Verification of Tree-Based Hierarchical Read-Copy Update in the Linux Kernel

Lihao Liang
University of Oxford

Paul E. McKenney
IBM Linux Technology Center

Daniel Kroening and Tom Melham
University of Oxford

***Abstract*—Read-Copy Update (RCU) is a scalable, high-performance Linux-kernel synchronization mechanism that runs low-overhead readers concurrently with updaters. Production-quality RCU implementations are decidedly non-trivial and their stringent validation is mandatory. This suggests use of formal verification. Previous formal verification efforts for RCU either focus on simple implementations or use modeling languages. In this paper, we construct a model directly from the source code of Tree RCU in the Linux kernel, and use the CBMC program analyzer to verify its safety and liveness properties. To the best of our knowledge, this is the first verification of a significant part of RCU's source code—an important step towards integration of formal verification into the Linux kernel's regression test suite.**

I. Introduction

The Linux operating system kernel [1] is widely used, for example in servers, safety-critical embedded systems, household appliances, and mobile devices. Over the past 25 years, many technologies have been added to the Linux kernel, one example being Read-Copy Update (RCU) [2].

RCU is a synchronization mechanism used to replace reader-writer locks in read-mostly scenarios, allowing low-overhead readers to run concurrently with updaters. Production-quality implementations for multi-core systems must provide excellent scalability, high throughput, low latency, modest memory footprint, excellent energy efficiency, and reliable response to CPU hotplug operations. They must therefore avoid cache misses, lock contention, frequent updates to shared variables, and excessive use of atomic read-modify-write and memory-barrier instructions. Finally, implementations must cope with the extremely diverse workloads and platforms of Linux [3].

RCU is now widely used in the Linux-kernel networking, device-driver, and file-storage subsystems [3], [4]. There are at least 75 million Linux servers [5] and 1.4 billion Android devices [6], so a "million-year" bug can occur several times per day across the installed base. Stringent validation of RCU's complex implementation is thus critically important.

Formal verification has already been applied to some aspects of RCU design, including Tiny RCU [7], userspace RCU [8], sysidle [7], and interactions between dyntick-idle and non-maskable interrupts (NMIs) [9]. But these efforts either validate trivial single-CPU RCU implementations in C or use special-purpose languages such as Promela [10]. A major disadvantage of special-purpose modeling languages is difficult and error-prone translation from source code. Other research has done manual proof of simple RCU implementations [11], [12], but this requires significant effort beyond translation. Linux kernel releases are only about 60 days apart, and RCU changes with each release. So any manual work must be replicated about six times a year.

If formal verification is to be part of Linux-kernel RCU's regression suite, it must be scalable and automated. This paper describes how to build a model directly from the Linux kernel source code, and use the C Bounded Model Checker (CBMC) [13] to verify RCU's safety and liveness properties.

This work is supported by EPSRC EP/H017585/1, a gift from Intel Corporation for research on Effective Validation of Firmware, ERC project 280053 "CPROVER", the H2020 FET OPEN 712689 SC^2 and SRC contracts 2012-TJ-2269 and 2016-CT-2707.

II. Background

RCU is used in read-mostly situations. Readers run concurrently with updaters, so RCU maintains multiple versions of objects and ensures they are not freed until all pre-existing readers complete, after a *grace period* elapses. The idea is to split updates into removal and reclamation phases [2]. The removal phase makes objects inaccessible to readers, waits during the grace period, and then reclaims them. Grace periods need wait only for readers whose runtime overlaps the removal phase. Readers starting after the removal phase ends cannot hold references to any removed objects and thus cannot be disrupted by objects being freed during the reclamation phase.

Modern CPUs guarantee that writes to single aligned pointers are atomic, so readers see either the old or new version of a data structure. This enables atomic insertions, deletions, and replacements in a linked structure. Readers can then avoid expensive atomic operations, memory barriers, and cache misses. In the most aggressive configurations of Linux-kernel RCU, readers use the same sequence of instructions that would be used in a single-threaded implementation, providing RCU readers with excellent performance and scalability.

A. Core RCU API Usage

The core API has five primitives [3], which we now introduce. A read-side critical section begins with `rcu_read_lock()` and ends with `rcu_read_unlock()`. When nested, they are flattened into one critical section. Within a critical section, it is illegal to block, but preemption is legal. RCU-protected data accessed by a read-side critical section will not be reclaimed until it completes. The function `synchronize_rcu()` marks the boundary between removal and reclamation, so must block until all pre-existing read-side critical sections have completed.

© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. Published in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19–23, 2018*, edited by J. Madsen and A. K. Coskun (EDAA, 2018), pp. 61–66. DOI: 10.23919/DATE.2018.8341980

But synchronize_rcu() need not wait for critical sections that begin after it does. Updaters use rcu_assign_pointer() to assign a new value to an RCU-protected pointer; readers use rcu_dereference() to fetch that RCU-protected pointer, which can then be safely dereferenced, but only within the enclosing read-side critical section.

B. Implementation of Tree RCU

The primary advantage of RCU is that it is able to wait for a very large number of readers to finish without tracking them all. Performance and scalability relies on efficient mechanisms to detect when a grace period has completed. A simplistic implementation might require each CPU to acquire a global lock during each grace period, but this would not scale beyond a few hundred CPUs. The fact that Linux runs on systems with thousands of CPUs motivated the creation of Tree RCU.

We focus on the 'vanilla' API in a non-preemptible build of the Linux kernel, specifically on rcu_read_lock(), rcu_read_unlock(), and synchronize_rcu(). The key idea is that RCU read-side primitives are confined to kernel code and do not voluntarily block. So when a CPU passes through a *quiescent state* (context switch, is idle/offline, or runs in user mode), that CPU's prior read-side critical sections must have finished. After each CPU has passed through a quiescent state, the corresponding RCU grace period ends.

The key challenge is to know when all quiescent states are reached. Recording quiescent states in one single location would result in extreme contention on large systems. To achieve excellent performance and scalability, Tree RCU uses a hierarchy of data structures, each leaf of which records the corresponding CPU's quiescent states. Once a node's children have recorded a full set of quiescent states, that node propagates the quiescent states up toward the root. When the root is reached, a grace period has ended and notification is propagated down. Shortly after a leaf receives this notification, synchronize_rcu() calls on the corresponding CPU will return.

Additional details on Tree RCU are available in [14].

III. Verification Scenario

We use the following example to show how Tree RCU guarantees that all pre-existing read-side critical sections finish before it allows a grace period to end.

```
int x = 0, y = 0, r1, r2;

void rcu_reader(void) {        void rcu_updater(void) {
        rcu_read_lock();               x = 1;
        r1 = x;                        synchronize_rcu();
        r2 = y;                        y = 1; }
        rcu_read_unlock(); }

assert(r2 == 0 || r1 == 1); // after both functions return
```

This example also drives the verification, which checks for violations of the assertion that follows the code.

We focus on the non-preemptible RCU-sched flavor. We assume there are only two CPUs, and that CPU 0 runs rcu_reader() and CPU 1 runs rcu_updater(). When the system boots, the Linux kernel first initializes RCU, which includes making the tree of rcu_node and rcu_data structures. The rcu_node structure records and propagates quiescent-state information from leaves to the root, and also propagates grace-period information from the root to the leaves. The per-CPU rcu_data structure detects quiescent states and handles RCU callbacks for that CPU [14]. Our example has a one-level tree, with one rcu_node root and two rcu_data children.

Suppose CPU 0 invokes rcu_reader() while CPU 1 invokes rcu_updater(), setting x to 1 and then invoking synchronize_rcu(). This then invokes wait_rcu_gp(), an internal function that uses callbacks to invoke wakeme_after_rcu() some time after rcu_reader() exits its critical section—i.e., after a grace period. As its name suggests, wakeme_after_rcu() wakes up wait_rcu_gp(); this returns, allowing synchronize_rcu() to return control to its caller.

This critical-section exit has no immediate effect. A later context switch will invoke rcu_note_context_switch(), which invokes rcu_sched_qs() to record the quiescent state in a field of the corresponding CPU's rcu_data structure. Later, a scheduling-clock interrupt will invoke rcu_check_callbacks() noting that this field is set. This will in turn cause rcu_check_callbacks() to invoke raise_softirq(RCU_SOFTIRQ), which, once the CPU has interrupts, preemption, and bottom halves enabled, calls rcu_process_callbacks().

RCU's softirq handler function rcu_process_callbacks() first calls rcu_check_quiescent_state() to report any recent quiescent states on the current CPU (CPU 0). Since a quiescent state has been recorded for CPU 0, rcu_report_qs_rnp() is invoked to traverse up the combining tree. It clears the first bit of the root rcu_node structure's qsmask field, which indicates which of this node's children still need to report quiescent states for the current grace period [14]. Since the second bit for CPU 1 has not been cleared, the function returns.

Since synchronize_rcu() blocks in CPU 1, it will result in a context switch. This triggers a sequence of events similar to that described above for CPU 1, which results in the clearing of the second bit of the root rcu_node structure's ->qsmask field, the value of which is now 0, indicating the end of the current grace period. CPU 1 therefore invokes rcu_report_qs_rsp() to awaken the grace-period kernel thread, which will clean up the ended grace period, and, if needed, start a new one.

Finally, rcu_process_callbacks() calls the function invoke_rcu_callbacks() to invoke any callbacks whose grace period has already elapsed, for example, wakeme_after_rcu(), which will allow synchronize_rcu() to return.

IV. Modeling RCU for CBMC

CBMC [13] implements bit-precise bounded model checking for C programs. CBMC can show violation of assertions or prove their safety under a given loop unwinding bound. It translates an input C program into a formula, which is passed to a SAT or SMT solver together with a set of error states. If the solver determines the formula to be satisfiable, an error trace is extracted from the satisfying assignment. CBMC supports verification of concurrent programs over a range of memory models, including SC, TSO, and PSO [15].

The remainder of this section describes building a model from Linux kernel v4.3.6 Tree RCU's code, which we verified with CBMC. Model construction entailed stubbing out calls to other parts of the kernel, removing irrelevant functionality (such as idle-CPU detection), removing irrelevant data (such as statistics), and conditionally injecting bugs (see Sec. V-A). The Linux kernel environment and most of the source-code changes are made through macros in separate files, reusable across different versions of the implementation. The biggest change in the source files is to use arrays to model per-CPU data, which can be scripted [16]. The resulting model has 8,626 lines of C code. Around 900 lines model the Linux kernel environment, which is much smaller than the actual Linux kernel code used by the Tree RCU implementation. The model contains assertions and can be also run as a user program, which provides important validation of the model itself.

Initialization: Our model first invokes `rcu_init()`. This then invokes `rcu_init_geometry()` to compute the `rcu_node` tree geometry, `rcu_init_one` to initialize the `rcu_state` structure (which includes an array of `rcu_node` structures organized as a tree with `rcu_data` structures at the leaves [14]), and `rcu_cpu_notify()` to initialize each CPU's `rcu_data` structure. This initialization tunes the data-structures to match the specific hardware used. The model then calls `rcu_spawn_gp_kthread()` to spawn the grace-period kthreads discussed below.

Per-CPU Variables and State: We model per-CPU `rcu_data` as an array, indexed by CPU ID. It is also necessary to model per-CPU state, including the currently running task and whether or not interrupts are enabled. Identifying the running task requires a (trivial) model of the Linux-kernel scheduler, which uses an integer array indexed by CPU ID. Each element of this array models an exclusive lock. When a task schedules on a given CPU, it acquires the corresponding CPU lock, and releases it when scheduling away. We currently do not model preemption, so need to model only voluntary context switches.

A pair of integer arrays `local_irq_depth` and `irq_lock` is used to model CPUs enabling and disabling interrupts. Both arrays are indexed by CPU ID, with the first recording each CPU's interrupt-disable nesting depth and the second recording whether or not interrupts are disabled.

Update-Side API: Our model omits CPU hotplug and callback handling, so we cannot use Tree RCU's normal callback mechanisms to detect the end of a grace period. We therefore use a global variable `wait_rcu_gp_flag`, initialized to 1 in `wait_rcu_gp()` before the grace period. Because `wait_rcu_gp()` blocks, it can result in a context switch; so the model invokes `rcu_note_context_switch()`, followed by a call to `rcu_process_callbacks()` to inform RCU of the resulting quiescent state. When the resulting quiescent states propagate to the root of the tree, the grace-period kernel thread is awakened. This kthread then invokes `rcu_gp_cleanup()`, the modeling of which is described below. Then `rcu_gp_cleanup()` calls `rcu_advance_cbs()`, which invokes `pass_rcu_gp()` to clear the `wait_rcu_gp_flag` flag. Inserting `__CPROVER_assume(wait_rcu_gp_flag == 0)` in `wait_rcu_gp()` prevents CBMC from continuing execution until `wait_rcu_gp_flag` is equal to 0, thus modeling the needed grace-period wait.

Scheduling-Clock Interrupt and Context Switch: `rcu_check_callbacks()` detects idle and usermode execution, as well as invokes RCU core processing in response to state changes. We model neither idle nor usermode execution, so the only state changes are context-switches and the beginnings and ends of grace periods. So we dispense with `rcu_check_callbacks()`. Instead, we directly call `rcu_note_context_switch()` just after releasing a CPU, which in turn calls `rcu_sched_qs()` to record the quiescent state. Finally, we call `rcu_process_callbacks()`, which notes grace-period beginnings and ends and reports quiescent states up RCU's combining tree.

Grace-Period Kernel Thread: `rcu_gp_kthread()` invokes `rcu_gp_init()`, `rcu_gp_fqs()`, and `rcu_gp_cleanup()` to initialize, wait for, and clean up after each grace period, respectively. To reduce the size of the formula CBMC generates, instead of spawning a separate thread, we invoke `rcu_gp_init()` from `rcu_spawn_gp_kthread()` and `rcu_gp_cleanup()` from `rcu_report_qs_rsp()`. Because we model neither idle nor usermode execution, we need not call `rcu_gp_fqs()`.

Kernel Spin Locks: CBMC's `__CPROVER_atomic_begin()`, `__CPROVER_atomic_end()`, and `__CPROVER_assume()` built-in primitives are used to construct atomic test-and-set for `spinlock_t` and `raw_spinlock_t` acquisition and atomic reset for release. We use GCC atomic builtins for user-space execution: `while (__sync_lock_test_and_set(lock, 1))` acquires a lock and `__sync_lock_release(lock)` releases it.

Limitations: We model only the fundamental parts of Tree RCU, excluding quiescent-state forcing, grace-period expediting, and callback handling. We assume all CPUs are busy executing RCU related tasks, so we do not model CPU hotplug, dyntick idle, RCU priority boosting, or thread-migration failure modes in the Linux kernel involving per-CPU variables. Nonetheless, we model real-world server-class RCU code paths and data layout on systems with up to 16 CPUs (default configurations) or up to either 32 or 64 CPUs (non-default configurations on either 32-bit or 64-bit CPUs). We also model scheduling-clock interrupts as function calls; as discussed later, this results in failure to model one of the bug-injection scenarios. Finally, our test harness passes through only one grace period, so cannot detect failures involving multiple grace periods.

V. Experiments

We now discuss our experiments, which were performed on a 64-bit machine running Linux 3.19.8 with eight Intel Xeon 3.07 GHz cores and 48 GB of memory. The source code of our RCU model and the experimental data are available at https://github.com/lihaol/verify-treercu/releases/tag/date18-camera-ready.

A. Bug-Injection Scenarios

We model non-preemptible Tree RCU, so each CPU runs exactly one RCU task as a separate thread. On completion, each task increments a global counter `thread_cnt`, enabling

the parent thread to verify the completion of all RCU tasks using a statement `__CPROVER_assume(thread_cnt == 2)`. The base case is the example in Sect. III, including the assertion, which does not hold when RCU's safety guarantee is violated: read-side critical sections cannot span grace periods.We also verify a weak form of liveness by inserting `assert(0)` after the above statement. This assertion cannot hold, so it will be violated if any grace period completes. This 'failure' is really correct RCU behavior. But if the assertion is *not* violated, grace periods never complete, indicating a liveness bug.

To validate our verification, we also run CBMC with the following bug-injection scenarios.[1] These are simplified versions of bugs encountered in actual practice. Bugs 2–6 are liveness checks and so use the aforementioned `assert(0)`; the others are safety checks, which use the assertion in Sect. III.

Bug 1: This makes `synchronize_rcu()` return immediately (line 523 in `tree_plugin.h`). Updaters then never wait for readers, which should result in a safety violation.

Bug 2: This stops individual CPUs realizing that quiescent states are needed, preventing the CPUs from recording them. Grace periods then do not complete. In `rcu_gp_init()`, for each `rcu_node` structure, we set the field `rnp->qsmask` to 0 (line 1889 in `tree.c`). When `rcu_process_callbacks()` is called, `rcu_check_quiescent_state()` will invoke `__note_gp_changes()` that sets `rdp->qs_pending` to 0, indicating that RCU needs no quiescent state from the corresponding CPU. So `rcu_check_quiescent_state()` will return without calling `rcu_report_qs_rdp()`, preventing grace periods completing.

Bug 3: This is a variant of Bug 2, in which each CPU remains aware that quiescent states are needed but incorrectly believes it has already reported a quiescent state for the current grace period. In `__note_gp_changes()`, we clear `rnp->qsmask` by adding `rnp->qsmask &= ~rdp->grpmask;` in the last `if` code block (line 1739 in `tree.c`). So `rcu_report_qs_rnp()` never walks up the `rcu_node` tree, resulting in a liveness violation.

Bug 4: This is an alternative code change that gets the same effect as Bug 2. In `__note_gp_changes()`, we set the `rdp->qs_pending` field to 0 directly (line 1749 in `tree.c`).

Bug 5: CPUs remain aware of the need for quiescent states but are prevented from recording theirs, so grace periods do not complete. We modify function `rcu_sched_qs()` to return immediately (line 246 in `tree.c`), so that quiescent states are not recorded. Grace periods therefore never complete.

Bug 6: CPUs are aware of the need for quiescent states and also record them locally, but CPUs are prevented from reporting them up the `rcu_node` tree, which again prevents grace periods from completing. We modify `rcu_report_qs_rnp()` to return immediately (line 2227 in `tree.c`). This prevents RCU from walking up the `rcu_node` tree, thus preventing grace periods from ending. This is a liveness violation similar to Bug 2.

Bug 7: This bug causes quiescent states to be reported up the tree prematurely, before the CPUs covered by a given subtree have all reported quiescent states. In `rcu_report_qs_rnp()`, we remove the `if`-block checking for `rnp->qsmask != 0 || rcu_preempt_blocked_readers_cgp(rnp)` (line 2251 in `tree.c`). Tree-walking will then not stop until it reaches the root, resulting in too-short grace periods.

[1] https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.3.6.tar.xz, kernel/rcu.

B. Validating the RCU Model in User-Space

We ran our model in user space before formal analysis by CBMC. We performed 1000 runs for each bug scenario with a 60 s timeout to wait for the end of a grace period and a random delay of up to 1 s in the RCU reader task.

As expected, testing the model without bug injection always ran to completion successfully. Testing a weak form of liveness using `assert(0)`, as described in Sec. V-A, evidenced the end of a grace period by triggering an assertion violation in all runs. For Bug 1, an assertion violation was triggered in 559 out of 1000 runs. For Bugs 2–6, the user program timed out in all the runs, thus a grace period did not complete. For Bug 7 with one reader thread the testing harness failed to trigger an assertion violation. But we were able to observe a failure in 242 out of 1000 runs with two reader threads.

C. Getting CBMC to work on Tree RCU

Getting CBMC to work on our RCU model is non-trivial, owing to Tree RCU's complexity combined with CBMC's bit-precise verification. Early attempts resulted in very large SAT formulas. After the optimizations described below, the largest formula contained around 90 million variables and 450 million clauses, enabling CBMC to run to completion.

First, instead of placing the scheduling-clock interrupt in its own thread, we invoke functions `rcu_note_context_switch()` and `rcu_process_callbacks()` directly, as described in Sec. IV. Also, we invoke `__note_gp_changes()` from `rcu_gp_init()` to notify each CPU of a new grace period, instead of invoking `rcu_process_callbacks()`.

Second, support for linked lists in CBMC 5.4 is limited, resulting in unreachable code in CBMC's symbolic execution. So we stubbed all list-related code in our model, including that for callback handling.

Third, CBMC's structure-pointer and array encodings result in large formulas and long formula-generation times. Our focus on the RCU-sched flavor allowed us to eliminate the data structures of other flavors and trivialize the `for_each_rcu_flavor()` flavor-traversal loops. Our focus on small numbers of CPUs meant that RCU-sched's `rcu_node` tree contained only a root node, so we also trivialized the loops traversing this tree.

Fourth, CBMC unwinds every loop to the depth specified in its command line, even when the actual loop depth is smaller. This unnecessarily increases formula size. Since loops in our model can be decided at compile time, we used a command line option to state an unwinding depth for each loop.

Finally, since our test harness only requires one `rcu_node` structure and two `rcu_data` structures, we can use 32-bit encodings for `int`, `long`, and pointers. This reduces CBMC's formula size by half compared to the 64-bit default.

D. Verification Results and Discussion

Table I shows the results of our experiments using CBMC 5.4. Scenario Prove verifies our RCU model without bug injection

TABLE I: Experimental Results of CBMC

Scenario	#Const	#Variable	#Clause	Max VM	Solver Time	Total Time	Result
Prove	5.2m	30.0m	149.7m	23 GB	9h 24m	9h 36m	Safe
Prove-TSO	5.6m	42.0m	210.7m	34 GB	10h 51m	11h 4m	Safe
Prove-PSO	5.6m	41.3m	207.0m	34 GB	11h 23m	11h 36m	Safe
Prove-GP	5.4m	30.6m	152.7m	24 GB	3h 52m	4h 5m	GP Completed
Prove-GP-TSO	5.6m	42.0m	210.7m	34 GB	13h 1m	13h 14m	GP Completed
Prove-GP-PSO	5.6m	41.3m	207.0m	34 GB	8h 24m	8h 37m	GP Completed
Bug 1	1.3m	11.7m	56.0m	8 GB	31m	33m	Assertion Violated
Bug 1-TSO	1.5m	17.1m	83.3m	13 GB	53m	56m	Assertion Violated
Bug 1-PSO	1.5m	16.5m	80.4m	12 GB	46m	48m	Assertion Violated
Bug 2	5.2m	30.0m	149.6m	23 GB	4h 25m	4h 37m	GP Hung
Bug 2-TSO	5.6m	42.0m	210.5m	34 GB	9h 57m	10h 10m	GP Hung
Bug 2-PSO	5.6m	41.2m	206.9m	34 GB	8h 51m	9h 4m	GP Hung
Bug 3	6.3m	34.8m	174.1m	28 GB	7h 11m	7h 25m	GP Hung
Bug 3-TSO	6.8m	48.7m	245.1m	41 GB	19h 40m	19h 55m	GP Hung
Bug 3-PSO	6.7m	48.0m	241.2m	41 GB	19h 19m	19h 35m	GP Hung
Bug 4	4.8m	27.8m	138.1m	22 GB	4h 3m	4h 14m	GP Hung
Bug 4-TSO	5.1m	38.4m	192.6m	31 GB	8h 18m	8h 30m	GP Hung
Bug 4-PSO	5.1m	37.7m	188.9m	31 GB	8h 14m	8h 26m	GP Hung
Bug 5	5.1m	29.5m	146.7m	23 GB	4h 6m	4h 18m	GP Hung
Bug 5-TSO	5.5m	41.2m	206.5m	34 GB	5h 46m	5h 59m	GP Hung
Bug 5-PSO	5.4m	40.5m	202.9m	33 GB	5h 42m	5h 55m	GP Hung
Bug 6	1.4m	13.1m	63.3m	9 GB	19m	21m	GP Hung
Bug 6-TSO	1.5m	17.2m	84.1m	13 GB	1h 32m	1h 33m	GP Hung
Bug 6-PSO	1.5m	16.7m	81.4m	12 GB	1h 22m	1h 24m	GP Hung
Bug 7 (1R)	5.0m	29.2m	145.3m	23 GB	8h 48m	9h	**Safe (Bug Missed)**
Bug 7-TSO (1R)	5.2m	40.1m	200.8m	32 GB	11h 6m	11h 18m	Assertion Violated
Bug 7-PSO (1R)	5.1m	39.4m	197.2m	32 GB	11h 32m	11h 44m	Assertion Violated
Bug 7 (2R) *	15.1m	71.2m	359.0m	59 GB	19h 2m	19h 40m	Assertion Violated
Bug 7-TSO (2R) *	15.6m	90.4m	456.9m	75 GB	78h 12m	78h 53m	Assertion Violated
Bug 7-PSO (2R) *	15.6m	89.3m	451.6m	75 GB	84h 21m	85h 2m	**Out of Memory**

* Done on a 64-bit machine running Linux 3.19.8 with twelve Intel Xeon 2.40 GHz cores and 96 GB of main memory

over Sequential Consistency (SC). We also exercise the model over the weak memory models TSO and PSO, with Prove-TSO and Prove-PSO. Prove-GP performs the same reachability check as in Sec. V-B over SC. We perform the same reachability verification over TSO and PSO with Prove-GP-TSO and Prove-GP-PSO, respectively. Scenarios Bug 1–7 are the bug-injections in Sec. V-A and are verified over SC, TSO and PSO.

In our experiments, CBMC returned all the expected results except for Bug 7, where it failed to trigger the assertion `assert(r2 == 0 || r1 == 1)` with one RCU reader thread running over SC. This was due to approximation of the scheduling-clock interrupt by a direct function call, as described in Sec. IV. However, CBMC did report a violation of the assertion either when two RCU reader threads were present or when run over TSO or PSO. All of these cases decrease determinism, which in turn more faithfully model non-deterministic scheduling-clock interrupts, allowing the assertion to be violated.

CBMC took over 9 hours to verify our model over SC. The formulas for Prove-TSO and Prove-PSO are about 40% larger than for Prove. Although this verification consumed considerable memory and CPU, it verified all possible executions and reorderings permitted by TSO and PSO, a tiny subset of which are reached by the `rcutorture` test suite.

CBMC proved that grace periods can end over SC (Prove-GP), TSO (Prove-GP-TSO), and PSO (Prove-GP-PSO). The formula size and memory consumption are similar to those of the three Prove scenarios. It took CBMC about 4, 13, and 8.5 hours to find violations of `assert(0)` in Prove-GP, Prove-GP-TSO, and Prove-GP-PSO, respectively. For the bug-injection scenarios described in Sec. V-A, CBMC was able to return the expected results in all scenarios over SC except for Bug 7.

Figures 1–3 compare the formula size between SC, TSO and TSO. Table I also shows that runtime and memory overhead for TSO and PSO are quite similar, except for Bug 7. But TSO and PSO overhead significantly exceeds that of SC, with up to 340% (Bug 6 runtime) and 50% (Bug 1 memory) increases. The runtime was 5–19 hours and memory use exceeded 31 GB in all scenarios except Bug 1 and 6, owing to the large amount of code removed for these two scenarios. The numbers of variables and clauses are around 130% greater than for SC. The two-reader variant of Bug 7 has by far the longest runtime. It also used more than double the memory of the one-reader variant. For PSO, with two reader threads (marked '2R') CBMC's solver ran out of memory after 85 hours whereas with one reader it completed in less than 12 hours. The increased overhead is due to the additional RCU reader's call to `rcu_process_callbacks()`. This in turn results in more than a 125% increase in the number of constraints, variables, and clauses.

This work demonstrates the nascent ability and potential of SAT-based formal-verification tools to handle real-world production-quality synchronization primitives, as exemplified by Linux-kernel Tree RCU on weakly ordered TSO and PSO systems. Although modeling weak ordering incurs a significant performance penalty, this penalty is not excessive. It also confirms the tractability and practicality of the use of bug injection to validate both the model and the tools.

VI. Related Work

McKenney has applied the SPIN model checker to verify RCU's `NO_HZ_FULL_SYSIDLE` functionality [7], and interactions between dyntick-idle and non-maskable interrupts [9]. Desnoyers et al. [8] propose a virtual architecture to model out-of-order

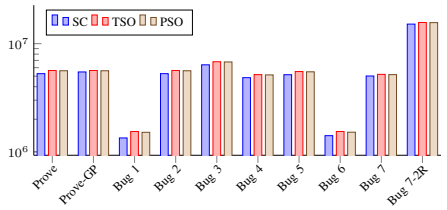

Fig. 1: Number of Constraints

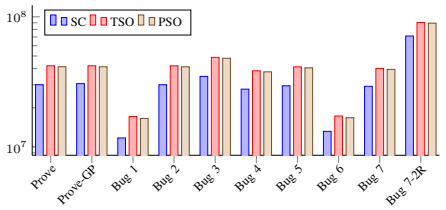

Fig. 2: Number of Variables

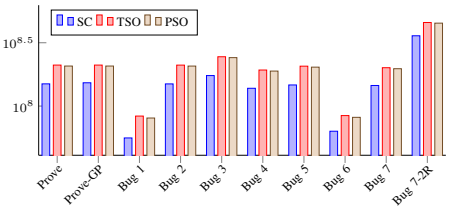

Fig. 3: Number of Clauses

memory accesses and instruction scheduling. User-level RCU [17] is modeled and verified in the proposed architecture using the SPIN model checker. These efforts require an error-prone manual translation from C to SPIN's modeling language, and therefore are not appropriate for regression testing. By contrast, our work constructs an RCU model directly from its source code from the Linux kernel.

McKenney has used CBMC to verify Tiny RCU [7], a trivial Linux-kernel RCU implementation for uni-core systems. Roy has applied the same tool to verify a significant portion of Sleepable RCU (SRCU). CBMC is now part of the regression test suite of SRCU in the Linux kernel [16].

Concurrently with our work, Kokologiannakis et al. verify Tree RCU using Nidhugg [18]. Since Nidhugg has better list support and does not model data non-determinism, they are able to verify more scenarios with less CPU and memory consumption. But some portions of RCU use atomic read-modify-write operations that can give nondeterministic results. So we hypothesize that data non-determinism will be required to verify RCU's dyntick-idle and `rcu_barrier()` components.

Groce et al. [19] introduce a falsification-driven verification methodology based on mutation testing. Using CBMC, they are able to find two holes in `rcutorture`, RCU's stress testing suite, one of which was hiding a real bug in Tiny RCU. Further work on real hardware has identified two more `rcutorture` holes; one was hiding a real bug in Tasks RCU [20] and the other was hiding a minor performance bug in Tree RCU.

Gotsman et al. [11] use an extended concurrent separation logic to formalize grace periods and prove an abstract implementation of RCU over SC. Tassarotti et al. [12] use the GPS program logic to verify a simple implementation of user-level RCU for a singly-linked list. They assume the "release-acquire" semantics, which is weaker than SC but stronger than memory models used by real-world RCU implementations. These proofs are done manually on simple implementations of RCU.

VII. Conclusion

This paper shows how to use the CBMC model checker to verify a significant part of the Tree RCU implementation automatically, which to the best of our knowledge is unprecedented. This work shows that RCU is a rich example to drive research: it is small enough to provide models that can just barely be verified by existing tools, but it also has enough concurrency and complexity to drive advances in techniques and tooling.

For future work, we plan to verify safety and liveness of quiescent-state forcing and grace-period expediting, using more sophisticated test harnesses that pass through multiple grace periods. We also plan to model and verify the preemptible version of Tree RCU, which we expect to be quite challenging.

There are also potential improvements to CBMC to better support RCU verification. For instance, better list support is needed to verify RCU's callback mechanism. A field-sensitive SSA encoding for structures and a thread-aware slicer will help reduce encoding size and so improve scalability.

References

[1] "The Linux kernel," https://www.kernel.org/.
[2] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *PDCS*, 1998.
[3] P. E. McKenney and J. Walpole, "Introducing technology into the Linux kernel: a case study," *ACM OSR*, vol. 42, no. 5, 2008.
[4] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole, "RCU usage in the Linux kernel: One decade later," 2013, technical Report.
[5] "Where is the Internet?" http://www.whoishostingthis.com/blog/2013/12/06/internet-infographic/, 2013.
[6] J. Callaham, "Google says there are now 1.4 billion active Android devices worldwide," http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide, 2015.
[7] P. E. McKenney, "Verification challenges," 2017, http://paulmck.livejournal.com/tag/verification%20challenge.
[8] M. Desnoyers, P. E. McKenney, and M. R. Dagenais, "Multi-core systems modeling for formal verification of parallel algorithms," *ACM OSR*, vol. 47, no. 2, 2013.
[9] P. E. McKenney, "Integrating and validating dynticks and preemptable RCU," 2008, https://lwn.net/Articles/279077/.
[10] G. J. Holzmann, "The model checker SPIN," *IEEE TSE*, vol. 23, no. 5, 1997.
[11] A. Gotsman, N. Rinetzky, and H. Yang, "Verifying concurrent memory reclamation algorithms with grace," in *ESOP*, 2013.
[12] J. Tassarotti, D. Dreyer, and V. Vafeiadis, "Verifying read-copy-update in a logic for weak memory," in *PLDI*, 2015.
[13] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, 2004.
[14] L. Liang, P. E. McKenney, D. Kroening, and T. Melham, "Verification of the tree-based hierarchical read-copy update in the Linux kernel," *CoRR*, vol. abs/1610.03052, 2016. [Online]. Available: http://arxiv.org/abs/1610.03052
[15] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient bounded model checking of concurrent software," in *CAV*, 2013.
[16] L. Roy, "rcutorture: Add CBMC-based formal verification for SRCU," 2017, https://www.spinics.net/lists/kernel/msg2421833.html.
[17] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE TPDS*, vol. 23, no. 2, 2012.
[18] M. Kokologiannakis and K. Sagonas, "Stateless model checking of the Linux kernel's hierarchical Read-Copy-Update (Tree RCU)," in *SPIN*, 2017.
[19] A. Groce, I. Ahmed, C. Jensen, and P. E. McKenney, "How verified is my code? Falsification-driven verification," in *ASE*, 2015.
[20] J. Corbet, "The RCU-tasks subsystem," 2014, http://lwn.net/Articles/607117/.