Stateless Model Checking of the Linux Kernel's Hierarchical Read-Copy-Update (Tree RCU)

Michalis Kokologiannakis[1] Konstantinos Sagonas[1,2]

[1] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
[2] Department of Information Technology, Uppsala University, Sweden

Abstract

Read-Copy-Update (RCU) is a synchronization mechanism used heavily in key components of the Linux kernel, such as the virtual filesystem (VFS), to achieve scalability by exploiting RCU's ability to allow concurrent reads and updates. RCU's design is non-trivial, requires significant effort to fully understand it, let alone become convinced that its implementation is faithful to its specification and provides its claimed properties. The fact that as time goes by Linux kernels are becoming increasing more complex and are employed in machines with more and more cores and weak memory does not make the situation any easier.

This paper presents an approach to systematically test the code of the main flavor of RCU used in the Linux kernel (Tree RCU) for concurrency errors, both under sequential consistency and weak memory. Our modeling allows Nidhugg, a stateless model checking tool, to reproduce, within seconds, safety and liveness bugs that have been reported for RCU. More importantly, we were able to validate the Grace-Period guarantee, the basic guarantee that RCU offers, on different non-preemptible Linux kernel versions. Our approach is effective, both in dealing with the increased complexity of recent kernels and in terms of time that the validation requires. We have good reasons to believe that our effort constitutes a big step towards making tools such as Nidhugg part of the standard testing infrastructure of the Linux kernel.

1. Introduction

The Linux kernel is used in a surprisingly large number of devices today: from PCs and servers, to routers and smart TVs. For example, more than one billion smart phones use a modified version of the Linux kernel in 2015 [5], and almost all modern supercomputers use Linux as well [26]. It is self-evident that the correct and reliable operation of the Linux kernel is of great importance, which renders its thorough testing for concurrency errors and its validation a necessity.

Naturally, thorough testing of the Linux kernel requires the testing of all of its components and subsystems. One particular subsystem with a non-trivial implementation is the Read-Copy-Update (RCU) mechanism [22, 23]. RCU is a synchronization mechanism that provides excellent scalability by enabling concurrent reads and updates.

However, RCU's implementation is quite involved, making the testing of this mechanism challenging. Moreover, the lockless design of its fastpaths, and the fact that it needs to operate in heavily concurrent environments make its validation extremely challenging. Still, the fact that concurrency bugs manage to survive (maybe only under particular configurations, architectures and memory models) even after heavy *stress testing* underlines the need for employing more powerful software testing techniques such as model checking.

An extra difficulty for this is the relatively short release cycle of the Linux kernel (there is a new release every approximately two months), in conjunction with the increasing complexity of each version, which introduces many changes, requiring that such techniques and the corresponding tools operate on as big a percentage of the *actual code* as possible, and are able to cope with its increasing complexity.

This paper reports on the use of stateless model checking (also known as systematic concurrency testing) for testing the core of Tree RCU, the main RCU flavor used in the Linux kernel. In particular, using Nidhugg [2], we were able to validate that its implementation preserves the *Grace-Period guarantee*, the basic guarantee that RCU offers. Our effort concentrated on a non-preemptible kernel environment, but we also investigated the effects that weak memory models (TSO in particular) may have in RCU's operation. We used the source code from five different versions of the Linux kernel, and were able to reproduce our results in all of them.

In order to strengthen the validation claim, we injected bugs similar to ones that existed throughout the development of RCU and, in all cases, our tool was able to come up with scenarios in which they occur. In particular, we were able to demonstrate that a submitted patch intended to impose a locking design, in reality fixed a much more serious bug. We report on this issue and also present the exact conditions under which this bug occurs. Finally, as we will show, our technique handles real code employed in today's production systems in a very efficient and scalable way. In fact, Nidhugg copes extremely well with the increasing complexity the newer kernel versions induce.

Overview After an introduction to RCU and to stateless model checking in the next two sections, we describe details of the implementation of Tree RCU in Section 4. Sec-

tion 5 presents our modeling of the kernel's environment. In Section 6 we reproduce an older known kernel bug and in Section 7 we report on the validation of the Grace-Period guarantee of RCU. The paper ends with a comparison with related work and some concluding remarks.

2. Read-Copy-Update (RCU)

2.1 Introduction to RCU

Read-Copy-Update is a synchronization mechanism invented by McKenney and Slingwine that is a part of the Linux kernel since 2002 [22, 23]. The key feature of RCU is the good scalability it provides by allowing concurrent reads and updates. While this may seem counter-intuitive or impossible at first, RCU allows this in a very simple yet extremely efficient way: by maintaining multiple data versions. RCU is carefully orchestrated in a way that not only ensures that reads are coherent and no data will be deleted until it is certain that no one holds references to them, but also uses efficient and scalable mechanisms which make read paths extremely fast. Most notably, in non-preemptible kernels, RCU imposes zero overhead to readers.

The basic idea behind RCU is to split updates in two phases: the *removal phase* and the *reclamation phase*. During the removal phase, an updater *removes* references to data either by destroying them (i.e. setting them to `NULL`), or by replacing them with references to newer versions of these data. This phase can run concurrently with reads due to the fact that modern microprocessors guarantee that a reader will see either the old or the new reference to an object, and not a weird mash-up of these two or a partially updated reference. During the reclamation phase, the updater frees the items removed in the removal phase, i.e., these items are *reclaimed*. Of course, since RCU allows concurrent reads and updates, the reclamation phase must begin after the removal phase and, more specifically, when it is certain that there are no readers accessing or holding references to the data being reclaimed.

The typical update procedure using RCU looks as follows [22]. Note that steps 2 and 4 in this procedure are not necessarily performed by the same thread.

1. Ensure that all readers accessing RCU-protected data structures carry out their references from within an RCU read-side critical section.
2. Remove pointers to a data structure, so that subsequent readers cannot gain a reference to it (removal phase).
3. Wait until all pre-existing readers complete their RCU read-side critical section, so that there no one holding a reference to the item being removed.
4. At this point, there cannot be any readers still holding references to the data structure, which may now be safely freed (reclamation phase).

Waiting for pre-existing readers can be achieved either by blocking (via `synchronize_rcu()`), or by registering a callback that will be invoked after all pre-existing readers have completed their RCU read-side critical sections (via `call_rcu()`).

In order to formalize some of the aspects presented above, we provide some definitions.

Definition 2.1. *Any statement that is not within an RCU read-side critical section is said to be in a* quiescent state.

Statements in quiescent states are not permitted to hold references to RCU-protected data structures (in Linux kernel, this is checked with the tool `sparse` [31]). Note that different RCU flavors have different sets of quiescent states.

Definition 2.2. *Any time period during which each CPU resides at least once in a quiescent state is called a* grace period.

Consequently, if an RCU read-side critical section started before the beginning of a specified grace period, it would have to complete before the end of that grace period. This means that the reclamation phase has to wait for *at least* one grace period to elapse before it begins. Once a grace period has elapsed, there can no longer be any readers holding references to the old version of a newly updated data structure (since each CPU has passed through a quiescent state) and the reclamation phase can safely begin.

2.2 RCU Specifications

Let us now present some requirements that every RCU implementation must fulfill. We do not attempt to present a formal or a complete specification for RCU here.[1] Instead, we only present the basic guarantees of RCU.

Grace-Period Guarantee The fact that in RCU updaters wait for all pre-existing readers to complete their read-side critical sections, constitutes the only interaction between the readers and the updaters. The Grace-Period guarantee is what allows updaters to wait for all pre-existing RCU read-side critical sections to complete. Such critical sections start with the macro `rcu_read_lock()` and end with `rcu_read_unlock()`. What this guarantee means is that the RCU implementation must ensure that any read-side critical sections in progress at the start of a given grace period will have completely finished (including memory operations, etc.) before that grace period ends. This very fact allows RCU validation to be focused; every correct implementation has to adhere to the following rule:

> *If any statement in a given RCU read-side critical section precedes a grace period GP, then all statements (including memory operations) in that RCU read-side critical section must complete before GP ends.*

Memory operations are included here in order to prevent the compiler or the CPU from undoing work done by RCU.

[1] RCU specifications are part of the Linux kernel documentation [29].

Initially: `int x = 0, y = 0, r_x = 0, r_y = 0;`

```
void reader(void)
{
  rcu_read_lock();
  r_x = READ_ONCE(x);
  r_y = READ_ONCE(y);
  rcu_read_unlock();
}
```

```
void updater(void)
{
  WRITE_ONCE(x, 1);
  synchronize_rcu();
  WRITE_ONCE(y, 1);
}
```

Figure 1. RCU's Grace-Period guarantee litmus test.

In order to see what this guarantee really implies, consider the code fragment in Figure 1. In this code, since `synchronize_rcu()` has to wait for all pre-existing readers to complete their RCU read-side critical sections, the outcome:

```
r_x == 0 && r_y == 1                (1)
```

should be impossible. This is what the Grace-Period guarantee is all about. It is the most important guarantee that RCU provides; in effect, it constitutes the core of RCU.

Publish-Subscribe Guarantee This guarantee is used in order to coordinate read-side accesses to data structures. The Publish-Subscribe mechanism is used in order for data to be inserted into data structures (e.g., lists), without disrupting concurrent readers. Since updaters run concurrently with readers, this mechanism ensures that readers will not see uninitialized data, and that updaters will have completed all initialization operations before publishing a data structure. For this, RCU offers the `rcu_assign_pointer()` and `rcu_dereference()` primitives.

The `rcu_assign_pointer()` has similar semantics to C11's `memory_order_release` operation, but also prevents "nasty" compiler optimizations. In effect, it is similar to an assignment but also provides additional ordering guarantees.

The `rcu_dereference()` primitive can be considered as a subscription to a value of a specified pointer, and it guarantees that subsequent dereference operations will see any initialization that took place before the `rcu_assign_pointer()` (publish) operation. The `rcu_dereference()` primitive has semantics similar to C11's `memory_order_consume` load, and uses both volatile casts and memory barriers in order for it to provide the aforementioned guarantee.

Using `rcu_assign_pointer()` and `rcu_dereference()` is required for programs that access RCU-protected pointers. Moreover, if a pointer is RCU-protected (annotated with `__rcu`), all dereferences of that pointer have to be performed within RCU read-side critical sections using the primitive `rcu_dereference()`, and all assignments to RCU-protected pointers have to use the `rcu_assign_pointer()` primitive.

3. Stateless Model Checking

Stateless model checking [15], also known as systematic concurrency testing, is a technique with low memory requirements that is applicable to programs with executions of finite length. Stateless model checking tools explore the state space of a program without explicitly storing global states. The technique has been successfully implemented in tools such as VeriSoft [16], CHESS [24], Concuerror [6], and Nidhugg [2].

Some of these tools also try to combat the problem of combinatorial explosion in the number of interleavings that need to be examined in order to maintain full coverage of all program behaviors by using *partial order reduction* [7, 14, 25, 33] techniques. Partial order reduction is based on the observation that two interleavings can be considered equivalent if one can be obtained from the other by swapping adjacent, independent execution steps. *Dynamic Partial Order Reduction* (DPOR) techniques capture dependencies between operations of concurrent threads while the program is running [12]. The exploration begins with an arbitrary interleaving whose steps are then used to identify dependent operations and points where alternative interleavings need to be explored in order to capture all program behaviors.

Stateless model checking and DPOR techniques have been extended to handle memory model non-determinism in addition to scheduling non-determinism. Nidhugg [2], for example, is a stateless model checker for C/C++ programs that use pthreads, which incorporates extensions for finding bugs caused by weak memory models such as TSO, PSO and POWER. Nidhugg's implementation employs a very effective dynamic partial order algorithm called source-DPOR [1]. In our work we used Nidhugg for all our tests.

In stateless model checking all tests need to be *data-deterministic* in the sense that, in a given state, a given execution step must always lead the system to the same new state. This means that the test case cannot depend on some unknown input or on timing properties (e.g., take some action depending on the value of the clock). In addition, all test cases need to be *finite* in the sense that there must be a bound $n \in \mathbb{N}$ such that all executions of the program terminate within n execution steps.

4. Tree RCU Implementation

The Linux kernel offers many different RCU implementations, each one serving a different purpose. The first RCU implementation (available since 2002 and Linux kernel v2.5.43) was Classic RCU. A problem with Classic RCU was lock contention due to the presence of one global lock that had to be acquired from each CPU wishing to report a quiescent state to RCU. In addition, Classic RCU's dyntick-idle interface had room for improvements, since Classic RCU had to wake up every CPU (even idle ones) at least once per grace period, thus increasing power consumption.

Tree RCU offers a solution to both these problems since it reduces lock contention and avoids awakening dyntick-idle CPUs. Tree RCU scales to thousands of CPUs easily, while Classic RCU could scale only to several hundred.

In this section, we will present a high-level explanation of Tree RCU along with some implementation details (the implementation of Tree RCU greatly varies between different

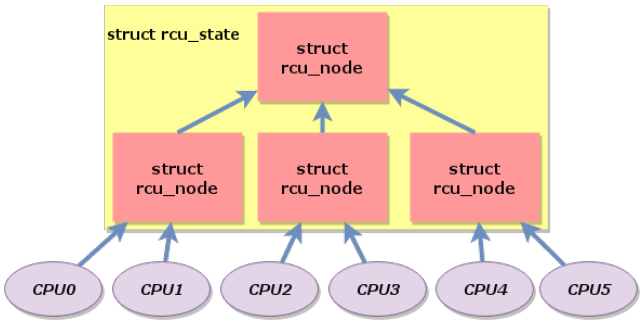

Figure 2. Tree RCU node hierarchy (adapted from [20]).

kernel versions), a brief overview of its data structures, and some use cases that are helpful in understanding the way RCU's fundamental mechanisms are actually implemented.

4.1 High-Level Explanation

In Classic RCU each CPU had to clear its bit in a field of a global data structure after it had passed through a quiescent state. Since CPUs operated concurrently on this data structure, a spinlock was used to protect the mask, something that could potentially suffer from extreme contention.

Tree RCU addresses this issue by creating a heap-like node hierarchy. The key here is that CPUs will not try to acquire the same node's lock when trying to report a quiescent state to RCU; in contrast, CPUs are split into groups and each group will contend for a different node's lock. Each CPU has to clear its bit in the corresponding node's mask once per grace period. The last CPU to check in (i.e. to report a quiescent state to RCU) for each group, will try to acquire the lock of the node's parent, until the root node's mask is cleared. This is when a grace period can end. (We will examine the exact way this happens in Section 4.3.) A simple node hierarchy for a 6-CPU system is presented in Figure 2. As can be seen in the figure, CPU0 and CPU1 will acquire the lower-left node's lock, CPU2 and CPU3 will acquire the lower-middle node's lock, and CPU4 and CPU5 will acquire the lower-right node's lock. The last CPU reporting a quiescent state for each of the lower nodes, will try to acquire the root node's lock, and this procedure happens once per grace period.

The node hierarchy created by Tree RCU is tunable, and is controlled, among others, by two `Kconfig` options, namely:

`CONFIG_RCU_FANOUT_LEAF`: Controls the leaf-level fanout of Tree RCU, i.e. the maximum number of CPUs contending for a leaf-node's lock. Default value is 16.

`CONFIG_RCU_FANOUT`: Controls the fanout of Tree RCU, i.e. the maximum number of CPUs contending for an inner-node's lock. Default value is 32 for 32-bit systems and 64 for 64-bit systems.

More information can be found at the `init/Kconfig` file.

4.2 Data Structures

Let us now describe three major data structures of Tree RCU's implementation: `rcu_data`, `rcu_node`, and `rcu_state`. Suppose that a CPU registers a callback that will eventually be invoked. Tree RCU needs to store some information regarding this callback. For this, the implementation maintains some data organized in the per-CPU `rcu_data` structures, which include, among others:

- The last completed grace period number this CPU has seen; used for grace-period ending detection (`completed`).
- The highest grace period number this CPU is aware of having started (`gpnum`).
- A `bool` variable indicating whether this CPU has passed through a quiescent state for this grace period.
- A pointer to this CPU's leaf of hierarchy.
- The mask that will be applied to the leaf's mask (`grpmask`).
- Variables related to callback handling, including this CPU's callback list.
- Variables related to the dynticks interface, as well as a pointer to this CPU's `rcu_dynticks` structure.
- A pointer to the RCU global `rcu_state` structure.

Thus, when a CPU registers a callback, it stores it in the respective per-CPU data structure.

Now, when a CPU passes through a quiescent state, it has to report it to RCU by clearing its bit in the respective leaf node. The node hierarchy consists of `rcu_node` structures which include, among others:

- A lock protecting the respective node.
- The current grace period number for this node.
- The last completed grace period number for this node.
- A bit-mask indicating CPUs or groups that need to check in in order for this grace period to proceed (`qsmask`). In leaf nodes, each bit corresponds to an `rcu_data` structure, and in inner nodes, each bit corresponds to a child `rcu_node`.
- A bit-mask that will be applied to parent node's mask (`grpmask`).
- The number of the lowest and the highest CPU or group for this node.
- A pointer to the node's parent.

Lastly, the RCU global state, as well as the node hierarchy are included in an `rcu_state` structure. The node hierarchy is represented in heap form in a linear array, which is allocated statically at compile time based on the values of `NR_CPUS` and the `Kconfig` options. Note that small systems have a hierarchy consisting of a single `rcu_node`. This structure contains, among others:

- The node hierarchy.
- A pointer to the per-CPU `rcu_data` variable.

- The current grace-period number.
- The number of last completed grace period.

There are several values that are propagated through these different structures, e.g., the grace period number. However, this was not always the case, and it was the discovery of bugs that often led to changes in the source code.

Finally, we have already mentioned that Classic RCU had a sub-optimal dynticks interface, and that one of the main reasons for the creation of Tree RCU was to leave sleeping CPUs lie, in order to conserve energy. Tree RCU avoids awakening low-power-state dynticks-idle CPUs using a per-CPU data structure called `rcu_dynticks`. This structure contains, among others:

- A counter tracking the irq/process nesting level.
- A counter containing an even value for dynticks-idle mode, else containing an odd value.

These counters enable Tree RCU to wait only for CPUs that are not sleeping, and to let sleeping CPUs lie. How this is achieved is described below.

4.3 Use Cases

The common usage of RCU involves registering a callback, waiting for all pre-existing readers to complete, and finally, invoking the callback. During all these, special care is taken to accommodate sleeping CPUs, offline CPUs and CPU hot plugs, CPUs in user-land, and CPUs that fail to report a quiescent state to RCU within a reasonable amount of time.

Registering a Callback A CPU registers a callback by invoking `call_rcu()`. This function queues an RCU callback that will be invoked after a specified grace period. The callback is placed in the callback list of the respective CPU's `rcu_data` structure. This list is partitioned in four segments:

1. The first segment contains entries that are ready to be invoked (`DONE` segment).
2. The second segment contains entries that are waiting for the current grace period (`WAIT` segment).
3. The third segment contains entries that are known to have arrived before the current grace period ended (`NEXT_READY` segment).
4. The fourth segment contains entries that *might* have arrived after the current grace period ended (`NEXT` segment).

When a new callback is added to the list, it is inserted at the end of the fourth segment.

In older kernels (e.g., v2.6.x), `call_rcu()` could start a new grace period directly, but this is no longer the case. In newer kernels, the only way a grace period can start directly by `call_rcu()` is if there are too many callbacks queued and no grace period in progress. Otherwise, a grace period will start from *softirq* context.

Every softirq is associated with a function that will be invoked when this type of softirqs is executed. For Tree RCU, this function is called `rcu_process_callbacks()`. So, when an RCU softirq is raised, this function will eventually be invoked (either at the exit from an interrupt handler or from a `ksoftirq/n` kthread), and will start a grace period if there is need for one (e.g., if there is no grace period in progress and the CPU has newly registered callbacks, or there are callbacks that require an additional grace period). RCU softirqs are raised from `rcu_check_callbacks()` which is invoked from scheduling-clock interrupts. If there is RCU-related work (e.g., if this CPU needs a new grace period), `rcu_check_callbacks()` raises a softirq.

Note that, in Tree RCU, `synchronize_rcu()` is implemented on top of `call_rcu()`. `synchronize_rcu()` registers a callback that will awake the caller after a grace period has elapsed. The caller waits on a completion variable, and is consequently put on a wait queue.

Starting a Grace Period Function `rcu_start_gp()` is responsible for starting a new grace period; it is normally invoked from softirq context and an `rcu_process_callbacks()` call. However, in newer kernels, `rcu_start_gp()` neither directly starts a new grace period nor initializes the necessary data structures. It rather advances the CPU's callbacks (i.e. properly re-arranges the segments), and then sets a flag at the `rcu_state` structure to indicate that a CPU requires a new grace period. The grace-period kthread is the one that will initialize the node hierarchy and the `rcu_state` structure, and by extension start the new grace period.

The RCU grace-period kthread excludes concurrent CPU-hotplug operations and then sets the quiescent-state-needed bits in all the `rcu_node` structures in the hierarchy corresponding to online CPUs. It also copies the grace period number and the number of the last completed grace period in all the `rcu_node` structures. Concurrent CPU accesses will check only the leaves of the hierarchy, and other CPUs may or may not see their respective node initialized. But each CPU has to enter the RCU core in order to acknowledge that a grace period has started and initialize its `rcu_data` structure. This means that each CPU (except for the one on which the grace-period kthread runs) needs to enter softirq context in order to see the new grace period beginning (via the `rcu_process_callbacks()` function).

The grace-period kthread resolved many races present in older kernels, where when a CPU required a new grace period, it tried to directly initialize the node hierarchy, something that can potentially lead to bugs; see Section 6.

Passing Through a Quiescent State Quiescent states for Tree RCU (RCU-sched) include: (i) context switch, (ii) idle mode (idle loop or dynticks-idle), and (iii) user-mode execution. When a CPU passes through a quiescent state, it updates its `rcu_data` structure by invoking `rcu_sched_qs()`. This function is invoked from scheduling-related functions, from `rcu_check_callbacks()`, and from the `ksoftirq/n` kthreads. However, the fact that a CPU has passed through a quiescent state does not mean that RCU knows about it. Be-

sides, this fact has been recorded in the respective per-CPU `rcu_data` structure, and not in the node hierarchy. So, a CPU has to report to RCU that it has passed through a quiescent state, and this will happen —again— from softirq context, via the `rcu_process_callbacks()` function; see below.

Reporting a Quiescent State to RCU After a CPU has passed through a quiescent state, it has to report it to RCU via `rcu_process_callbacks()`. This function has a lot of duties, including:

- Awakening the RCU grace-period kthread (by invoking the `rcu_start_gp()` function), in order for it to initialize and start a new grace period, if there is need for one.
- Acknowledging that a new grace period has started/ended. Every CPU except for the one on which the RCU grace-period kthread runs has to enter the RCU core and see that a new grace period has started/ended. This is done by invoking `rcu_check_quiescent_state()`, which in turn invokes the `note_gp_changes()` function. The latter advances this CPU's callbacks accordingly and records to the respective `rcu_data` structure all the necessary information regarding the grace-period beginning/end.
- Reporting that the current CPU has passed through a quiescent state (via the `rcu_report_qs_rdp()` function, which is invoked from `rcu_check_quiescent_state()`). If the current CPU is the last one to report a quiescent state, the RCU grace-period kthread is awakened once again, in order for it to clean up after the old grace period and propagate the new `->completed` value to the `rcu_node` structures of the hierarchy.
- Invoking any callbacks whose grace period has ended.

As can be seen, the RCU grace-period kthread is used heavily to coordinate grace-period beginnings and ends. Apart from this, the locks of the nodes in the hierarchy are used to prevent concurrent accesses which might lead to problems.

Entering/Exiting Dynticks-Idle Mode When a CPU enters dynticks-idle mode, `rcu_idle_enter()` is invoked. This function decrements a per-CPU nesting variable (namely `dynticks_nesting`) and increments a per-CPU counter (namely `dynticks`), both of which are located in the per-CPU `rcu_dynticks` structure. The `dynticks` counter must have an even value when entering dynticks-idle mode. When a CPU exits dynticks-idle mode, `rcu_idle_exit()` is invoked. This function increments the `dynticks_nesting` and the `dynticks` counter (which must now have an odd value).

However, dynticks-idle mode is a quiescent state for Tree RCU. So, the reason these two variables are needed is the fact that they can be sampled by other CPUs. This way, we can determine if a CPU is in a quiescent state, or if a CPU has been in a quiescent state at some point during the grace period in progress. The sampling process is performed when a CPU has not reported a quiescent state for a long time and the grace period needs to end (quiescent state forcing).

Interrupts and Dynticks-Idle Mode When a CPU enters an interrupt handler, `rcu_irq_enter()` is invoked from `irq_enter()`. This function increments the value of the `dynticks_nesting` variable, and if the prior value was zero (which means that the CPU was in dynticks-idle mode), also increments the `dynticks` counter. When a CPU exits an interrupt handler, `rcu_irq_exit()` decrements the `dynticks_nesting` variable, and if the new value is zero (which means that the CPU is entering dynticks-idle mode), also increments the `dynticks` counter. It is self-evident that entering an interrupt handler from dynticks-idle mode means exiting the dynticks-idle mode. Conversely, exiting an interrupt handler might mean entrance into dynticks-idle mode.

Forcing Quiescent States If not all CPUs have reported a quiescent state and several `jiffies` have passed, then the grace-period kthread is awakened and it will try to force quiescent states on CPUs that have not reported one. More specifically, it will invoke the `rcu_gp_fqs()` function. This function works in two phases: in the first phase snapshots of the `dynticks` counters of all CPUs are collected, in order to credit them with implicit quiescent states. In the second phase, CPUs that have yet to report a quiescent state are scanned again, in order to determine if they have passed through a quiescent state from the moment their snapshots were collected. If there are still CPUs that have not checked in, they are forced into the scheduler in order for them to report a quiescent state to RCU.

5. Kernel Environment Modeling

Let us now present the way we scaffolded a non-preemptible Linux-kernel SMP environment. In order to achieve this, we had to disable some timing-based warnings, and to stub out some primitives used in functions that were not included in our tests (e.g., RCU-expediting related primitives). However, we note that the only changes we made in the source code of Tree RCU involved the replacement of per-CPU variables with arrays; the rest of the source code remains untouched.

5.1 CPU, Interrupts and Scheduling

CPU Since we emulate an SMP system, we need some kind of mutual exclusion between threads running on the same CPU, for each CPU of the system. Thus, we provide an array of locks (namely `cpu_lock`), with each array entry corresponding to a CPU. When one of these locks is held, the corresponding thread is running on the respective CPU.

We assume that all CPUs are online, that there are no CPU hot plug operations, and that `CONFIG_NO_HZ_FULL=n`. All CPUs are initially idle, and when a thread wishes to run on a CPU, it acquires the CPU's lock and exits idle mode (if necessary). Accordingly, when a thread finishes its execution it enters idle mode and then releases the CPU's lock.

We also need to emulate per-CPU variables. In the kernel, these variables are created using compiler/linker directives along with carefully written preprocessor code. However,

since these variables require significant runtime support, we used arrays to emulate them. Each array entry represents the respective processor's copy of a per-CPU variable.

Since a thread needs to have knowledge regarding the CPU it runs on, we have implemented two macros (`set_cpu()` and `get_cpu()`), which manipulate a thread-local variable indicating the CPU on which a thread runs. The processor on which a thread runs has to be manually set, via `set_cpu()`. The total number of CPUs can be manipulated by setting the `-DCONFIG_NR_CPUS` preprocessor option appropriately.

Interrupts and Softirqs In order to emulate interrupts and softirqs we used an array of locks (`irq_lock()`), with each lock corresponding to a CPU. An entry's lock must be held across an interrupt handler by the thread servicing the interrupt on the respective CPU. Of course, a thread has to acquire the CPU's lock first, and then the interrupt lock. In a similar manner, when a thread disables interrupts on a CPU, the same lock has to be acquired. Since we are dealing with non-preemptible kernels, this lock is not contended.

RCU mostly relies on scheduling-clock interrupts and the `rcu_check_callbacks()` function, so we also need to model scheduling-clock interrupts. As mentioned, stateless model checking requires the tests to be deterministic, meaning that timing-based actions cannot be included in the tests. However, the exact time an interrupt occurs is not so important; instead what interests us are the implications a scheduling-clock interrupt might have at a certain point of a program's execution given a concurrency context. Consequently, our version of the interrupt handler first calls the `rcu_check_callbacks()` function and then, if an RCU softirq is raised, the `rcu_process_callbacks()` function is invoked. Of course, we could have just called the `rcu_process_callbacks()` function, but, in the Linux kernel, this function is not invoked unconditionally and we wanted our model to be as precise to the real code as possible.

Scheduling The `cond_resched()` function is modeled by having the running thread drop the CPU's lock and then (possibly) re-acquire it, but with `rcu_note_context_switch()` being invoked before releasing the lock of the incoming CPU.

While the best way to model this function would probably have been to drop the current CPU's lock and then acquire the lock of a random CPU, the deterministic nature of stateless model checking renders this approach impossible. Nevertheless, our tests aim to be CPU-specific and not thread-specific, in the sense that we care about the actions of each CPU and not for the actions of each thread. We care about CPUs executing interrupts or softirqs, performing context switches and accessing shared variables, and not about the specific threads performing these actions. Last, we note that we stubbed the `resched_cpu()` function, since this is only used in kthread kicking which is not included in our tests.

5.2 Kernel Definitions

Many kernel definitions were copied directly from the Linux kernel. These include data types like `u8`, `u16`, etc., compiler directives like `offsetof()`, macros like `ACCESS_ONCE()`, list data types and functions, memory barriers, as well as various other kernel primitives. On the other hand, many primitives had to be replaced or stubbed; we supplied empty files for `#include` directives, and provided some other definitions based on some specific `Kconfig` options. These include CPU-relevant definitions (e.g., `NR_CPUS`), RCU-related definitions that are normally configured at compile time (e.g., `CONFIG_RCU_BOOST`), special compiler directives, tracing functions, etc. The `BUG_ON()` macro and its relatives (e.g., `WARN_ON()`) have been replaced by `assert()` statements. Note that we only stubbed primitives irrelevant to our tests (e.g., some primitives related to grace-period expediting), and provided our own definitions for some other primitives in order for them to work with our modeling of the CPUs and interrupts.

All of the definitions we used reside in separate files; these can be copied and reused across multiple kernel versions.

5.3 Synchronization Mechanisms

The emulation of the Linux kernel's synchronization mechanisms used in Tree RCU's implementation is as follows:

Atomic Operations While we copied the atomic data type `atomic_t` directly from the Linux kernel, this is not the case for atomic operations like `atomic_read()`, `atomic_set()`, `atomic_add()`, etc., since their implementation is architecture dependent. In order to emulate those, we used some `GCC` language extensions [13] supported by `clang` [19], the compiler that produces the `LLVM IR` code that Nidhugg analyzes.

Spinlocks and Mutexes We used `pthread_mutex` for the emulation of spinlocks. In like manner, `pthread_mutex_t` has been used in place of `struct mutex`. Since many spinlocks and mutexes are initialized statically in the kernel, the Nidhugg option `--disable-mutex-init-requirement` is required for most tests to run.

Completions In order to emulate completion variables, we copied the data type definition directly from the Linux kernel, but we had to model wait queues. For this, we emulated completions by spin-waiting. Since a thread waiting on a completion is put on a wait queue until a condition is satisfied, we used spin loops in order to emulate wait queues. Nidhugg automatically transforms all spin loops to `__VERIFIER_assume()` statements where, if the condition does not hold, the execution blocks indefinitely. Before waiting on a spin loop, the thread drops the corresponding CPU's lock; it will try to re-acquire it after the condition has been satisfied. Since this is a quiescent state for RCU, the function `rcu_note_context_switch()` (and possibly also the `do_IRQ()` function to report a quiescent state to RCU) could be invoked before the thread releases the CPU's lock.

```
completed_snap = ACCESS_ONCE(rsp->completed); /* outside of lock */

/* Did another grace period end? */
if (rdp->completed != completed_snap) {
  /* Advance callbacks.  No harm if list empty. */
  rdp->nxttail[RCU_DONE_TAIL] = rdp->nxttail[RCU_WAIT_TAIL];
  rdp->nxttail[RCU_WAIT_TAIL] = rdp->nxttail[RCU_NEXT_READY_TAIL];
  rdp->nxttail[RCU_NEXT_READY_TAIL] = rdp->nxttail[RCU_NEXT_TAIL];

  /* Remember that we saw this grace-period completion. */
  rdp->completed = completed_snap;
}
```

Figure 3. Snippet of the `rcu_process_gp_end()` function.

However, if the thread waiting on the completion variable is not the only thread running on the CPU, this is unnecessary; these functions can be called from other threads running on the same CPU as well.

6. Reproducing an Older Kernel Bug

In Section 4.3 we mentioned that the grace-period kthread cleans up after grace-period ends. However, in older kernel versions, the RCU grace-period kthread did not exist; when a CPU entered the RCU core or invoked `call_rcu()`, it checked for grace-period ends by directly comparing the number of the last completed grace period in the `rcu_state` structure with the number of the last completed grace period in the respective `rcu_data` structure. In newer kernels, the `note_gp_changes()` function compares the number of the last completed grace period in the respective `rcu_node` structure with the number of the last completed grace period in the current `rcu_data` structure, *while holding the node's lock*, that way excluding concurrent operations on this node.

In kernel v2.6.32, commit `d09b62dfa336` fixed a synchronization issue exposed by unsynchronized accesses to the `->completed` counter in the `rcu_state` structure [27, 28], which caused the advancement of callbacks whose grace period had not yet expired. Below we will create a test case that shows such a situation, but this test case will also demonstrate that the problem is actually deeper: these unsynchronized accesses also lead to too-short grace periods.

To construct a test case that exposes this issue, we started by taking a look at the `rcu_process_gp_end()` function, since the issue was related to it. Figure 3 shows a relevant portion of its code. As can be seen, the access to the `->completed` counter is completely unprotected. So, we injected a `BUG_ON()` statement in the `if`-body to determine if it was possible for a thread to pick up the `->completed` value and then use the `completed_snap` while the `->completed` variable had changed. The answer was affirmative. Our next step was to determine if this could potentially lead to a CPU starting a new grace period without having noticed that the last grace period's ended. Again, an injection of a `BUG_ON()` statement, comparing the current grace period's number with the number of the grace period whose completion was noticed by the CPU, showed that this was possible.

0. `x = y = r_x = r_y = 0`
1. CPU0 has already registered a callback, started a grace period, passed through a quiescent state and reported it to RCU.
2. CPU1 sees that there is a grace period in progress (by taking a timer interrupt) and passes through a quiescent state, but does not report it to RCU yet.
3. CPU1 starts an RCU read-side critical section and executes `r_x = x`.
4. CPU1 takes a timer interrupt and enters RCU core (but does not end the grace period just yet).
5. CPU0 executes `x = 1` and then `synchronize_rcu()`. This call registers a callback by invoking `call_rcu()`. In kernel v2.6.31.1 and v2.6.32.1, `__call_rcu()` (invoked from `call_rcu()`) calls function `rcu_process_gp_end()`. So CPU0 calls `rcu_process_gp_end()`, but CPU1 has not ended the grace period yet. Hence, CPU0 sees that the current grace period is still in progress. Note that CPU1 had already started its RCU read-side critical section when CPU0 executed `synchronize_rcu()`.
6. CPU1 ends the first grace period and updates the `->completed` value. CPU1 does not need a new grace period, so it does not start one.
7. CPU0 checks whether a new grace period has started (by comparing `rdp->gpnum` with `rsp->gpnum`), but this start has not happened yet. This check occurs when CPU0 executes `rcu_check_for_new_grace_period()`.
8. CPU0 has registered callbacks and sees that no grace period is in progress (by checking `rcu_gp_in_progress() == 0`), starts a new grace period, and advances its callbacks from `NEXT` to `WAIT` segment. CPU0 has still not seen the previous grace-period ending.
9. CPU0 takes another timer interrupt and enters RCU core. It sees that a grace period has completed (via `rcu_process_gp_end()`) and advances its callbacks from `WAIT` to `DONE`. That means that the callback corresponding to `synchronize_rcu()` is ready to invoke, and CPU0 invokes it during the same interrupt.
10. CPU0 executes `y = 1`.
11. CPU1 executes `r_y = 1`.

At this point we conclude that the outcome `r_x == 0 && r_y == 1` is possible, something that violates the RCU Grace-Period guarantee, since the time between registration and invocation of a callback needs to span a grace period (and thus a full set of quiescent states), and this is not the case here.

Figure 4. Sequence of events resulting in an RCU Tree bug.

With these clues, we were able to construct a simple test case which proved that these unsynchronized accesses can lead to too-short grace periods. The test case has a reader seeing changes happening before the beginning of a grace period *and* after the end of the same grace period within a single RCU read-side critical section. In our test case there are three threads and two CPUs: `updater()` runs on CPU0 and `reader()` runs on CPU1. A thread running a `helper()` function represents a random thread running on CPU0 that can, potentially, occupy the CPU after `updater()` has blocked due to the invocation of `synchronize_rcu()`; it can be considered as a separate thread whose only purpose is to service an interrupt at CPU0. A sequence of events (produced by Nidhugg) for this test, which exposes this bug, is shown in Figure 4.

Let us end this section with some notes regarding this bug:

- The bug does not rely on interactions with the node hierarchy so it existed in both single-node and multi-level hierarchies. (A slightly different test case with the

respective `Kconfig` options set appropriately would be required for multi-level hierarchies.)

- Nidhugg reports that this bug is not present in Linux kernel v3.0, which means that it was indeed fixed. In kernel v3.0, `rcu_start_gp()` calls `__rcu_process_gp_end()`, thus guaranteeing that a CPU will see a grace-period ending before a grace-period beginning, something that does not happen in v2.6.32.1. However, the bug was present in previous versions as well, e.g., 2.6.31.1.
- Only two CPUs are required to provoke the bug, and only one of them has to invoke `call_rcu()`.
- Only one grace period is required to provoke the bug, meaning that it does not rely on CPUs being unaware of grace period ends and beginnings (e.g., when a CPU is in dynticks-idle mode). Of course, the latter implies that `CONFIG_NO_HZ=y/n` should not affect the test results. However, this bug *does* require some actions to occur during and after the ending of a grace period, meaning that a simple grace-period guarantee test would not have exposed this bug.
- `force_quiescent_state()` is not required to provoke the bug, although frequent calls to this function would expose it more easily in real-life scenarios.
- This bug is not caused by weak memory ordering; the test fails under sequential consistency.
- Nidhugg produced the sequence of events in Figure 4 in only $0.56s$ (compilation and Nidhugg transformation time included), and used 30.85MB of memory in total.

7. Tree RCU Validation

In this section we will mechanically validate the Grace-Period guarantee of Tree RCU for a non-preemptible Linux kernel environment, using the model we created in Section 5. We have applied this model to three different Linux kernels (v3.0, v3.19 and v4.3), and we were able to validate that the actual RCU code satisfies the Grace-Period guarantee under both SC and TSO, using a litmus test similar to the one in Figure 1.

7.1 Test Configuration

Let us first briefly discuss our modeling of the Linux kernel architecture on which the validation took place. All our experiments focused on the RCU-sched flavor of Tree RCU.

First of all, we model a system with two cores, represented by two mutexes, respectively. We also have three basic threads: the updater, the reader and the RCU grace-period kthread. The RCU-bh grace period kthread is disabled in order to reduce the state space, but it can be re-enabled by setting the `-DENABLE_RCU_BH` preprocessor option. We can assume that the updater and the RCU grace-period kthread run on the same CPU (e.g., CPU0), and that the reader runs on the other CPU (e.g., CPU1). We mention that different configurations were tried as well and they did not affect the outcome; this particular configuration was chosen because the updater and the RCU grace-period kthread take advantage of each other's context switches. Note that we could have ignored the RCU grace-period kthread and invoked `rcu_gp_init()` and `rcu_gp_cleanup()` appropriately, in order to further reduce the state space. However, this is an approximation and not the way the real kernel works, so we kept the RCU grace-period kthread in our modeling. For RCU initialization, the `rcu_init()` function is called. Since there are only two CPUs in our modeling, a single-node hierarchy is created. All CPUs start out idle (`rcu_idle_enter()` is called for each CPU), and `rcu_spawn_gp_kthread()` is called in order to spawn the RCU grace-period kthread.

Of course, interrupt context needs to be emulated as well. For this, we sprinkled calls to `do_IRQ()` in various points of the test code. In general, even though we do not care about the exact timing of interrupts, it is the occurrence of an interrupt within a specific context that causes a grace period to advance. Thus, in the current test, calls to `do_IRQ()` have been inserted into places that enable the advancement of a grace period. This may not always be the case (i.e. a grace period may not end for some explored executions), but in fact we want to allow both of these scenarios to happen. A set of per-CPU locks for interrupts (and interrupt disabling) and the approximation of interrupts with a separate thread might also be a good approach, but this makes the state space significantly larger.

7.2 Validating the Grace-Period Guarantee

All experiments have been run on a (quite outdated) standard desktop: a 64-bit machine with an Intel Core 2 Duo E8400 processor with 2GB of RAM running Debian Linux 3.16.0-4-amd64. After running the test with an unroll value in order for the test to be finite, Nidhugg reports that the test is successful for all three kernel versions. Moreover, despite running on a slow machine, the validation is quite fast. As shown on the first row of Table 1, the validation of the Grace-Period guarantee under SC requires about 6.5 minutes for kernel v3.0, 17.5 minutes for v3.19, and about 30.5 minutes for the v4.3 kernel. Another set of runs, validating this guarantee under the TSO memory model does not require considerably more time. Nidhugg tells us that there is no possible thread or memory model interleaving that violates the Grace-Period guarantee in Tree RCU's implementation.

But, can we really trust these results? After all, there might be a bug in our scaffolding of the Linux-kernel's environment, or there might be a bug in Nidhugg itself. In order to increase our confidence, we injected a number of bugs similar to ones that have occurred in real systems in production over the years. These bugs were added both in the test and the RCU source code. More specifically, we injected two kinds of bugs:

1. Bugs that make the grace period too short, thus permitting an RCU read-side critical section to span the grace period.
2. Bugs that prevent the grace period from ending.

Both kinds of bug injections represent RCU failures. Injections of the first kind result in a test failure, since the grace-period guarantee is violated. Injections of the second kind have to be used with an `assert(0)` statement after `synchronize_rcu()`. If this assertion does not trigger for *any* execution of the litmus test, then the grace period does not end for any execution, which in turn signifies that a successful —as opposed to a failed— completion of the test is a liveness violation.

Below, we present a list of preprocessor options that enable these scenarios, along with an explanation of each injection and the test outcome. For the `FORCE_FAILURE_6` test, an unroll value of 19 (`unroll=19`) has been used and `CONFIG_NR_CPUS` has been set appropriately, while for all other tests an unroll value of 5 has been used. All tests had the desired outcome, something that increases our confidence in our modeling and the validation result for the Grace-Period guarantee of Tree RCU's implementation that we report.

`-DASSERT_0`: An `assert(0)` statement is inserted after the function `synchronize_rcu()`. Obviously, this results in a test failure. What this assertion does, however, is that it shows that the grace period *can* end, and that there are *some* explored executions in which it does; i.e., it provides liveness guarantees. We will use this injection in conjunction with some of the next bug injections in order to determine whether the grace period ends or not.

`-DFORCE_FAILURE_1`: This injection forces the reader to pass through and report a quiescent state during its read-side critical section. Of course, this is not permitted and, as expected, results in a failure.

`-DFORCE_FAILURE_2`: A `return` statement is placed at the beginning of the `synchronize_rcu()` function. Of course, this results in a test failure since the updater does not wait for pre-existing readers to complete their RCU read-side critical sections, and such critical sections are not permitted to span a grace period.

`-DFORCE_FAILURE_3`: This injection makes `rcu_gp_init()` clear the `->qsmask` variables instead of setting them appropriately. The `rcu_gp_init()` function is invoked from the RCU grace-period kthread at the beginning of each grace period in order to initialize the grace period. Of course, since the `->qsmask` variables are cleared from the start of the grace period, the grace period can end immediately. In other words, the grace-period kthread does not wait for pre-existing readers to complete. (This can be considered a more complex variant of injection #2.) As expected, this injection results in a test failure.

`-DFORCE_FAILURE_4`: In this injection the `rcu_gp_fqs()` function is made to clear the `->qsmask` variables instead of waiting for the CPUs to clear their respective bits. Of course, in order for `rcu_gp_fqs()` to clear the `->qsmask` variables, the respective CPUs (in our case, the reader) has to be in dynticks-idle mode (or the CPU must have passed through a quiescent state at some point – the respective dynticks counters are sampled). Consequently, in our code, CPU0 calls the `rcu_gp_fqs()` function, and CPU1 enters and exits dynticks-idle mode within its RCU read-side critical section, which enables CPU0 to prematurely end the grace period. This can be considered an even more complex variant of injection #2, and results in a test failure, as expected.

`-DFORCE_FAILURE_5`: This injection makes the function `__note_gp_changes()` clear the respective `rnp->qsmask` bit for this CPU (`rnp->qsmask &= ~rdp->grpmask`). This function is called when a CPU enters RCU core in order to record the beginnings and ends of grace periods. However, instead of just recording a grace period beginning, `__note_gp_changes()` is now made to also clear the `rnp->qsmask` bit, which implies that this CPU reported a quiescent state for the new grace period. As expected, this injection results in test failure.

`-DFORCE_FAILURE_6`: Essentially, what this injection does is delete the `if` statement checking for zero `->qsmask` and calling `rcu_preempt_blocked_readers_cgp()`, in the `rcu_report_qs_rnp()` function. This `if` statement just checks whether the bitmask for this node is cleared in order for a node to acquire its parent's lock. In a real kernel, this should result in too short grace periods, since a signal that will prematurely awake the grace-period kthread is sent, if there are multiple CPUs. In our case, however, it does not lead to too-short grace periods since, in our modeling, `wake_up()` boils down to a no-op – there is no need to wake up someone who is just spinning. However, if we were dealing with a two-level tree, the caller of `rcu_report_qs_rnp()` would move up one level and trigger a `WARN_ON_ONCE()` statement that checks whether the child node's bits are cleared. Hence, this test automatically sets the number of CPUs to `CONFIG_RCU_FANOUT_LEAF + 1` (i.e. to 17, since the default value of `CONFIG_RCU_FANOUT_LEAF` is 16 in these kernels). Also, this test requires the use of a higher unroll value because there are some loops that need to be unrolled at least as many times as the number of CPUs used plus one. So, we used an unroll value of 19 for this case.

`-DLIVENESS_CHECK_1`: This sets `rdp->qs_pending` to zero in the `__note_gp_changes()` function. This function updates the per-CPU `rcu_data` structure. So, since `rdp->qs_pending` is set to zero, there is no need for a CPU to report a quiescent state to RCU, which prevents grace periods from completing. When the injection is used in conjunction with `-DASSERT_0`, no execution triggers the `assert(0)` statement after `synchronize_rcu()`, which, as mentioned, signifies a liveness violation.

`-DLIVENESS_CHECK_2`: A `return` statement is placed at the beginning of the `rcu_sched_qs()` function. In effect, this means that CPUs cannot record in their `rcu_data`

Table 1. Results for Tree RCU litmus test on three Linux kernel versions (time in seconds, memory in MB).

Preprocessor Options	v3.0					v3.19					v4.3				
	SC		TSO		*Traces*	SC		TSO		*Traces*	SC		TSO		*Traces*
	Time	*Memory*	*Time*	*Memory*	*Explored*	*Time*	*Memory*	*Time*	*Memory*	*Explored*	*Time*	*Memory*	*Time*	*Memory*	*Explored*
-	385.29	33.11	436.48	33.52	19398	1051.83	64.34	1130.94	64.84	24760	1838.01	102.78	1841.20	101.78	28996
-DASSERT_0	3.41	32.22	3.75	32.64	145	2.08	34.42	2.29	34.40	37	2.65	37.72	3.26	37.41	29
-DFORCE_FAILURE_1	3.52	32.35	3.72	32.44	146	2.20	34.43	2.55	34.37	41	3.32	37.41	3.50	37.59	33
-DFORCE_FAILURE_2	0.55	32.51	0.60	32.51	4	0.80	34.24	0.89	34.28	3	1.44	37.75	1.47	37.53	3
-DFORCE_FAILURE_3	48.91	32.50	47.70	32.37	2372	567.86	47.92	612.75	49.74	13264	510.40	57.77	558.21	56.74	8114
-DFORCE_FAILURE_4	2.03	32.55	2.20	32.59	84	4.40	34.50	4.97	34.54	79	2.97	37.46	3.25	37.45	24
-DFORCE_FAILURE_5	87.27	32.41	96.62	32.45	4888	1.34	34.62	1.46	34.86	9	1.91	37.74	2.07	37.47	9
-DFORCE_FAILURE_6	1.66	32.75	1.78	32.97	1	4.58	109.75	4.93	109.87	2	7.69	232.30	7.79	232.17	2
-DLIVENESS_CHECK_1 -DASSERT_0	36.22	32.68	40.93	32.60	2024	15.62	34.38	17.08	34.29	608	16.84	37.77	18.31	37.57	488
-DLIVENESS_CHECK_2 -DASSERT_0	71.00	32.38	78.01	32.38	3888	15.52	34.82	17.09	34.82	608	18.64	37.72	20.52	37.57	516
-DLIVENESS_CHECK_3 -DASSERT_0	39.33	32.38	41.86	32.56	2184	17.70	34.52	21.26	34.68	688	16.97	37.78	18.42	37.72	488

structures their passing through a quiescent state, which also prevents grace periods from completing. Used in conjunction with `-DASSERT_0` this bug injection also results in no executions triggering the assertion, thus signifying a liveness violation.

`-DLIVENESS_CHECK_3`: A `return` statement is placed at the beginning of `rcu_report_qs_rnp()`. This means that CPUs cannot report their passing through a quiescent state to RCU, which in turn means that a grace period cannot complete. This injection also needs to be used together with `-DASSERT_0` to discover the liveness violation.

7.3 Results and Discussion

In Table 1, the "Time" columns represent the total wall-clock time in seconds (compilation and Nidhugg transformation time are included). As can be seen, there is very little overhead when going from SC to TSO, which shows the power of stateless model checking with source DPOR [1] and chronological traces [2] in this setting. In fact, we show only one "Traces Explored" column for each kernel, since in all tests the total number of explored executions is the same for both SC and TSO. The reason for that is that there are a lot of memory fences in the code of Tree RCU, which prevent store buffer reorderings from happening. But even if reorderings were possible, all bug injections here do not rely on the employed memory model but instead violate the assertions algorithmically.

As expected, since the model checking is stateless, the memory requirements are very low, esp. considering the size of the source code which is tested. The only case that sticks out is `FORCE_FAILURE_6` that requires more memory due to using a higher unroll value (`unroll=19`) and an increased number of RCU data structures (`CONFIG_NR_CPUS=17`).

The most interesting row here is the first one, shown in green. Here Nidhugg explores the complete set of traces; in all other cases a test failure is detected and exploration stops at that point. How fast this happens depends on the order in which traces are explored. In some cases errors are detected immediately (in the first traces and in less than two seconds) and in other cases only after many traces have been explored.

It can also be observed that newer kernel versions, which support more features and have greater code complexity, require more computational power in terms of time and memory. However, Nidhugg copes very well with the extra complexity, and the increase when going from v3.0 to v3.19, or from v3.19 to v4.3 is not that big. Note that, in kernel v3.0, the `-DFORCE_FAILURE_3` and `-DFORCE_FAILURE_5` injections are transformed into liveness checks, due to the absence of the grace-period kthread, which explains the increased times in these tests. Also, the `-DFORCE_FAILURE_6` injection in v3.0 does not consume much memory since only one interleaving is explored in order for an assertion to be violated, and Nidhugg does not need to allocate extra memory. A more fair comparison would be the one between v3.19 and v4.3, due to the similarity in their implementation. Still, overall there is a noticeable increase, in terms of time and memory, as the kernel grows. This very fact underlines the need for efficient and scalable testing and validation tools. Our experience with Nidhugg is very positive in this respect.

8. Related Work

Previous work on RCU verification includes the expression of RCU's formal semantics in terms of separation logic [17] and the verification of user-space RCU in a logic for weak memory [32]. A virtual architecture to model out-of-order memory accesses and instruction scheduling has been proposed [9], and a verification of user-space RCU has been done using the SPIN model checker [10]. Moreover, researchers at Stony Brook University produced an RCU-aware data race detector [11, 30]. Alglave *et al.* verified that RCU's *actual* kernel code preserves data consistency of the object it is protecting [4] using CBMC [8]. Subsequently, McKenney [21] verified the Grace-Period guarantee for Tiny RCU (a flavor of RCU for uniprocessor systems). Finally, mutation testing strategies have been applied to RCU's code [3] as well.

Concurrently with our work, Liang *et al.* used CBMC to verify the Grace-Period guarantee for Tree RCU [18]. However, compared to the work presented here, their approach has some limitations. First of all, due to CBMC's limited support for lists, their modeling does not include callback handling. This has some implications for verification. The most basic one is that bugs in the callback handling mechanism (e.g., a bug similar to the one we reproduced in Section 6) can not be exposed. Considering the fact that RCU's update side primitives are based on callback handling, this limitation is serious. For example, primitives like `call_rcu()`

were not included in the tests, and `synchronize_rcu()`'s implementation (which, in reality, is based on `call_rcu()`) has to be emulated. This in turn means that only the underlying grace period mechanism was modeled; not the callback mechanism that mediates between that mechanism and `synchronize_rcu()`. A second limitation is that the grace-period kthread was not included in the tests. Although in older kernel versions the grace-period kthread did not exist, for newer Linux kernels excluding the kthread from the tests implies alteration of the kernel's operation. In addition, this thread's exclusion means that the way a grace period started and ended also needs to be changed, since the grace-period kthread plays a crucial role in these operations. Finally, the approach of Liang *et al.* does not include the emulation of dynticks-idle mode. In our approach, the dynticks-idle mode is indeed modeled, and our results show that the basic properties of the `dyntick` counters do hold.

Despite the simpler modeling and these limitations, [18] reports that CBMC needs more than 11 hours and 34GB of memory in order to claim successful verification for Tree RCU in kernel v4.3 under TSO, whereas Nidhugg only needs 30.5 minutes and 102MB of memory. More generally, our results are orders of magnitude faster, which we attribute to the different algorithms that the two tools employ.

On the other hand, it should be stated that CBMC's underlying algorithm in principle also handles data non-determinism, something that stateless model checking tools in general (and Nidhugg in particular) do not consider. Still, we do not see how data non-determinism plays any role in validating the Grace-Period guarantee of Tree RCU. Some supporting evidence for this claim offers the fact that the bug injections we listed in Section 7 are a proper superset of those identified by CBMC.[2] Furthermore, because our approach does include callback handling, we were able to reproduce an older, real kernel bug that was caused by premature callback invocations, which could potentially lead to too short grace periods that violate the Grace-Period guarantee. As explained, this bug can not be reproduced with CBMC, due to CBMC's limited support for lists.

9. Concluding Remarks

We described a way to construct a test suite for the systematic concurrency testing of Linux kernel's RCU mechanism. For this, we emulated a non-preemptible Linux-kernel SMP environment and managed to validate the most basic guarantee that is provided by Tree RCU, the main RCU flavor used in the Linux kernel.

More specifically, using the stateless model checking tool Nidhugg we validated the Grace-Period guarantee for three different kernel versions (v3.0, v3.19, and v4.3), under both a sequentially consistent and a TSO memory model. For all our tests we used the source code from the Linux kernel directly, with only a handful of changes, which can be scripted.

To show that our emulation of the kernel's environment is sound and to further strengthen our results, we injected RCU failures in our tests, inspired from real bugs that occurred throughout RCU's deployment in production, and Nidhugg was able to identify them all. Moreover, we demonstrated that a patch that applied a well-defined locking design to a variable in two older kernels [28] resolved a much more complex issue that was in effect a bug. We identified and reproduced this bug, providing the exact circumstances under which it occurred. In addition, we tested whether the bug exists in later kernel versions and the answer was negative.

Our work demonstrates that stateless model checking tools like Nidhugg can be used to test *real* code from today's production systems with large codebases. The small time and memory consumption of our tests, especially considering the size and the dynamic nature of the codebase tested, underlines the strength of our approach. All the above, along with the fact that our model of the kernel's environment was reused across different kernel versions show that stateless model checking tools can be integrated in Linux kernel's regression testing, and that they can produce useful results.

Still, we are not yet at a point where we can claim with certainty that the complete implementation of Tree RCU is bug-free; there may be bugs in components of Tree RCU that are not included in our modeling and our tests. In addition, there are are many other requirements that RCU must meet. Thus, our work could be extended to include more aspects of RCU, and test them under different memory models (e.g., POWER). For example, we could construct tests that include quiescent-state forcing, grace-period expediting and CPU hot plugs. The same applies for the full-dynticks mode which was fully merged in the kernel only relatively recently. On a separate note, the approximation of interrupts with separate threads is also an interesting approach. Last but not least, the scalability of our results renders the construction of test cases aiming at the thorough testing of the preemptible Tree RCU extremely interesting as well.

[2] Injections `-DFORCE_FAILURE_1` and `-DFORCE_FAILURE_4` are not considered by Liang *et al.* [18]; the latter due to not modeling the dynticks-idle mode.

Acknowledgments

We are very much obliged to Paul E. McKenney for all his help, advice, and suggestions throughout this effort. His profound insight into RCU was extremely helpful in numerous occasions. Also, this paper would not have been possible without Nidhugg, whose main developer, Carl Leonardsson, deserves a big thanks for all the hard work he has put into it.

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 373–384, New York, NY, USA, 2014. ACM.

[2] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *LNCS*, pages 353–367, New York, NY, USA, 2015. Springer.

[3] I. Ahmed, A. Groce, C. Jensen, and P. E. McKenney. How verified is my code? Falsification-driven verification. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 737–748, 2015.

[4] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.

[5] 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, Sept. 2015.

[6] M. Christakis, A. Gotovos, and K. Sagonas. Systematic testing for detecting concurrency errors in Erlang programs. In *Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, pages 154–163, Los Angeles, CA, USA, 2013. IEEE Computer Society.

[7] E. M. Clarke, O. Grumberg, M. Minea, and D. A. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[8] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[9] M. Desnoyers, P. E. McKenney, and M. R. Dagenais. Multi-core systems modeling for formal verification of parallel algorithms. *SIGOPS Oper. Syst. Rev.*, 47(2):51–65, July 2013.

[10] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of Read-Copy Update. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):375–382, Feb. 2012.

[11] A. Duggal. *Stopping Data Races Using Redflag*. PhD thesis, Stony Brook University, 2010.

[12] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 110–121, New York, NY, USA, 2005. ACM.

[13] Built-in functions for memory model aware atomic operations. https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html.

[14] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. PhD thesis, University of Liège, 1996. Also, volume 1032 of LNCS, Springer.

[15] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 147–186. ACM, 1997.

[16] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[17] A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with Grace. In *Programming Languages and Systems*, volume 7792 of *LNCS*, pages 249–269, Berlin, Heidelberg, 2013. Springer.

[18] L. Liang, P. E. McKenney, D. Kroening, and T. Melham. Verification of the tree-based hierarchical Read-Copy Update in the Linux kernel. CoRR abs/1610.03052, Oct. 2016.

[19] LLVM atomic instructions and concurrency guide. http://llvm.org/docs/Atomics.html#libcalls-atomic.

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

[21] P. E. McKenney. Verification challenge 4: Tiny RCU. http://paulmck.livejournal.com/39343.html, Mar. 2015.

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

[23] P. E. McKenney and J. Walpole. What is RCU, fundamentally? http://lwn.net/Articles/262464/, Dec. 2007.

[24] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[25] D. Peled. All from one, one for all: On model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification*, LNCS, pages 409–423, London, UK, UK, 1993. Springer-Verlag.

[26] A. Prakash. Linux now runs on 99.6% of top 500 supercomputers. https://itsfoss.com/linux-99-percent-top-500-supercomputers/, Nov. 2016.

[27] RCU: Clean up locking for `->completed` and `->gpnum` fields. https://lkml.org/lkml/2009/10/30/212.

[28] RCU: Fix synchronization for `rcu_process_gp_end()` uses of `->completed` counter. https://lkml.org/lkml/2009/11/4/69.

[29] RCU Linux kernel documentation. https://www.kernel.org/doc/Documentation/RCU/.

[30] J. Seyster. *Runtime Verification of Kernel-Level Concurrency Using Compiler-Based Instrumentation*. PhD thesis, Stony Brook University, Dec. 2012.

[31] Sparse - a semantic parser for C. https://sparse.wiki.kernel.org/index.php/Main_Page.

[32] J. Tassarotti, D. Dreyer, and V. Vafeiadis. Verifying Read-Copy-Update in a logic for weak memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 110–120, New York, NY, USA, 2015. ACM.

[33] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pages 491–515, London, UK, UK, 1991. Springer-Verlag.