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

Lihao Liang
University of Oxford
lihao.liang@cs.ox.ac.uk

Paul E. McKenney
Linux Technology Center, IBM
paulmck@linux.vnet.ibm.com

Daniel Kroening
University of Oxford
daniel.kroening@cs.ox.ac.uk

Tom Melham
University of Oxford
tom.melham@cs.ox.ac.uk

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 for multi-core systems are decidedly non-trivial. Giving the ubiquity of Linux, a rare "million-year" bug can occur several times per day across the installed base. Stringent validation of RCU's complex behaviors is thus critically important. Exhaustive testing is infeasible due to the exponential number of possible executions, which suggests use of formal verification.

Previous verification efforts on RCU either focus on simple implementations or use modeling languages, the latter requiring error-prone manual translation that must be repeated frequently due to regular changes in the Linux kernel's RCU implementation. In this paper, we first describe the implementation of Tree RCU in the Linux kernel. We then discuss how to construct a model directly from Tree RCU's source code in C, and use the CBMC model checker to verify its safety and liveness properties. To our best knowledge, this is the first verification of a significant part of RCU's source code, and is an important step towards integration of formal verification into the Linux kernel's regression test suite.

Categories and Subject Descriptors [*D.2.4*]: Software/Program Verification—Model checking; [*D.1.3*]: Concurrent Programming—Parallel programming

Keywords Software Verification, Parallel Computing, Read-Copy Update, Linux Kernel

1. Introduction

The Linux operating system kernel [**?**] is widely used in a variety of computing platforms, including servers, safety-critical embedded systems, household appliances, and mobile devices such as smartphones. Over the past 25 years, many technologies have been added to the Linux kernel, one example being Read-Copy Update (RCU) [**?**].

RCU is a synchronization mechanism that can be used to replace reader-writer locks in read-mostly scenarios. It allows low-overhead readers to run concurrently with updaters. Production-quality RCU 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. The implementation 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, the implementation must cope with the extremely diverse workloads and platforms of Linux [**?**].

RCU is now widely used in the Linux-kernel networking, device-driver, and file-storage subsystems [**? ?**]. To date, there are at least 75 million Linux servers [**?**] and 1.4 billion Android devices [**?**], which means that 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.

Most validation efforts for concurrent software rely on testing, but unfortunately there is no cost-effective test strategy that can cover all corner cases. Worse still, some of errors that testing does detect might be difficult to reproduce, diagnose, and repair. The concurrent nature of RCU and the sheer size of the search space suggest use of formal verification, particularly model checking [**?**].

Formal verification has already been applied to some aspects of RCU design, including Tiny RCU [**?**], userspace RCU [**?**], sysidle [**?**], and interactions between dyntick-idle and non-maskable interrupts (NMIs) [**?**]. But these efforts either validate trivial single-CPU RCU implementations in C (Tiny RCU), or use special-purpose languages such as Promela [**?**]. Although special-purpose modeling languages do have advantages, a major disadvantage in the context of the Linux kernel is the difficult and error-prone translation from source code. Other researchers have applied manual proofs in formal logics to simple RCU implementations [**? ?**]. These proofs are quite admirable, but require even more manual work, in addition to the translation effort.

Worse yet, Linux kernel releases are only about 60 days apart, and RCU changes with each release. Thus, any manual work must be replicated about six times a year so that mechanical and manual models or proofs remain synchronized with the Linux-kernel RCU implementation. Therefore, if formal verification is to be part of Linux-kernel RCU's regression suite, the verification methods must be scalable and automated. To this end, this paper describes how to build a model directly from the Linux kernel source code, and use the C Bounded Model Checker (CBMC) [**?**] to verify RCU's safety and liveness properties. To the best of our knowledge, this is the first automatic verification of a significant part of the Linux-kernel RCU source code.

2. Background

2.1 What is RCU?

Read-copy update (RCU) is a synchronization mechanism that is often used to replace reader-writer locking. RCU readers run concurrently with updaters, and so RCU avoids read-side contention by maintaining multiple versions of objects and ensuring they are not freed until all pre-existing readers complete, that is, until after a *grace period* elapses. The basic idea is to split updates into removal and reclamation phases that are separated by a grace period [**?**]. The removal phase removes reader-accessible references to objects, perhaps by replacing them with new versions.

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

As illustrated in Figure 1, grace periods are only needed for those readers whose runtime overlaps the removal phase. Those that start after the removal phase cannot hold references to the removed objects and thus cannot be disrupted by objects being freed during the reclamation phase.

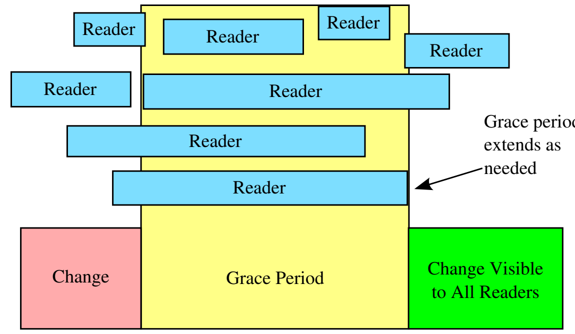

Figure 1: RCU Concepts

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

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

void rcu_updater(void) {
  x = 1;
  synchronize_rcu();
  y = 1;
}

...

// after both rcu_reader()
// and rcu_updater() return
assert(r2 == 0 || r1 == 1);
```

Figure 2: Verifying RCU Grace Periods

2.2 Core RCU API Usage

The core RCU API is quite small and consists of only five primitives: `rcu_read_lock()`, `rcu_read_unlock()`, `synchronize_rcu()`, `rcu_assign_pointer()`, and `rcu_dereference()` [**?**].

An RCU read-side critical section begins with `rcu_read_lock()` and ends with a corresponding `rcu_read_unlock()`. When nested, they are flattened into one large critical section. Within a critical section, it is illegal to block, but preemption is legal in a preemptible kernel. RCU-protected data accessed by a read-side critical section will not be reclaimed until after that critical section completes.

The function `synchronize_rcu()` marks the end of the updater code and the beginning of the reclaimer code. It blocks until all pre-existing RCU read-side critical sections have completed. Note that `synchronize_rcu()` does not necessarily wait for critical sections that begin after it does.

Consider the example given in Figure 2. If the RCU read-side critical section in function `rcu_reader()` begins before `synchronize_rcu()` in `rcu_updater()` is called, then it must finish before `synchronize_rcu()` returns, so that the value of `r2` must be 0. If it ends after `synchronize_rcu()` returns, then the value of `r1` must be 1.

Finally, to assign a new value to an RCU-protected pointer, RCU updaters use `rcu_assign_pointer()`, which returns the new value. RCU readers can use `rcu_dereference()` to fetch an RCU-protected pointer, which can then be safely dereferenced. The returned value is only valid within the enclosing RCU read-side critical section. The `rcu_assign_pointer()` and `rcu_dereference()` functions work together to ensure that if a given reader dereferences an RCU-protected pointer to a just-inserted object, the dereference operation will return valid data rather than pre-initialization garbage.

3. Implementation of Tree RCU

The primary advantage of RCU is that it is able to wait for an arbitrarily large number of readers to finish without keeping track every single one of them. The number of readers can be large (up to the number of CPUs in non-preemptible implementations and up to the number of tasks in preemptible implementations). Although RCU's read-side primitives enjoy excellent performance and scalability, update-side primitives must defer the reclamation phase till all pre-existing readers have completed, either by blocking or by registering a callback that is invoked after a grace period. The performance and scalability of RCU relies on efficient mechanisms to detect when a grace period has completed. For example, a simplistic RCU implementation might require each CPU to acquire a global lock during each grace period, but this would severely limit performance and scalability. Such an implementation would be quite unlikely to scale beyond a few hundred CPUs. This is woefully insufficient because Linux runs on systems with thousands of CPUs. This has motivated the creation of Tree RCU.

3.1 Overview

We focus on the "vanilla" RCU API in a non-preemptible build of the Linux kernel, specifically on the `rcu_read_lock()`, `rcu_read_unlock()`, and `synchronize_rcu()` primitives. The key idea is that RCU read-side primitives are confined to kernel code and, in non-preemptible implementations, do not block. Thus, when a CPU is blocking, in the idle loop, or running in user mode, all RCU read-side critical sections that were previously running on that CPU must have finished. Each of these states is therefore called a *quiescent state*. After each CPU has passed through a quiescent state, the corresponding RCU grace period ends. The key challenge is to determine when all necessary quiescent states have completed for a given grace period—and to do so with excellent performance and scalability.

For example, if RCU used a single data structure to record each CPU's quiescent states, the result would be extreme lock contention on large systems, in turn resulting in poor performance and abysmal scalability. Tree RCU therefore instead uses a tree hierarchy of data structures, each leaf of which records quiescent states of a single CPU and propagates the information up to the root. When the root is reached, a grace period has ended. Then the grace-period information is propagated down from the root to the leaves of the tree. Shortly after the leaf data structure of a CPU receives this information, `synchronize_rcu()` will return.

In the remainder of this section, we discuss the implementation of the non-preemptible Tree RCU in the Linux kernel version 4.3.6. We first briefly discuss the implementation of read/write-side primitives. We then explain Tree RCU's hierarchical data structure which records quiescent states while maintaining bounded lock contention. Finally, we discuss how RCU uses this data structure to detect quiescent states and grace periods without individually tracking readers.

3.2 Read/Write-Side Primitives

In a non-preemptible kernel, any region of kernel code that does not voluntarily block is implicitly an RCU read-side critical section. Therefore, the implementations of `rcu_read_lock()` and `rcu_read_unlock()` need do nothing at all, and in fact in production kernel builds that do not have debugging enabled, these two primitives have absolutely no effect on code generation.

In the common case where there are multiple CPUs running, the update-side primitive `synchronize_rcu()` calls `wait_rcu_gp()`, which is an internal function that uses a callback mechanism to invoke `wakeme_after_rcu()` at the end of some later grace period. As its name suggests, `wakeme_after_rcu()` function wakes up `wait_rcu_gp()`, which returns, in turn allowing `synchronize_rcu()` to return control to its caller.

3.3 Data Structures of Tree RCU

RCU's global state is recorded in the `rcu_state` structure, which consists of a tree of `rcu_node` structures with a child count of up to 64 (32 in a 32-bit system). Every leaf node can have at most 64 `rcu_data` structures (again 32 on a 32-bit system), each representing a single CPU, as illustrated in Figure 3. Each `rcu_data` structure records its CPU's quiescent states, and the `rcu_node` tree propagates these states up to the root, and then propagates grace-period information back down to the leaves. Quiescent-state information does not propagate upwards from a given node until a quiescent state has been reported by each CPU covered by the subtree headed by that node. This propagation scheme dramatically reduces the lock contention experienced by the upper levels of the tree. For example, consider a default `rcu_node` tree for a 4,096-CPU system, which will have have 256 leaf nodes, four internal nodes, and one root node. Dur-

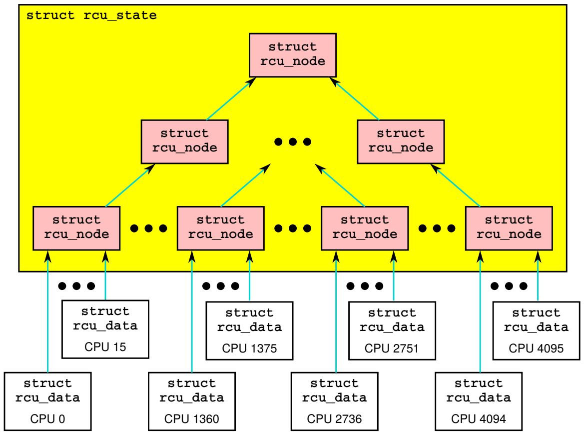

Figure 3: Tree RCU Hierarchy

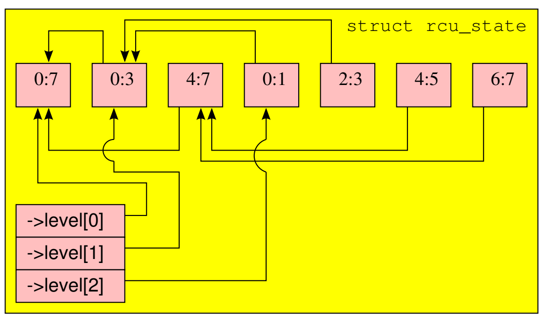

Figure 4: Array Representation for a Tree of `rcu_node` Structures

ing a given grace period, each CPU will report its quiescent states to its leaf node, but there will only be 16 CPUs contending for each of those 256 leaf nodes. Only 256 of the CPUs will report quiescent states to the internal nodes, with only 64 CPUs contending for each of the four internal nodes. Only four CPUs will report quiescent states to the root node, resulting in extremely low contention on the root node's lock, so that contention on any given `rcu_node` structure is sharply bounded even in very large configurations. The current RCU implementation in the Linux kernel supports up to a four-level tree, and thus in total $64^4 = 16,777,216$ CPUs in a 64 bit machine.[1]

3.3.1 `rcu_state` Structure

Each flavor of RCU has its own global `rcu_state` structure. The `rcu_state` structure includes a array of `rcu_node` structures organized as a tree `struct rcu_node node[NUM_RCU_NODES]`, with `rcu_data` structures connected to the leaves. Given this organization, a breadth-first traversal is simply a linear scan of the array. Another array `struct rcu_node *level[NUM_RCU_LVLS]` is used to point to the left-most node at each level of the tree, as shown in Figure 4.

The `rcu_state` structure uses `unsigned long` fields `->gpnum` and `->completed` to track RCU's grace periods. The `->gpnum` field records the most recently started grace period, whereas `->completed` records the most recently ended grace period. If the two numbers are equal, then corresponding flavor of RCU is idle. If `gpnum` is one greater than `completed`, then RCU is in the middle of a grace period. All other combinations are invalid.

3.3.2 `rcu_node` Structure

The tree of `rcu_node` structures records and propagates quiescent-state information from the leaves to the root, and also propagates grace-period information from the root to the leaves. The `rcu_node` structure has a spinlock `->lock` to protect its fields. The `->parent` field references the parent `rcu_node` structure, and is `NULL` for the root. The `->level` field indicates the level in the tree, counting from zero at the root. The `->grpmask` field identifies this node's bit in the `->qsmask` field of its parent. The `->grplo` and `->grphi` fields indicates the lowest and highest numbered CPU that are covered by this `rcu_node` structure, respectively.

The `->qsmask` field indicates which of this node's children still need to report quiescent states for the current grace period. As with `rcu_state`, the `rcu_node` structure has `->gpnum` and `->completed` fields that have values identical to those of the enclosing `rcu_state` structure, except at the beginnings and ends of grace periods when the new values are propagated down the tree. Each of these fields can be smaller than its `rcu_state` counterpart by at most one.

3.3.3 `rcu_data` structure

The `rcu_data` structure detects quiescent states and handles RCU callbacks for the corresponding CPU. The structure is accessed primarily from the corresponding CPU, thus avoiding synchronization overhead. As with the `rcu_state` structure, different flavors of RCU maintain their own per-CPU `rcu_data` structures. The `->cpu` field identifies the corresponding CPU, the `->rsp` field references the corresponding `rcu_state` structure, and the `->mynode` field references the corresponding leaf `rcu_node` structure. The `->grpmask` field identifies this `rcu_data` structure's bit in the `->qsmask` field of its leaf `rcu_node` structure.

The `rcu_data` structure's `->qs_pending` field indicates that RCU needs a quiescent state from the corresponding CPU, and the `->passed_quiesce` indicates that the CPU has already passed through a quiescent state. The `rcu_data` also has `->gpnum` and `->completed` fields, which can lag arbitrarily behind their counterparts in the `rcu_state` and `rcu_node` structures on idle CPUs. However, on the non-idle CPUs that are the focus of this paper, they can lag at most one grace period behind their leaf `rcu_node` counterparts.

The `rcu_state` structure's `->gpnum` and `->completed` fields represent the most current values, and are tracked

[1] Four-level trees are only used in stress testing, but three-level trees are used in production by 4096-CPU systems.

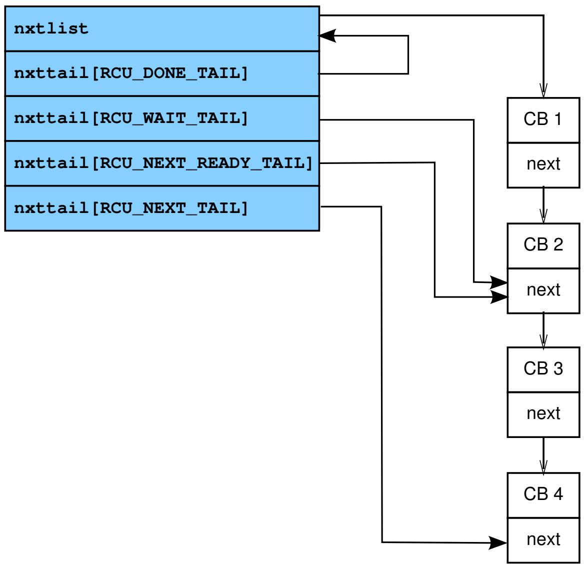

Figure 5: Callback Queuing in `rcu_data`

closely by those of the `rcu_node` structure, which allows the `->gpnum` and `->completed` fields in the `rcu_data` structures to be are compared against their counterparts in the corresponding leaf `rcu_node` to detect a new grace period. This scheme allows CPUs to detect beginnings and ends of grace periods without incurring lock- or memory-contention penalties. The `rcu_data` structure manages RCU callbacks using a four-segment list [?].

3.3.4 RCU Callbacks

The `rcu_data` structure manages RCU callbacks using a `->nxtlist` pointer tracking the head of the list and an array of `->nxttail[]` tail pointers that form a four-segment list of callbacks [?], with each element of the `->nxttail[]` array referencing the tail of the corresponding segment, as shown in Figure 5. The segment ending with `->nxttail[RCU_DONE_TAIL]` (the "`RCU_DONE_TAIL` segment") contains callbacks handled by a prior grace period that are therefore ready to be invoked. The `RCU_WAIT_TAIL` and `RCU_NEXT_READY_TAIL` segments contain callbacks waiting for the current and the next grace period, respectively. Finally, the `RCU_NEXT_TAIL` segment contains callbacks that are not yet associated with any grace period. The `->qlen` field counts the total number of callbacks, and the `->blimit` field specifies the maximum number of RCU callbacks that may be invoked at a given time, thus limiting response-time degradation due to long lists of callbacks.[2]

Back in Figure 5, the `->nxttail[RCU_DONE_TAIL]` array element references `->nxtlist`, which means none of the callbacks are ready to invoke. The `->nxttail[RCU_WAIT_TAIL]` element references callback 2's `->next` pointer, meaning that callbacks CB 1 and CB 2 are waiting for the current grace period. The `->nxttail[RCU_NEXT_READY_TAIL]` element references that same `->next` pointer, meaning that no callbacks are waiting for the next grace period. Finally, the callbacks between the `->nxttail[RCU_NEXT_READY_TAIL]` and `->nxttail[RCU_NEXT_TAIL]` elements (CB 3 and CB 4) are not yet assigned to a specific grace period. The `->nxttail[RCU_NEXT_TAIL]` element always references either the last callback or, when the entire list is empty, `->nxtlist`.

Cache locality is promoted by invoking callbacks on the CPU that registered them. For example, RCU's update-side primitive `synchronize_rcu()` appends callback `wakeme_after_rcu()` to the end of the `->nxttail[RCU_NEXT_TAIL]` list in the current CPU (Section **??**). They are advanced one segment towards the head of the list (via `rcu_advance_cbs()`) when the CPU detects the current grace period has ended, which is indicated by the `->completed` field of the CPU's `rcu_data` structure being one smaller than its counterpart in the corresponding leaf `rcu_node` structure. The CPU also periodically merges the `RCU_NEXT_TAIL` segment into the `RCU_NEXT_READY_TAIL` segment by calling `rcu_accelerate_cbs()`. In a few special cases, the CPU merges the `RCU_NEXT_TAIL` segment into the `RCU_WAIT_TAIL` segment, bypassing the `RCU_NEXT_TAIL` segment. This optimization applies when the CPU is starting a new grace period. It does *not* apply when a CPU notices a new grace period because that grace period might well have started before the callbacks were added to the `RCU_NEXT_TAIL` segment. This is a deliberate design choice: It is more important for the CPUs to operate independently (thus avoiding contention and synchronization overhead) than it is to decrease grace-period latencies. In those rare occasions where low grace-period latency is important, the `synchronize_rcu_expedited()` should be used. This function has the same semantics as does `synchronize_rcu()`, but trades off efficiency optimizations in favor of reduced latency.

Each RCU callbacks is an `rcu_head` structure which has a `->next` field that points to the next callback on the list and a `->func` field that references the function to be invoked at the end of an upcoming grace period.

3.4 Quiescent State Detection

RCU has to wait until all pre-existing read-side critical sections have finished before it can safely allow a grace period to end. The performance and scalability of RCU rely on its ability to efficiently detect quiescent states and determine whether the set of quiescent states detected thus far allows the grace period to end. If each CPU (or, in the case of preemptible RCU, each task) has passed through a quiescent state, a grace period has elapsed.

The non-preemptible RCU-sched flavor's quiescent states apply to CPUs, and are user-space execution, context switch, idle, and offline state. Therefore, RCU-sched only needs to

[2] Workloads requiring aggressive real-time guarantees should use callback offloading, which is outside of the scope of this paper.

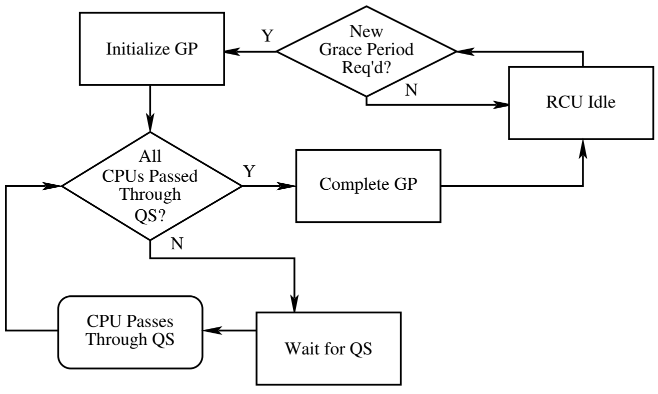

Figure 6: Grace-Period Detection State Diagram

track tasks and interrupt handlers that are actually running because blocked and preempted tasks are always in quiescent states. Thus, RCU-sched needs only track CPU states.

3.4.1 Scheduling-Clock Interrupt

The `rcu_check_callbacks()` is invoked from the scheduling-clock interrupt handler, which allows RCU to periodically check whether a given busy CPU is in the user-mode or idle-loop quiescent states. If the CPU is in one of these quiescent states, `rcu_check_callbacks()` invokes `rcu_sched_qs()`, which sets the per-CPU `rcu_sched_data.passed_quiesce` fields to 1.

The `rcu_check_callbacks()` function invokes `rcu_pending()` to determine whether a recent event or current condition means that RCU requires attention from this CPU. If so, `rcu_check_callbacks()` invokes `raise_softirq()`, which will cause `rcu_process_callbacks()` to be invoked once the CPU reaches a state where it is safe to do so (roughly speaking, once the CPU has interrupts, preemption, and bottom halves enabled). This function is discussed in detail in Section 3.5.

3.4.2 Context-Switch Handling

The context-switch quiescent state is recorded by invoking `rcu_note_context_switch()` from `__schedule()` (and, for the benefit of virtualization, also from `rcu_virt_note_context_switch()`). The `rcu_note_context_switch()` function invokes `rcu_sched_qs()` to inform RCU of the context switch, which is a quiescent state of the CPU.

3.5 Grace Period Detection

Once each CPU has passed through a quiescent state, a grace period for RCU has completed. As discussed in Section 3.3, Tree-RCU uses a hierarchy of `rcu_node` structures to manage quiescent state and grace period information. Quiescent-state information is passed up the tree from the leaf per-CPU `rcu_data` structures. Grace-period information is passed down from the root. We focus on grace-period detection for busy CPUs, as illustrated in Figure 6.

3.5.1 Softirq Handler for RCU

RCU's busy-CPU grace period detection relies on the `RCU_SOFTIRQ` handler function `rcu_process_callbacks()`, which is scheduled from the scheduling-clock interrupt. This function first calls `rcu_check_quiescent_state()` to report recent quiescent states on the current CPU. Then `rcu_process_callbacks()` starts a new grace period if needed, and finally calls `invoke_rcu_callbacks()` to invoke any callbacks whose grace period has already elapsed.

Function `rcu_check_quiescent_state()` first invokes `note_gp_changes()` to update the CPU-local `rcu_data` structure to record the end of previous grace periods and the beginning of new grace periods. Any new values for these fields are copied from the leaf `rcu_node` structure to the `rcu_data` structure. If an old grace period has ended, `rcu_advance_cbs()` is invoked to advance all callbacks, otherwise, `rcu_accelerate_cbs()` is invoked to assign a grace period to any recently arrived callbacks. If a new grace period has started, `->passed_quiesce` is set to zero, and if in addition RCU is waiting for a quiescent state from this CPU, `->qs_pending` is set to one, so that a new quiescent state will be detected for the new grace period.

Next, `rcu_check_quiescent_state()` checks whether `->qs_pending` indicates that RCU needs a quiescent state from this CPU. If so, it checks whether `->passed_quiesce` indicates that this CPU has in fact passed through a quiescent state. If so, it invokes `rcu_report_qs_rdp()` to report that quiescent state up the combining tree.

The `rcu_report_qs_rdp()` function first verifies that the CPU has in fact detected a legitimate quiescent state for the current grace period, and under the protection of the leaf `rcu_node` structure's `->lock`. If not, it resets quiescent-state detection and returns, thus ignoring any redundant quiescent states belonging to some earlier grace period. Otherwise, if the `->qsmask` field indicates that RCU needs to report a quiescent state from this CPU, `rcu_accelerate_cbs()` is invoked to assign a grace-period number to any new callbacks, and then `rcu_report_qs_rnp()` is invoked to report the quiescent state to the `rcu_node` combining tree.

The `rcu_report_qs_rnp()` function traverses up the `rcu_node` tree, at each level holding the `rcu_node` structure's `->lock`. At any level, if the child structure's `->qsmask` bit is already clear, or if the `->gpnum` changes, traversal stops. Otherwise, the child structure's bit is cleared from `->qsmask`, after which, if `->qsmask` is non-zero, traversal stops. Otherwise, traversal proceeds on to the parent `rcu_node` structure. Once the root is reached, traversal stops and `rcu_report_qs_rsp()` is invoked to awaken the grace-period kthread (kernel thread). The grace-period kthread will then clean up after the now-ended grace period, and, if needed, start a new one.

3.5.2 Grace-Period Kernel Thread

The RCU grace-period kthread invokes `rcu_gp_kthread()`, which contains an infinite loop that initializes, waits for, and cleans up after each grace period.

When no grace period is required, the grace-period kthread sets its `rcu_state` structure's `->flags` field to `RCU_GP_WAIT_GPS`, and then waits within an inner infinite loop for that structure's `->gp_state` field to be set. Once set, `rcu_gp_kthread()` invokes `rcu_gp_init()` to initialize a new grace period, which rechecks the `->gp_state` field under the root `rcu_node` structure's `->lock`. If the field is no longer set, `rcu_gp_init()` returns zero. Otherwise, it increments `rsp->gpnum` by 1 to record a new grace period number. Finally, it performs a breadth-first traversal of the `rcu_node` structures in the combining tree. For each `rcu_node` structure `rnp`, we set the `rnp->qsmask` to indicate which children must report quiescent states for the new grace period (Section 3.3.2), and set `rnp->gpnum` and `rnp->completed` to their `rcu_state` counterparts. If the `rcu_node` structure `rnp` is the parent of the current CPU's `rcu_data`, we invoke `__note_gp_changes()` to set up the CPU-local `rcu_data` state. Other CPUs will invoke `__note_gp_changes()` after their next scheduling-clock interrupt.

To clean up after a grace period, `rcu_gp_kthread()` calls `rcu_gp_cleanup()` after setting the `rcu_state` field `rsp->gp_state` to `RCU_GP_CLEANUP`. After the function returns, `rsp->gp_state` is set to `RCU_GP_CLEANED` to record the end of the old grace period. Function `rcu_gp_cleanup()` performs a breadth-first traversal of `rcu_node` combining-tree. It first sets each `rcu_node` structure's `->completed` field to the `rcu_state` structure's `->gpnum` field. It then updates the current CPU's CPU-local `rcu_data` structure by calling `__note_gp_changes()`. For other CPUs, the update will take place when they handle the scheduling-clock interrupts, in a fashion similar to `rcu_gp_init()`. After the traversal, it marks the completion of the grace period by setting the `rcu_state` structure's `->completed` field to that structure's `->gpnum` field, and invokes `rcu_advance_cbs()` to advance callbacks. Finally, if another grace period is needed, we set `rsp->gp_flags` to `RCU_GP_FLAG_INIT`. Then in the next iteration of the outer loop, the grace-period kthread will initialize a new grace period as discussed above.

4. Verification Scenario

We use the example in Figure 2 to demonstrate how the different components of Tree RCU work together to guarantee that all pre-existing read-side critical sections finish before RCU allows a grace period to end. This example will drive the verification, which will check for violations of the assertion at this end of the code.

We focus on the implementation of the non-preemptible RCU-sched flavor. We further assume there are only two CPUs, and that CPU 0 executes function `rcu_reader()` and CPU 1 executes `rcu_updater()`. When the system boots, the Linux kernel calls `rcu_init()` to initialize RCU, which includes constructing the combining tree of `rcu_node` and `rcu_data` structures via `rcu_init_geometry()` and initializing the fields of the nodes in the tree for each RCU flavor via `rcu_init_one()`. In our example it will be a one-level tree that has one `rcu_node` structure as root and two children that are `rcu_data` structures for each CPU. Function `rcu_spawn_gp_kthread()` is also called to initialize and spawn the RCU grace-period kthread for each RCU flavor.

Referring again to Figure 2, suppose that `rcu_reader()` begins execution on CPU 0 while `rcu_updater()` concurrently sets `x` to 1 and then invokes `synchronize_rcu()` on CPU 1. As discussed in Section 3.2, `synchronize_rcu()` invokes `wait_rcu_gp()`, which in turn registers an RCU callback that will invoke `wakeme_after_rcu()` some time after `rcu_reader()` exits its critical section.

However, this critical-section exit has no immediate effect. Instead, a later context switch will invoke `rcu_note_context_switch()`, which in turn invokes `rcu_sched_qs()`, recording the quiescent state in the CPU's `rcu_sched_data` structure's `->passed_quiesce` field. Later, a scheduling-clock interrupt will invoke `rcu_check_callbacks()`, which calls `rcu_pending()` and notes that the `->passed_quiesce` field is set. This will cause `rcu_pending()` to return `true`, which in turn causes `rcu_check_callbacks()` to invoke `rcu_process_callbacks()`. In its turn, `rcu_process_callbacks()` will invoke `raise_softirq(RCU_SOFTIRQ)`, which, once the CPU has interrupts, preemption, and bottom halves enabled, calls `rcu_process_callbacks()`.

As discussed in Section 3.5.1, 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). Then it checks whether the CPU 0 has passed a quiescent state. Since a quiescent state has been recorded for CPU 0, `rcu_report_qs_rnp()` is invoked to traversal up the combining tree. It clears the first bit of the root `rcu_node` structure's `qsmask` field (recall that the RCU combining tree has only one level). 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 `->qs_mask` 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 kthread, which will clean up the ended grace period, and, if needed, start a new one (Section 3.5.2).

Lastly, `rcu_process_callbacks()` calls `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.

5. Modeling RCU for CBMC

The C Bounded Model Checker (CBMC)[3] is a program analyzer that implements bit-precise bounded model checking for C programs [?]. CBMC can demonstrate violation of assertions in C programs, or prove their safety under a given loop unwinding bound. It translates an input C program into a formula, which is then passed to a modern SAT or SMT solver together with a constraint that specifies the set of error states. If the solver determines the formula to be satisfiable, an error trace giving the exact sequence of events is extracted from the satisfying assignment. Recently, support has been added for verifying concurrent programs over a wide range of memory models, including SC, TSO, and PSO [?].

In the remainder of this section we describe how to construct a model from the source code of the Tree RCU implementation in the Linux kernel version 4.3.6, which can be verified by 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 adding preprocessor directives to conditionally inject bugs (described in Section 6.1). The Linux kernel environment and the majority of these changes to the source code are made through macros in separate files that can be reused across different versions of the Tree RCU implementation. The biggest change in the source files is to use arrays to model per-CPU data, which could potentially be scripted. The resulting model is C code with assertions that can be also run as a user program, which provides important validation of the model itself.

Initialization

Our model first invokes `rcu_init()` which in turn invokes: (1) `rcu_init_geometry()` to compute the `rcu_node` tree geometry; (2) `rcu_init_one` to initialize the `rcu_state` structure; (3) `rcu_cpu_notify()` to initialize each CPU's `rcu_data` structure. This boot initialization tunes the data-structure configuration to match that of the specific hardware at hand. For example, a large-system tree might resemble Figure 3, while a small configuration has a single `rcu_node` "tree". The model then calls `rcu_spawn_gp_kthread()` to spawn the grace-period kthreads discussed below.

Per-CPU Variables and State

RCU uses per-CPU data to provide cache locality and to reduce contention and synchronization overhead. For example, the per-CPU structure `rcu_data` records quiescent states and handles RCU callbacks (Section 3.3.3). We model this per-CPU 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 `cpu_lock`, 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 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 `synchronize_sched()`

Because our model omits CPU hotplug and callback handling, 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`, which is initialized to 1 in `wait_rcu_gp()` before the grace period. Because `wait_rcu_gp()` blocks, it can result in a context switch, 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 combining tree, the grace-period kthread 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. The `__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

The `rcu_check_callbacks()` function detects idle execution, usermode execution, and to invoke RCU core processing in response to state changes. Because we model neither idle nor usermode execution, the only state changes are quiescent states and the beginnings and ends of grace periods. We therefore dispense with `rcu_check_callbacks()` (Section 3.5.1). 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 Kthread

As discussed in Section 3.5.2, `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 generated by CBMC, instead of spawning a separate thread, we directly call `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()`.

[3] `http://www.cprover.org/cbmc/`

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 components of Tree RCU, excluding, for example, quiescent-state forcing, grace-period expediting, and callback handling. In addition, we make the assumption that all CPUs are busy executing RCU related tasks. As a result, we do not model the following scenarios: 1. CPU hotplug and dyntick-idle; 2. Thread-migration failure modes in the Linux kernel involving per-CPU variables; 3. RCU priority boosting. Moreover, we model scheduling-clock interrupts as direct function calls, which, as discussed later, results in failures to model one of the bug-injection scenarios. Lastly, the test harness we use only passes through a single grace period, so cannot detect failures involving multiple grace periods.

6. Experiments

In this section we discuss our experiments verifying the Linux-kernel Tree RCU implementation. We first describe several bug-injection scenarios used in the experiments. Next, we report results of user-space runs of the RCU model. Then we describe how verify our RCU model using CBMC. Finally, we discuss the experimental results. We performed our experiments on a 64-bit machine running Linux 3.19.8 with eight Intel Xeon 3.07 GHz cores and 48 GB of memory.

6.1 Bug-Injection Scenarios

Because we model non-preemptible Tree RCU, each CPU runs exactly one RCU task as a separate thread. Upon 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 uses the example in Figure 2, including its assertion `assert(r2 == 0 || r1 == 1)`. This assertion does not hold when RCU's fundamental safety guarantee is violated: read-side critical sections cannot span grace periods [**?**]. We also verify a *weak form* of liveness by inserting an `assert(0)` after the `__CPROVER_assume(thread_cnt == 2)` statement. This assertion cannot hold, and so it will be violated if at least one grace period completes. Such a "verification failure" is in fact the expected behavior for a correct RCU implementation. On the other hand, if the assertion is not violated, grace periods never complete, which indicates a liveness bug.

To validate our verification, we also run CBMC with the bug-injection scenarios described below,[4] which are simplified versions of bugs encountered in actual practice. Bugs 2–6 are liveness checks and thus use the aforementioned `assert(0)`, and the remaining scenarios are safety checks which thus use the base-case assertion in Figure 2.

Bug 1 This bug-injection scenario makes the RCU update-side primitive `synchronize_rcu()` return immediately (line 523 in `tree_plugin.h`). With this injected bug, updaters never wait for readers, which should result in a safety violation, thus preventing Figure 2's assertion from holding.

Bug 2 The key idea behind this bug-injection scenario is to prevent individual CPUs from realizing that quiescent states are needed, thus preventing them from recording quiescent states. As a result, it prevents grace periods from completing. Specifically, in function `rcu_gp_init()`, for each `rcu_node` structure in the combining tree, we set the field `rnp->qsmask` to 0 instead of `rnp->qsmaskinit` (line 1889 in `tree.c`). Then when `rcu_process_callbacks()` is called, `rcu_check_quiescent_state()` will invoke `__note_gp_changes()` that sets `rdp->qs_pending` to 0. Thus, `rcu_check_quiescent_state()` will return without calling `rcu_report_qs_rdp()`, preventing grace periods from completing. This liveness violation should fail to trigger a violation of the end-of-execution `assert(0)`.

Bug 3 This bug-injection scenario is a variation of Bug 2, in which each CPU remains aware that quiescent states are required, but incorrectly believes that it has already reported a quiescent state for the current grace period. To accomplish this, in `__note_gp_changes()`, we clear `rnp->qsmask` by adding a statement `rnp->qsmask &= ~rdp->grpmask;` in the last `if` code block (line 1739 in `tree.c`). Then function `rcu_report_qs_rnp()` never walks up the `rcu_node` tree, resulting in a liveness violation as in Bug 2.

Bug 4 This bug-injection scenario is an alternative code change that gets the same effect as does Bug 2. For this alternative, in `__note_gp_changes()`, we set the `rdp->qs_pending` field to 0 directly (line 1749 in `tree.c`). This is a variant of Bug 2 and thus also a liveness violation.

Bug 5 In this bug-injection scenario, CPUs remain aware of the need for quiescent states. However, CPUs are prevented from recording their quiescent states, thus preventing grace periods from ever completing. To accomplish this, 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, which constitutes a liveness violation similar to Bug 2.

Bug 6 In this bug-injection scenario, CPUs are aware of the need for quiescent states, and they also record them locally. However, they are prevented from reporting them up

[4] Source code is available: `http://lxr.free-electrons.com/source/kernel/rcu/?v=4.3`

the `rcu_node` tree, which again prevents grace periods from ever completing. This bug modifies function `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 again a liveness violation similar to Bug 2.

Bug 7 Where Bug 6 prevents quiescent states from being reported up the `rcu_node` tree, this bug-injection scenario causes quiescent states to be reported up the tree prematurely, before all the CPUs covered by a given subtree have all reported quiescent states. To this end, 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`). Then the tree-walking process will not stop until it reaches the root, resulting in too-short grace periods. This is therefore a safety violation similar to Bug 1.

Bugs 2 and 3 would result in a too-short grace period given quiescent-state forcing, but such forcing falls outside the scope of this paper.

6.2 Validating the RCU Model in User-Space

To validate our RCU model before performing verification using CBMC, we executed it in user space. We performed 1000 runs for each scenario in Section 6.1 using a 60 s timeout to wait for the end of a grace period and a random delay between 0 to 1 s in the RCU reader task.

The results are reported in Table 1. Column 1 gives the verification scenarios. Scenario Prove tests our RCU model without bug injection. Scenario Prove-GP tests a weak form of liveness by replacing Figure 2's assertion with `assert(0)` as described in Section 6.1. The next three columns present the number and the percentage of successful, failing, and timeout runs, respectively. The following two columns give the maximum memory consumption and the total runtime. The last column explains the results.

As expected, for scenario Prove, the user program ran to completion successfully in all runs. For Prove-GP, it was able to detect the end of a grace period by triggering an assertion violation in all the 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. However, we were able to observe a failure in 242 out of 1000 runs with two reader threads.

6.3 Getting CBMC to work on Tree RCU

We have found that getting CBMC to work on our RCU model is non-trivial due to Tree RCU's complexity combined with CBMC's bit-precise verification. In fact, early attempts resulted in SAT formulas that were so large that CBMC ran out of memory. After the optimizations described in the remainder of this section, the largest formula contained around 90 million variables and 450 million clauses, which enabled 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 Section 5. 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, the support for linked lists in CBMC version 5.4 is limited, resulting in unreachable code in CBMC's symbolic execution. Thus, we stubbed all the list-related code in our RCU model, including those 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 RCU-BH's data structures 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 `rcu_for_each_node_breadth_first()` loops traversing this tree.

Fourth, CBMC unwinds each loop to the depth specified in its command line option `--unwind`, even when the actual loop depth is smaller. This unnecessarily increases formula size, especially for loops containing intricate RCU code. Since loops in our model can be decided at compile time, we therefore used the command line option `--unwindset` to specify unwinding depths for each individual 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 by using the command line option `--ILP32`. This reduces CBMC's formula size by half compared to the 64-bit default.

6.4 Results and Discussion

Table 2 presents the results of our experiments applying CBMC version 5.4 to verify our RCU model. Scenario Prove verifies our RCU model without bug injection over Sequential Consistency (SC). We also exercise the model over the weak memory models TSO and PSO in scenarios Prove-TSO and Prove-PSO, respectively. Scenario Prove-GP performs the same reachability check as in Section 6.2 over SC. We perform the same reachability verification over TSO and PSO in scenarios Prove-GP-TSO and Prove-GP-PSO, respectively. Scenarios Bug 1–7 are the bug-injection scenarios discussed in Section 6.1, and are verified over SC, TSO and PSO. Columns 2–4 give the number of constraints (symbolic program expressions and partial orders), variables, and clauses of the generated formula. The next three columns give the maximum (virtual) memory consumption, solver runtime, and total runtime of our experiments. The final column gives the verification result.

Since Tree RCU's implementation in the Linux kernel is sophisticated, its test suite is non-trivial [**?**], comprising several thousand lines of code. Therefore, it comes as little surprise that its verification is challenging.

Scenario	#Successful Runs	#Failing Runs	#Timeouts	Max VM	Runtime	Result
Prove	1,000 (100.0%)	0 (0.0%)	0 (0.0%)	361.5 MB	3mins 51s	Safe
Prove-GP	0 (0.0%)	1,000 (100.0%)	0 (0.0%)	361.5 MB	5mins 9s	End of GP Reachable
Bug 1	461 (46.1%)	539 (53.9%)	0 (0.0%)	361.5 MB	5mins 26s	Assertion Violated
Bug 2	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	End of GP Unreachable
Bug 3	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	End of GP Unreachable
Bug 4	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	End of GP Unreachable
Bug 5	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	End of GP Unreachable
Bug 6	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	End of GP Unreachable
Bug 7	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	**Safe (Bug Missed)**
Bug 7 (2 readers)	758 (75.8%)	242 (24.2%)	0 (0.0%)	369.7 MB	4mins 40s	Assertion Violated

Table 1: Experimental Results of Testing the RCU Model in User-Space

Scenario	#Constraints	#Variables	#Clauses	Max VM	Solver Time	Total Time	Result
Prove	5,279,600	30,085,337	149,758,548	23.27 GB	9h 24mins	9h 36mins	Safe
Prove-TSO	5,646,959	42,042,386	210,708,442	34.00 GB	10h 51mins	11h 4mins	Safe
Prove-PSO	5,617,154	41,327,066	207,042,629	33.76 GB	11h 23mins	11h 36mins	Safe
Prove-GP	5,476,540	30,655,428	152,743,545	23.90 GB	3h 52mins	4h 5mins	End of GP Reachable
Prove-GP-TSO	5,646,940	42,041,740	210,705,615	34.00 GB	13h 1mins	13h 14mins	End of GP Reachable
Prove-GP-PSO	5,617,135	41,326,420	207,039,802	33.76 GB	8h 24mins	8h 37mins	End of GP Reachable
Bug 1	1,343,449	11,719,966	56,027,980	8.24 GB	31mins	33mins	Assertion Violated
Bug 1-TSO	1,540,645	17,120,555	83,392,397	12.60 GB	53mins	56mins	Assertion Violated
Bug 1-PSO	1,514,657	16,548,819	80,481,851	12.42 GB	46mins	48mins	Assertion Violated
Bug 2	5,279,584	30,056,615	149,643,492	23.26 GB	4h 25mins	4h 37mins	End of GP Unreachable
Bug 2-TSO	5,646,940	42,013,372	210,592,015	34.01 GB	9h 57mins	10h 10mins	End of GP Unreachable
Bug 2-PSO	5,617,135	41,298,052	206,926,202	33.75 GB	8h 51mins	9h 4mins	End of GP Unreachable
Bug 3	6,374,373	34,856,577	174,131,331	28.04 GB	7h 11mins	7h 25mins	End of GP Unreachable
Bug 3-TSO	6,805,631	48,788,433	245,157,184	41.18 GB	19h 40mins	19h 55mins	End of GP Unreachable
Bug 3-PSO	6,773,763	48,023,601	241,237,629	40.95 GB	19h 19mins	19h 35mins	End of GP Unreachable
Bug 4	4,847,980	27,804,363	138,197,043	22.18 GB	4h 3mins	4h 14mins	End of GP Unreachable
Bug 4-TSO	5,170,928	38,480,891	192,605,939	31.49 GB	8h 18mins	8h 30mins	End of GP Unreachable
Bug 4-PSO	5,141,123	37,765,571	188,940,126	31.27 GB	8h 14mins	8h 26mins	End of GP Unreachable
Bug 5	5,161,874	29,510,828	146,787,005	23.02 GB	4h 6mins	4h 18mins	End of GP Unreachable
Bug 5-TSO	5,522,168	41,239,083	206,569,643	33.65 GB	5h 46mins	5h 59mins	End of GP Unreachable
Bug 5-PSO	5,492,607	40,529,619	202,933,839	33.04 GB	5h 42mins	5h 55mins	End of GP Unreachable
Bug 6	1,410,495	13,165,176	63,302,559	9.03 GB	19mins	21mins	End of GP Unreachable
Bug 6-TSO	1,541,937	17,286,058	84,131,818	12.59 GB	1h 32mins	1h 33mins	End of GP Unreachable
Bug 6-PSO	1,518,307	16,766,198	81,485,361	12.44 GB	1h 22mins	1h 24mins	End of GP Unreachable
Bug 7	5,022,249	29,242,760	145,389,516	22.87 GB	8h 48mins	9h	**Safe (Bug Missed)**
Bug 7-TSO	5,201,744	40,139,251	200,857,404	31.93 GB	11h 6mins	11h 18mins	Assertion Violated
Bug 7-PSO	5,172,720	39,442,675	197,287,644	31.71 GB	11h 32mins	11h 44mins	Assertion Violated
Bug 7 (2 readers) *	15,165,557	71,205,400	359,021,922	59.07 GB	19h 2mins	19h 40mins	Assertion Violated
Bug 7-TSO (2 readers) *	15,691,102	90,444,903	456,973,933	74.80 GB	78h 12mins	78h 53mins	Assertion Violated
Bug 7-PSO (2 readers) *	15,647,504	89,398,551	451,611,664	74.51 GB	84h 21mins	85h 2mins	**Solver Out of Memory**

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

Table 2: Experimental Results of CBMC

In our experiments, CBMC returned all the expected results except for Bug 7, for which it failed to report a violation of the assertion `assert(r2 == 0 || r1 == 1)` with one RCU reader thread running over SC. This failure was due to the approximation of the scheduling-clock interrupt by a direct function call, as described in Section 5. 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 more than 9 hours to verify our model over SC (scenario Prove). The resulting SAT formulas have more than 5m constraints, 30m variables and 149m clauses, and occupy 23 GB of memory. The formulas for scenarios Prove-TSO and Prove-PSO are about 40% larger than the scenario Prove. They have more than 40m variables and 200m clauses, and took more than 11 hours and 33 GB memory to solve. Although this verification consumed considerable

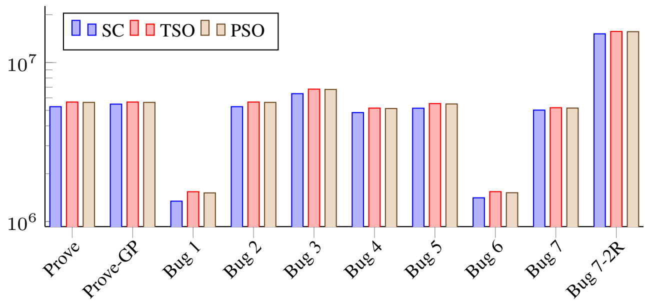

Figure 7: Number of Constraints in the SAT Formulas

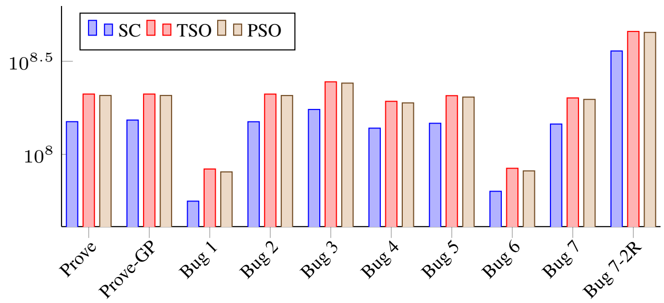

Figure 9: Number of Clauses in the SAT Formulas

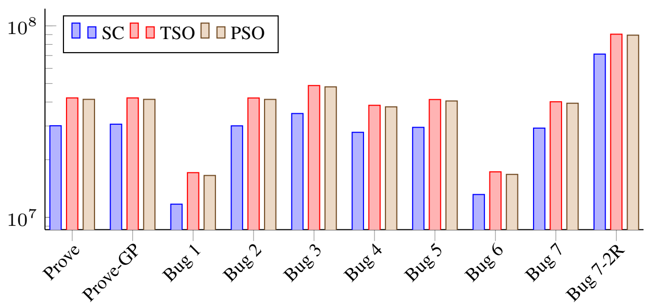

Figure 8: Number of Variables in the SAT Formulas

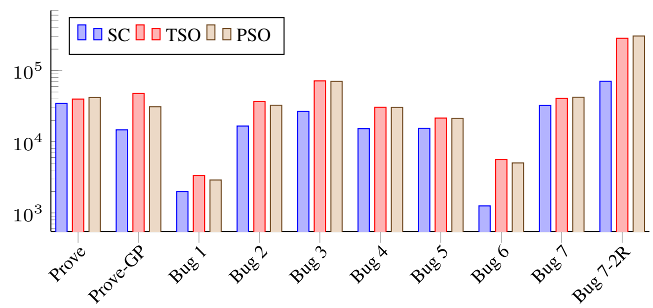

Figure 10: Total Runtime in Seconds

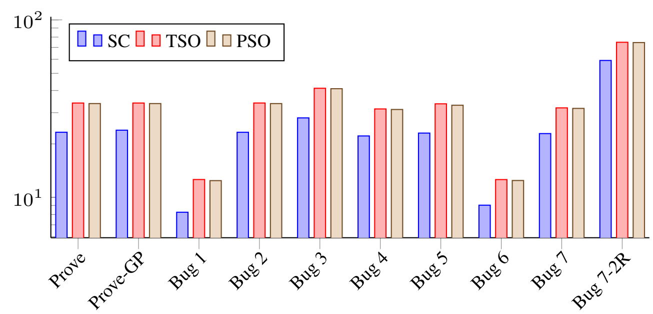

Figure 11: Maximum Memory Consumption in Gigabytes

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 (i.e., `assert(0)` is violated), over SC (Prove-GP), TSO (Prove-GP-TSO), and PSO (Prove-GP-PSO). The sizes of resulting formulas and memory consumption are similar to those of the three Prove scenarios. However, it took CBMC only about 4, 13, and 8.5 hours to find an violation of `assert(0)` in Prove-GP, Prove-GP-TSO, and Prove-GP-PSO, respectively.

For the bug-injection scenarios described in Section 6.1, CBMC was able to return the expected results in all scenarios over SC except for Bug 7, as noted earlier. The formula size varies from scenarios to scenarios, with 27m–35m variables and 138m–174m clauses. The runtime was 4–9 hours and memory consumption exceeded 22 GB. The exceptions are Bugs 1 and 6, which have fewer than 14m variables and 64m clauses, and took less than 35 mins and about 9 GB of memory to solve. This reduction was due to the large amount of code removed by the bug injections in these scenarios.

Figures 7–8 compare the formula size between SC, TSO and TSO. Comparison of runtime and memory can be found in Figures 10 and 11. As we can see, the runtime and memory overhead for the TSO and PSO variants of a given experiment are quite similar. The overheads of TSO are slightly higher than those of PSO in all bug-injection scenarios except for Bug 7 on which PSO had longer runtime. However, the overhead of TSO and PSO is significantly larger than that of SC, with up to 340% (Bug 6 runtime) and 50% (Bug 1 memory) increases. The runtime was 5–19 hours and memory consumption exceeded 31 GB in all scenarios except Bug 1 and 6. The numbers of variables and clauses are 37m–49m and 188m–245m, respectively, around 130% greater than SC.

The two-reader variant of Bug 7 has by far the longest runtime, consuming more than 19 hours and 78 hours over SC and TSO, respectively, comparing to 9 hours and 11 hours with one reader. It also consumed about 75 GB memory, more than double the one-reader variant. For PSO, with two reader threads 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. For example, the two-reader TSO formula has triple the constraints and double the variables and clauses of the one-reader case.

7. Related Work

McKenney applied the SPIN model checker to verify RCU's NO_HZ_FULL_SYSIDLE functionality [**?**], and interactions between dyntick-idle and non-maskable interrupts [**?**]. Desnoyers et al. [**?**] propose a virtual architecture to model out-of-order memory accesses and instruction scheduling. User-level RCU [**?**] is modeled and verified in the proposed architecture using the SPIN model checker.

These efforts require an error-prone 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, and verifies it using automated verification tool.

Alglave et al. [**?**] introduce a symbolic encoding for verifying concurrent software over a range of memory models including SC, TSO and PSO. They implement the encoding in the CBMC bounded model checker and use the tool to verify rcu_assign_pointer() and rcu_dereference().

McKenney used CBMC to verify Tiny RCU [**?**], a trivial Linux-kernel RCU implementation for uni-core systems.

Groce et al. [**?**] introduce a falsification-driven verification methodology that is based on a variation of mutation testing. By using CBMC, they were 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 identified two more rcutorture holes, one of which was hiding a real bug in Tasks RCU [**?**] and the other of which was hiding a minor performance bug in Tree RCU.

In this work, we use CBMC to verify the implementation of Linux-kernel Tree RCU for multi-core systems, which is more complex and sophisticated, over SC, TSO, and PSO.

Gotsman et al. [**?**] use a extended concurrent separation logic to formalise the concept of grace period and prove an abstract implementation of RCU over SC. Tassarotti et al. [**?**] use GPS, a recently developed program logic for the C/C++11 memory model, to carry out a formal proof of a simple implementation of user-level RCU for a singly-linked list assuming "release-acquire" semantics, which is weaker than SC but stronger than memory models used by real-world RCU implementations. These formal proofs were performed manually on simple implementations of RCU. By contrast, our work applies an automated verification tool with a test harness to verify the grace-period property of a real-world implementation of RCU over SC, TSO, and PSO.

Formal verification has started to make its way into real-world practice of verifying large non-trivial code bases. Calcagno et al. [**?**] describe integrating a static-analysis tool into Facebook's software development cycle. We believe that our work is an important step towards integration of verification into Linux-kernel RCU's regression test suite.

8. Conclusion

This paper overviews the implementation of Tree RCU in the Linux Kernel, and describes how to construct a model directly from its source code. It then 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 demonstrates 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 sufficient concurrency and complexity to drive significant advances in techniques and tooling.

For future work, we plan to add quiescent-state forcing and grace-period expediting into our model and verify their safety and liveness properties, using more sophisticated test harnesses that pass through multiple grace periods and operate on a larger tree structure. We also plan to model and verify the preemptible version of Tree RCU, which we expect to be quite challenging. Moreover, there is much fertile ground verifying uses of RCU in the Linux kernel, for example, the Virtual File System (VFS).

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

This work demonstrates the nascent ability 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. We therefore hypothesize that use of these tools for highly concurrent multithreaded software will reach mainstream within 3-5 years, especially given recent rates of improvement.