` using store-release semantics. However, RCU interoperates numerous update-side synchronization mechanisms, including transactional memory [14, 15, 37, 39, 47], non-blocking synchronization [9, 10, 34], and a single designated updater thread [10, 36].

[3] Note that passing a `NULL` pointer to `kfree()` is a no-op.

```
 1 #define rcu_read_lock()
 2 #define rcu_read_unlock()
 3 #define rcu_dereference(p) READ_ONCE(p)
 4 #define rcu_assign_pointer(p, v) smp_store_release(&(p), (v))
 5 void synchronize_rcu(void)
 6 {
 7   int cpu;
 8
 9   for_each_online_cpu(cpu)
10     while (raw_smp_processor_id() != cpu)
11       sched_setaffinity(current->pid, cpumask_of(cpu));
12 }
```

Figure 5: Trivial RCU Implementation for Non-Preemptive Environments

2.3 Trivial RCU Implementation

Section 2.1 noted that `rcu_read_lock()` and `rcu_read_unlock()` can be exceedingly lightweight and scalable. In fact, in non-preemptible Linux-kernel builds, they emit no code whatsoever. Furthermore, as mentioned in Section 2.2, `rcu_dereference()` can be implemented as a single load instruction. Hence, an RCU reader might traverse a linked data structure using precisely the same sequence of machine instructions used in a single-threaded environment, despite concurrent updates.

These are very attractive properties, but they naturally raise concerns about how this could possibly be implemented. This section provides a trivial implementation for non-preemptive (run-to-block) environments such as Linux kernels built with `CONFIG_PREEMPT=n`. Note that although building with `CONFIG_PREEMPT=n` causes the kernel itself to be non-preemptive, user code running on that kernel can still be preempted any time, anyplace, and anywhere.

The key insight is that non-preemptive environment prohibit blocking while holding a pure spinlock. Failing to observe this prohibition leads to a well-known deadlock scenario. To see this, suppose that the thread holding a given lock is blocked, and all CPUs are spinning on that lock. The thread holding the lock cannot release it until it resumes on some CPU, but the threads currently spinning cannot relinquish their CPUs until they acquire the lock. Therefore, the rule is that blocking is prohibited while holding a spinlock.

This same rule applies to RCU readers in the Linux kernel: Once a `rcu_read_lock()` is encountered, blocking is prohibited until after the matching `rcu_read_unlock()`. Therefore, because blocking results in a context switch, if a CPU executes a context switch, all prior RCU readers running on that CPU are guaranteed to have completed. Once all CPUs have executed a context switch, **all** pre-existing RCU read-side critical sections are guaranteed to have completed.

This means that a slow and fragile but functional implementation of synchronize_rcu() could simply force itself to run on each online CPU, for example, by using the Linux kernel's internal sched_setaffinity() function. This approach results in a synchronize_rcu() implementation that comprises only seven lines of code, as shown by lines 5-12 of Figure 5. Line 9 iterates through the online CPUs, and for each such CPU, line 10 loops over line 11, each execution of which attempts to force the current task to migrate over to that CPU. Therefore, once this function returns, it has forced a pair of context switches on each online CPU, which in turn implies that all RCU read-side critical sections that were running at the time of the call to synchronize_rcu() have completed, as required.

Note that rcu_read_lock() (line 1) and, rcu_read_unlock() (line 2) generate no code.[4] Note further that rcu_dereference() (line 3) generates a C-language volatile load, which will normally compile to a single load instruction. This trivial implementation therefore delivers on the promise of RCU readers using exactly the same sequence of instructions used by their single-threaded counterparts, thus incurring exactly zero synchronization overhead in the absence of updaters. These attractive performance and scalability properties have led to significant RCU use within the Linux kernel, with more than 15,000 calls to its API as of Linux kernel v5.0 [23].

2.4 RCU Flavors

There are three flavors of the RCU API. The original flavor, RCU-sched, relies on disabling read-side preemption and thus on context switch as its primary quiescent state [33]. However, Robert Olsson's network-overload tests resulted in lockups due to the kernel never exiting networking's bottom-half handlers long enough to do a context switch [45]. The solution was the RCU-bh flavor, which has an additional quiescent state upon exit (even momentary exit) from bottom-half handlers [42, 43]. The original RCU-sched flavor was retained because its read-side primitives are significantly faster than those of RCU-bh. However, both RCU-sched's and RCU-bh's readers disable preemption, which proved problematic for deep sub-millisecond workloads [5, 46]. Supporting these workloads required the addition of the third flavor, RCU-preempt, which allows read-side preemption [12]. RCU-sched was retained due to use cases that treat hardware interrupt handlers as readers. In non-realtime and thus non-preemptible kernels, the RCU-preempt flavor maps directly onto the RCU-sched flavor.

[4] This trivial implementation can be made safe for preemptible environments by defining rcu_read_lock() and rcu_read_unlock() as preempt_disable() and preempt_enable(), respectively.

```
1   rcu_read_lock_sched();          q = rcu_dereference(gp);
2   p = rcu_dereference(gp);        rcu_assign_pointer(gp, NULL);
3   do_something_with(p);           synchronize_rcu();
4   rcu_read_unlock_sched();        kfree(q);
```

Figure 6: Exploiting RCU's Ease-of-Use Bug

As of Linux kernel v4.20, RCU-preempt provides rcu_read_lock(), rcu_read_unlock(), rcu_dereference(), and synchronize_rcu(); RCU-bh provides rcu_read_lock_bh(), rcu_read_unlock_bh(), rcu_dereference_bh(), and synchronize_rcu_bh(); and RCU-sched provides rcu_read_lock_sched(), rcu_read_unlock_sched(), rcu_dereference_sched(), and synchronize_sched().[5] These three sets of primitives are similar to a set of three locks in that mixing and matching different flavors of RCU primitives is usually a bug. One exception to this rule is an implementation quirk that has permitted mixing rcu_read_lock_bh() readers with synchronize_sched() updaters. On the other hand, mixing rcu_read_lock_sched() readers with synchronize_rcu_bh() updaters really can fail. RCU's intolerance of such mixing and matching is the root cause of the ease-of-use bug described in the next section.

3 RCU'S EASE-OF-USE BUG

RCU has a long history of ease-of-use facilities, bugs, and fixes. For example, the original 2002 Linux-kernel RCU provided rcu_read_lock() and rcu_read_unlock(), which greatly eases detection of illegal blocking within RCU read-side critical sections. Further, in 2004, open-coded use of smp_read_barrier_depends() and smp_wmb() was replaced by rcu_dereference() and rcu_assign_pointer(), respectively, greatly easing the code reader's task of working out what RCU protects [44]. Finally, starting in 2010, RCU uses lockdep [6] to more easily detect traversal of RCU-protected pointers outside of the required critical sections and/or update-side locks [25].

Returning to the present-day ease-of-use bug, an example of buggy mixing and matching of different RCU flavors is shown in Figure 6, in which the read-side code is delimited by rcu_read_lock_sched() and rcu_read_unlock_sched(), but the updater incorrectly uses synchronize_rcu() to wait for readers. Given this mismatch, the reader might execute lines 1 and 2, thus picking up the pointer to the current object referenced by gp. The updater might execute lines 1-3, where line 3's synchronize_rcu() waits for RCU readers. Unfortunately, line 3 waits only for those RCU readers delimited by rcu_read_lock() and rcu_read_unlock(), which does not include Figure 6's reader, which is instead delimited by rcu_read_lock_sched() and rcu_read_unlock_sched().

[5] The rcu_assign_pointer() primitive is common across all three flavors.

This means that the updater's line 3 could return immediately, so that line 4's `kfree()` will free the object that is still being used by the reader's call to `do_something_with()`. This can result in arbitrary misbehavior, but worse yet, because usermode code has a high degree of control over the contents of kernel memory (e.g., due to pathnames, data transmitted over the network, and so on), a malicious user could potentially cause the kernel to execute arbitrary code of the attacker's choice. In other words, unlike RCU's prior ease-of-use bugs, in this case the misbehavior can be (and in the case at hand, was) exploitable!

Fortunately, the fix was straightforward: Make readers and updaters use compatible RCU primitives. However, and to his credit, Linus Torvalds asked me to update RCU to eliminate this ease-of-use issue, and with it, this class of RCU-usage bugs. Nor was Linus the first to make this request, in fact, many developers have requested this over a period of many years. Furthermore, I attempted to avoid this ease-of-use bug when originally implementing preemptible RCU for real-time Linux. Unfortunately, that attempt failed.

However, the kernel has changed considerably during the intervening decade: (1) Multiple copies of architecture-specific code (sometimes coded in assembly) have been consolidated into core-kernel C functions,[6] which enabled RCU to precisely handle idle and CPU-hotplug transitions, which in turn eliminated a number of troublesome race conditions; (2) There are formal memory models for several CPU families [2, 22, 38]; (3) The Linux kernel memory model [1] removes guesswork from the art of Linux-kernel concurrent coding; and (4) RCU's guarantees have been formalized [1, 10, 11]. The models and formalizations mentioned in these last three items are all executable, that is, there are tools implementing each of these models. A given tool takes as input a *litmus test* that specifies small fragments of concurrent code and includes a predicate. The tool then produces as output a list of all final states that can be reached by all possible executions of the specified code fragments, given the constraints specified by the formal model. Finally, the tool produces an indication of whether all, some, or none of these final states satisfy the specified predicate. Executable formal models provide a welcome alternative to the sheer paranoia guiding earlier implementations of Linux-kernel RCU. The time therefore seemed ripe for a second attempt to remove RCU's ease-of-use bug.

One straightforward way to eliminate this ease-of-use bug is to change `synchronize_rcu()` to wait on all RCU readers, regardless of whether they use `rcu_read_lock()`, `rcu_read_lock_bh()`, `rcu_read_lock_sched()`, or any of the many synonyms of the last two primitives [31].[7] Straightforward to describe, that is; as described in the next section, the implementation posed some challenges.

[6] In other words, a single C-language hook may be used where tens of architecture-specific hooks, some in assembly, were once required.

```
1 rcu_read_lock_bh();
2 do_something_1();
3 rcu_read_lock();
4 rcu_read_unlock_bh();
5 do_something_2();
6 rcu_read_lock_sched();
7 rcu_read_unlock();
8 do_something_3();
9 rcu_read_unlock_sched();
```

```
p = remove_something_1_2_3();
synchronize_rcu();
synchronize_rcu_bh();
synchronize_sched();
kfree(p);
```

Figure 7: Nesting Disparate Critical Sections

```
1 { rcu_read_lock_bh(); rcu_read_lock(); }
2 do_something_1();
3 rcu_read_lock();
4 { rcu_read_unlock(); rcu_read_unlock_bh(); }
5 do_something_2();
6 { rcu_read_lock_sched(); rcu_read_lock(); }
7 rcu_read_unlock();
8 do_something_3();
9 { rcu_read_unlock(); rcu_read_unlock_sched(); }
```

Figure 8: Flattening Disparate Critical Sections

4 RCU'S EASE-OF-USE BUG: FIXES

An overlapping set of disparate critical sections must still be flattened into one large critical section, but fortunately non-preemptable kernels already do this correctly. Unfortunately, this correctness is an accident of implementation, due to the fact that non-preemptable kernels do not permit quiescent states such as `schedule()` to be invoked within any of RCU's critical sections. Unfortunately, preemption *can* occur—by design—within an `rcu_read_lock()` critical section in preemptible kernels. For example, although the code shown in Figure 7 cannot be preempted in non-preemptible kernels, the preemptibility of `rcu_read_lock()`-based critical sections means that preemption can occur at line 5 in preemptible kernels.[8] This code can be made safe by using all three grace-period-wait primitives, as shown on the right-hand side of the figure, but this sort of burden on the user is exactly what we are trying to avoid. Addressing this issue is the subject of the following sections.

4.1 Flattening Disparate Critical Sections

The disparate overlapping RCU read-side critical sections spanning lines 1-4, 3-7, and 6-9 of Figure 7 must be

[7] For example, `local_bh_disable()` is a synonym for `rcu_read_lock_bh()`, `local_bh_enable()` for `rcu_read_unlock_bh()`, `preempt_disable()` for `rcu_read_lock_sched()`, and `preempt_enable()` for `rcu_read_unlock_sched()`.

[8] Overlapping (as opposed to nesting) disparate RCU read-side critical sections is unconventional and should generally be avoided. However, there are a few valid use cases, so RCU must support this notion.

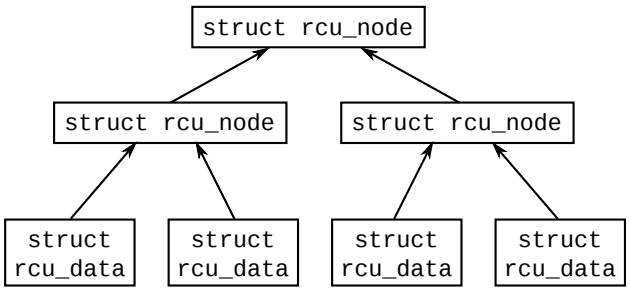

Figure 9: Linux-Kernel RCU Data Structures

flattened, that is, combined into a single critical section spanning lines 1-9. One way to do this is to make `rcu_read_lock_bh()` and `rcu_read_lock_sched()` invoke `rcu_read_lock()` just before returning and to make `rcu_read_unlock_bh()` and `rcu_read_unlock_sched()` invoke `rcu_read_unlock()` immediately upon entry, as depicted with curly braces in Figure 8. Because `rcu_read_lock()` and `rcu_read_unlock()` nest correctly, this would result in the following pairings: lines 3 and 4, lines 6 and 7, and lines 1 and 9. This last pairing covers the entire range, providing the required flattening.

This simple approach is correct by construction. Unfortunately, a few tests demonstrate that this correctness does not extend to the actual Linux kernel, in which preemption, bottom halves, and interrupts are enabled and disabled in assembly language and even by hardware. Thus, another idea is required, perhaps even the one in the next section.

4.2 Deferring Quiescent States

The problem in Figure 7 is that line 7's `rcu_read_unlock()` prematurely reports a quiescent state that was not reached until line 9. This suggests deferring reporting of any such quiescent states until after both preemption and bottom halves are enabled, in this case, on line 9, thus also delaying the end of the grace period. Understanding this alternative requires the brief overview of RCU's data structures. As shown in Figure 9, each CPU has an `rcu_data` structure, which is assigned to a leaf `rcu_node` structure making up a combining tree that limits lock contention while mediating reporting quiescent states. These quiescent states are reported up from the bottom of the tree, and when the root `rcu_node` structure detects that all required quiescent states have been reported, the grace period ends.

This deferral can be implemented straightforwardly by adding flags to the `rcu_data` and `rcu_node` structures, resulting in only five pages of hand-written code [32, slide 69]. Unfortunately, this initial code failed to account for stall warnings [29] (which RCU emits when CPUs or tasks become unresponsive), expedited grace periods [28] (which improve

```
 1 rcu_read_lock();
 2 do_something_1();
 3 preempt_disable();   /* AKA rcu_read_lock_sched() */
 4 do_something_2();
 5 rcu_read_unlock();
 6 do_something_3();
 7 rcu_read_lock();
 8 do_something_4();
 9 preempt_enable();    /* AKA rcu_read_unlock_sched() */
10 do_something_5();
11 rcu_read_unlock();
```

Figure 10: Quiescent-State-Deferral Nemesis

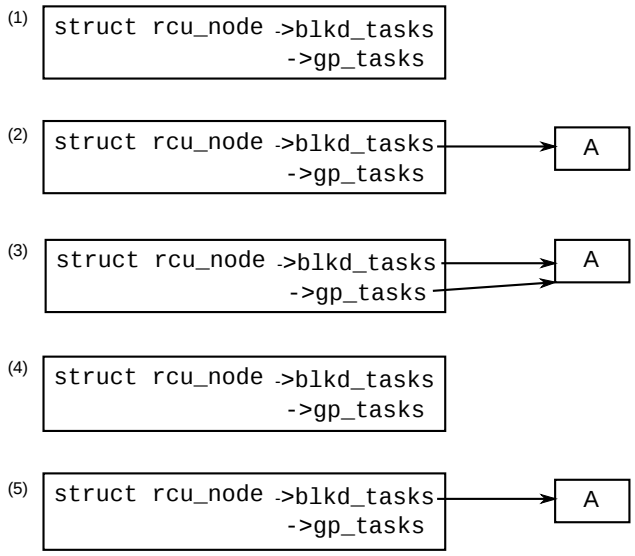

Figure 11: Preemptible RCU Blocked-Tasks List

latency at the expense of increased CPU consumption and degraded real-time response compared to normal grace periods), and preempted-task queuing corner cases [26] (in which the deferred quiescent state must sometimes follow the preempted task but other times stay with the CPU).

Accounting for these complications results in eight pages of code [32, slide 70], but fails to handle the overlapping readers shown in Figure 10. Figure 11 shows the resulting progression of states of one instance of RCU's leaf `rcu_node` structures, whose `blkd_tasks` and `gp_tasks` fields track tasks that have blocked within their current RCU read-side critical section. Because blocked tasks not on a `->blkd_tasks` list reside in a quiescent state, RCU can focus on tasks on these lists, avoiding expensive scans of the full task list.

If no task has recently blocked within a critical section, the system will be in state (1) of Figure 11. If Task A is preempted at line 2 of Figure 10, the system will advance to state (2): Task A has blocked within a critical section, but is not blocking a grace period. Once a grace period is initiated,

the system will advance to state (3), where ->gp_tasks indicates that Task A[9] blocks the current grace period. When Task A resumes and reaches line 5, it will remove itself from the list, and, because it is the last task in this list, NULL the ->gp_tasks pointer, resulting in state (4).

The problem arises when Task A reaches line 7. The grace period has already started, and Task A appears to be entering a new critical section that therefore fails to block the grace period. This failure to block is incorrect because the entire block of code spanning lines 1-11 must be treated as one large critical section. The ->gp_tasks pointer therefore (again incorrectly) remains NULL. The grace period might therefore end just after line 9 when it instead needs to extend past line 11. In short, this approach has failed to flatten the entire set of critical sections spanning lines 1-11.

Although adding more flags might solve this problem, the steady growth in code size is a subtle hint to step back and reconsider the situation, as is done in the next section.

4.3 Deferring Task Dequeuing

The problem with the last section's approach is that when it added Task A to the ->blkd_tasks list, it failed to record that Task A still blocked the grace period. In the spirit of the medical advice "Then don't do that!", let's try avoiding this second addition altogether. For example, rcu_read_unlock() could refrain from removing Task A when within some other type of critical section, in this case the critical section beginning with line 3's preempt_disable(). This would result in Task A remaining in state (3) after executing the rcu_read_unlock() on line 7. In fact, Task A would remain in that state through line 11, as is required to flatten all of Figure 10's disparate critical sections.

This means that tasks can now be on the ->blkd_tasks list despite not being within RCU read-side critical sections, and this expansion of RCU's state space deserves intensive validation. Also, such tasks must dequeue themselves upon reaching quiescent states. Expedited grace periods can require them to dequeue themselves sooner rather than later, which necessitates calls to the scheduler or use of softirq handlers [8, 21], depending on the situation [32, slide 83].

The code for this idea consumes only three sheets of paper, down from eight (and growing) for that of the previous section. Time to actually type it in!

5 IMPLEMENTATION

Implementing Section 4.3's core idea required only eight patches, and these patches added a total of only 274 lines and removed 112 [32, slide 100]. However, these eight patches could not be applied directly to the kernel. Instead, a large number of preparation and cleanup patches were needed, as summarized in the next section.

5.1 Preparation and Cleanup

For historical reasons, RCU uses a pair of counters to represent grace-period state, so that a lock must be held when reading out that state. The addition of deferred quiescent states adds readouts of grace-period state, including on the scheduler fastpath. Adding locks on such fastpaths is clearly not a strategy to win, so the grace-period state was consolidated into a single integer that can be read out locklessly. This conceptually simple task required 35 patches.

Because consolidating the three RCU flavors reduces the number of locks, it also increases lock contention. Therefore, three additional patches implemented funnel locking, which is a combining-tree mechanism that reduces lock contention during grace-period initiation [16, 20, 28]. Referring to Figure 9, a given CPU starts the funnel-locking process at its leaf rcu_node structure. The CPU acquires that structure's lock and checks to see if the desired grace period is already at least in the process of being started, and if so, the CPU releases locks as needed and returns to its caller: Its work is at least already being started by some other CPU. Otherwise, the CPU sets state indicating that the desired grace period is being started, thus preventing subsequent CPUs from wasting any further effort attempting to start this same grace period. The CPU then proceeds to the next rcu_node structure up the tree and repeats this process. If the CPU reaches the root node, and has not found any evidence of some other CPU starting the desired grace period, then this CPU starts that grace period.[10] The key point is that only one of the CPUs corresponding to a given rcu_node structure will advance to that structure's parent, which in turn means that lock contention on each rcu_node structure remains bounded, regardless of the number of CPUs in the system.

Another set of 15 patches fixed some rcutorture[11] false-positive reports, removing the possibility that these reports might hide an actual bug introduced by this work. Another 17 patches added debugging code to help locate such bugs.

Previous versions of rcutorture do not test partially nested disparate RCU read-side critical sections, nor do they include forward-progress tests, which verify that RCU can advance its grace periods despite call_rcu() being invoked in a tight loop. These tests were added by another 42 patches, which are discussed in Section 5.2.

Consolidating the RCU-bh, RCU-preempt, and RCU-sched flavors into a single flavor (RCU-preempt for preemptible

[9] And, if there were any, tasks following Task A in the ->blkd_tasks list.

[10] The actual funnel-locking algorithm must also handle the possibility that a prior grace period is still in progress, as well as a few concurrency issues. However, this description provides a useful conceptual view. Those wishing more detail should inspect rcu_start_this_gp() and its callers.

[11] This is Linux-kernel RCU's stress-test suite.

kernels and RCU-sched for non-preemptible kernels) enabled many simplifications, which were implemented by another 107 patches. These simplifications included eliminating `synchronize_rcu_bh()` and `synchronize_sched()` in favor of `synchronize_rcu()`, however, the read-side markers (including `rcu_read_lock_bh()` and `rcu_read_lock_sched()`) were retained for readability. For example, although it is possible to replace `rcu_read_lock_sched()` with `preempt_disable()`, the former provides a helpful indication that RCU is involved. These simplifications also slightly reduced scheduler fastpath overhead and significantly improved cache locality, resulting in an unplanned microbenchmark improvement [17].

Finally, the repeated inspection of the code required by this work located a number of simple optimizations, which were implemented by another 17 patches.

Although `rcutorture` results have been quite clean, the fact is that this work applied more than 200 patches to Linux-kernel RCU [32, slides 100–118]. Large changes imply a large need for validation, a topic taken up by the next section.

5.2 Validation

RCU commits are subject to kbuild test robot testing [18] as soon as they become public. The testing located a number of build issues on various architectures, a boot-time issue, and a number of runtime issues, including deadlocks and failure to properly handle RCU reader corner cases. Additional bugs were located by code review, `-next` testing [7], and other community processes, all of which are deeply appreciated.

This change requires RCU to flatten disparate RCU read-side critical sections such as those shown in Figures 7 and 10 even in preemptible kernels. Therefore, `rcutorture` must intensively test overlapping disparate critical sections. It randomly chooses a number of overlapping critical sections so that singleton critical sections will be used about 1.6% of the time and eight-fold overlapping critical sections about 40% of the time, with the expected case being six-fold overlapping critical sections. Similarly, a given critical section will use only one type of RCU reader protection about 90% of the time (as needed to ensure preemption of RCU readers happens reasonably frequently), and will use multiple types otherwise. The point of these seemingly arbitrary choices is to force random testing of transitions among elaborate combinations of RCU readers with an eye to catching improbable corner cases in the comfort and safety of the lab, as opposed to in much less friendly production environments.

RCU read-side performance is of the essence, but RCU's read-side fastpaths are unchanged, so it is no surprise that RCU read-side performance is not affected by this change.

Of course, the whole point of validation is to reduce risk during production use, a topic addressed by the next section.

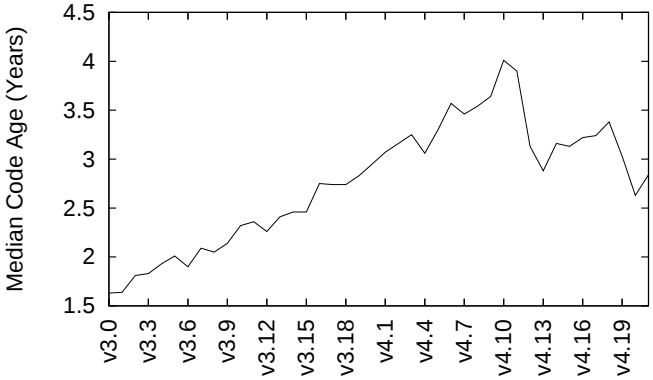

Figure 12: Median Age of Lines of RCU Code

6 EFFECT ON RCU RELIABILITY

One immediate measure of RCU reliability is `rcutorture` results, which are quite good, reflecting the resolution of the false-positive warnings as part of this work.

Another measure is code age, with older code often being considered more reliable. Of course, RCU is a mixture of code with a wide range of ages, so a statistical representation is necessary. The median was chosen as an easily understood and relatively stable statistic. Figure 12 shows that the median age of RCU lines of code has decreased about 30% since v4.10, first due to the 2018 rewrite of the Sleepable RCU (SRCU) [24] implementation, and, starting with v4.18, due to this effort. There is thus some cause for concern, although the median age has risen by about 75% since v3.0, despite the recent decrease.

Another approach is to look at incoming bug rates, calibrating using the SRCU rewrite. Of 2018's six most significant bugs, two were caused by the SRCU rewrite, one of which was a data race due to an omitted lock and the other of which was a real-time response issue [32, slide 116]. This data might lead one to predict that the work described in Section 5 would generate an additional pair of bugs in 2019, along with at least one more bug from the SRCU rewrite. One objection to this prediction is that the number of bugs is nowhere near large enough to be statistically significant, however, it would be extremely unwise to wish to live in a world where the number of Linux-kernel RCU bugs was statistically significant. Another objection is that Murphy cannot be expected to neglect Linux-kernel RCU.

Murphy notwithstanding, Linux-kernel RCU seems to be in reasonably good shape. Nevertheless, the large recent changes provide a golden opportunity for those working in the area of formal verification.

7 CONCLUSIONS

This experience underscores the well-known fact that making your software do exactly what you want is highly non-trivial. Worse yet, doing so is in fact insufficient: As was the case here, requirements might not be fully understood. Other experiences over the past few years show that these issues also affect hardware [13]. This experience also underscores the value of executable formal methods [1, 2, 22, 38], even to the famously pragmatic Linux kernel community.

But the most important lesson is the critical importance of ease of use, even for low-level APIs such as RCU. Of course, the need for careful API design has been long appreciated by the Linux kernel community [41, slides 39-57], and, as noted in Section 3, this is not the first RCU ease-of-use improvement. However, this is the first RCU ease-of-use problem known to result in an exploitable vulnerability. Designers of even low-level APIs must therefore keep the convenience and proclivities of their users firmly in mind, not just as a service to those users, but also to avoid exploitable vulnerabilities.

ACKNOWLEDGMENTS

I am grateful for review of early drafts of this paper from Joel Nider and Mike Rapoport, and for code review from Andrea Parri. We all owe Orna Agmon Ben-Yehuda a debt of gratitude for her shepherding, which helped render this paper human-readable. I am grateful to Mark Figley for his support of this effort.

REFERENCES

[1] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 405–418. https://doi.org/10.1145/3173162.3177156

[2] Jade Alglave, Luc Maranget, Pankaj Pawan, Susmit Sarkar, Peter Sewell, Derek Williams, and Francesco Zappa Nardelli. 2011. PPCMEM/ARMMEM: A Tool for Exploring the POWER and ARM Memory Models. (4 June 2011). http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/pldi105-sarkar.pdf.

[3] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. 2003. Using Read-Copy Update Techniques for System V IPC in the Linux 2.5 Kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*. USENIX Association, Berkeley, CA 94710, USA, 297–310. https://www.usenix.org/legacy/publications/library/proceedings/usenix03/tech/freenix03/full_papers/arcangeli/arcangeli.pdf

[4] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *9th USENIX Symposium on Operating System Design and Implementation*. USENIX, Vancouver, BC, Canada, 1–16.

[5] Jonathan Corbet. 2005. Realtime preemption and read-copy-update. (March 2005). URL: http://lwn.net/Articles/129511/.

[6] Jonathan Corbet. 2006. The kernel lock validator. (31 May 2006). Available: http://lwn.net/Articles/185666/ [Viewed: March 26, 2010].

[7] Jonathan Corbet. 2008. A day in the life of linux-next. (23 June 2008). https://lwn.net/Articles/287155/.

[8] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux Device Drivers, Third Edition* (3 ed.). O'Reilly Media, Inc., Sebastopol, CA 95472, USA. URL: https://lwn.net/Kernel/LDD3/.

[9] Mathieu Desnoyers. 2009. [RFC git tree] Userspace RCU (urcu) for Linux. (5 February 2009). http://liburcu.org.

[10] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23 (2012), 375–382. https://doi.org/10.1109/TPDS.2011.159

[11] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2012. Verifying Highly Concurrent Algorithms with Grace (extended version). (10 July 2012). http://software.imdea.org/~gotsman/papers/recycling-esop13-ext.pdf.

[12] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. 2008. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal* 47, 2 (May 2008), 221–236. https://doi.org/10.1147/sj.472.0221

[13] Jann Horn. 2018. Reading privileged memory with a side-channel. (3 January 2018). https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html.

[14] Phil Howard. 2012. *Extending Relativistic Programming to Multiple Writers*. Ph.D. Dissertation. Portland State University.

[15] Philip W. Howard and Jonathan Walpole. 2011. A Relativistic Enhancement to Software Transactional Memory. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism (HotPar'11)*. USENIX Association, Berkeley, CA, USA, 1–6. http://www.usenix.org/event/hotpar11/tech/final_files/Howard.pdf

[16] Wilson C. Hsieh and William E. Weihl. 1992. Scalable Reader-Writer Locks for Parallel Systems. In *Proceedings*

of the 6th International Parallel Processing Symposium. IEEE Computer Society, Washington, DC, USA, 216–230. https://doi.org/10.1109/IPPS.1992.222989

[17] kernel test robot. 2019. [LKP] [rcu] 7e28c5af4e: will-it-scale.per_process_ops 84.7% improvement. (28 February 2019). https://lkml.org/lkml/2019/2/27/829.

[18] Michael Kerrisk. 2012. KS2012: Kernel build/boot testing. (5 September 2012). https://lwn.net/Articles/514278/.

[19] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 361–372. https://doi.org/10.1145/2678373.2665726

[20] Beng-Hong Lim and Anant Agarwal. 1994. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems (ASPLOS VI)*. ACM, New York, NY, USA, 25–35. https://doi.org/10.1145/195473.195490 URL: http://groups.csail.mit.edu/cag/pub/papers/pdf/reactive.pdf.

[21] Robert Love. 2005. *Linux Kernel Development* (second ed.). Novell Press, Provo, UT USA.

[22] Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. (10 October 2012). https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.

[23] Paul E. McKenney. 2006. RCU Linux Usage. (October 2006). Available: http://www.rdrop.com/users/paulmck/RCU/linuxusage.html [Viewed January 14, 2007].

[24] Paul E. McKenney. 2006. Sleepable RCU. (9 October 2006). Available: http://lwn.net/Articles/202847/ Revised: http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf [Viewed August 21, 2006].

[25] Paul E. McKenney. 2010. Lockdep-RCU. (1 February 2010). https://lwn.net/Articles/371986/.

[26] Paul E. McKenney. 2010. Simplicity Through Optimization. In *linux.conf.au 2010*. linux.conf.au, Wellington, New Zealand, 109. Available: http://www.rdrop.com/users/paulmck/RCU/SimplicityThruOptimization.2010.01.21f.pdf [Viewed October 10, 2010].

[27] Paul E. McKenney. 2013. Structured deferral: synchronization via procrastination. *Commun. ACM* 56, 7 (July 2013), 40–49. https://doi.org/10.1145/2483852.2483867

[28] Paul E. McKenney. 2016. What Happens When 4096 Cores All Do `synchronize_rcu_expedited()`? (3 February 2016). linux.conf.au http://www2.rdrop.com/users/paulmck/RCU/4096CPU.2016.02.03i.pdf.

[29] Paul E. McKenney. 2018. Decoding Those Inscrutible RCU CPU Stall Warnings. (22 January 2018). linux.conf.au Kernel Mini-conf. Slides: http://www.rdrop.com/~paulmck/RCU/stallwarning.2018.01.22a.pdf Video: https://www.youtube.com/watch?v=23_GOr8Sz-E.

[30] Paul E. McKenney. 2018. *Is Parallel Programming Hard, And, If So, What Can You Do About It? (2018.12.08a Release)*. kernel.org, Corvallis, OR, USA. https://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.2018.12.08a.pdf

[31] Paul E. McKenney. 2019. The RCU API, 2019 Edition. (23 January 2019). https://lwn.net/Articles/777036/.

[32] Paul E. McKenney. 2019. RCU's First-Ever CVE, and How I Lived to Tell the Tale. (23 January 2019). linux.conf.au Slides: http://www.rdrop.com/users/paulmck/RCU/cve.2019.01.23e.pdf Video: https://www.youtube.com/watch?v=hZX1aokdNiY.

[33] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2001. Read-Copy Update. In *Ottawa Linux Symposium*. Ottawa Linux Symposium, Ottawa, Canada, 22. URL: https://www.kernel.org/doc/ols/2001/read-copy.pdf, http://www.rdrop.com/users/paulmck/RCU/rclock_OLS.2001.05.01c.pdf.

[34] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. 2013. The URCU hash table API. (12 November 2013). https://lwn.net/Articles/573432/.

[35] Paul E. McKenney, Maged Michael, and Peter Sewell. 2019. N2369: Pointer lifetime-end zap. (1 April 2019). http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2369.pdf.

[36] Paul E. McKenney and John D. Slingwine. 1998. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*. Acta Press, Las Vegas, NV, 509–518. http://www.rdrop.com/users/paulmck/RCU/rclockpdcsproof.pdf

[37] Donald E. Porter and Emmett Witchel. 2007. Lessons From Large Transactional Systems. (December 2007). Personal communication <20071214220521.GA5721@olive-green.cs.utexas.edu>.

[38] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158107

[39] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. 2007. MetaTM/TxLinux: transactional

memory for an operating system. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 92–103. https://doi.org/10.1145/1273440.1250675

[40] Geoff Romer and Andrew Hunter. 2018. An RAII Interface for Deferred Reclamation. (3 March 2018). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0561r4.html.

[41] Rusty Russell. 2003. Hanging Out With Smart People: or... Things I Learned Being A Kernel Monkey. (25 July 2003). 2003 Ottawa Linux Symposium Keynote http://ozlabs.org/~rusty/ols-2003-keynote/ols-keynote-2003.html.

[42] Dipankar Sarma. 2004. RCU: Introduce call_rcu_bh() [2/5]. (8 August 2004). https://lkml.org/lkml/2004/8/6/228.

[43] Dipankar Sarma. 2004. RCU: Use call_rcu_bh() in route cache [3/5]. (8 August 2004). https://lkml.org/lkml/2004/8/6/231.

[44] Dipankar Sarma. 2004. Re: RCU : Abstracted RCU dereferencing [5/5]. (August 2004). Available: http://lkml.org/lkml/2004/8/6/237 [Viewed June 8, 2010].

[45] Dipankar Sarma and Paul E. McKenney. 2004. Issues with Selected Scalability Features of the 2.6 Kernel. In *Ottawa Linux Symposium*. Ottawa Linux Symposium, Ottawa, Canada, 16. https://www.kernel.org/doc/ols/2004/ols2004v2-pages-195-208.pdf.

[46] Dipankar Sarma and Paul E. McKenney. 2004. Making RCU Safe for Deep Sub-Millisecond Response Realtime Applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)*. USENIX Association, Berkeley, CA 94710, USA, 182–191. http://www.rdrop.com/~paulmck/RCU/realtimeRCU.2004.06.12a.pdf

[47] Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2017. Combining HTM and RCU to Implement Highly Efficient Balanced Binary Search Trees. In *12th ACM SIGPLAN Workshop on Transactional Computing*. ACM, New York, NY, USA.