International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-019-00514-6

SPIN 2017

Stateless model checking of the Linux kernel's read–copy update (RCU)

Michalis Kokologiannakis[1] · Konstantinos Sagonas[2]

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 a 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 increasingly more complex and are employed in machines with more and more cores and weak memory does not make the situation any easier. This article presents an approach to systematically test the code of the main implementation of RCU used in the Linux kernel (Tree RCU) for concurrency errors, both under sequentially consistent 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. Additionally, we present the real cause behind some failures that have been observed in production systems in the past. More importantly, we were able to verify both the publish–subscribe and the grace-period guarantee, with the latter being the basic and most important guarantee that RCU offers, on several Linux kernel versions, for particular configurations. Our approach is effective, both in dealing with the increased complexity of recent Linux kernels and in terms of time that the process requires. We hold that our effort constitutes a good first step toward making tools such as Nidhugg part of the standard testing infrastructure of the Linux kernel.

Keywords Software model checking · Linux kernel · Read–copy update · Nidhugg

1 Introduction

The Linux kernel is used in a surprisingly large number of devices: from PCs and servers to routers and smart TVs. For example, in 2015, more than one billion smart phones used a modified version of the Linux kernel [1] and, in 2017, all modern supercomputers used Linux as well [2]. Now, with many IoT devices shifting their operating system to a Linux-based one [3], this number is only bound to increase. Therefore, it is self-evident that the correct and reliable operation of the Linux kernel is of great importance, which renders thorough testing and verification of its components a necessity.

✉ Michalis Kokologiannakis
michalis@mpi-sws.org

Konstantinos Sagonas
kostis@it.uu.se

1 Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern Saarbrücken, Germany

2 Department of Information Technology, Uppsala University, Uppsala, Sweden

Naturally, this process needs to span all of the kernel's components and subsystems. One particular subsystem with a non-trivial implementation is the read–copy update (RCU) mechanism [4,5]. RCU is a synchronization mechanism that provides excellent scalability by enabling concurrent reads and updates. RCU's implementation is quite involved, as RCU interacts with many other subsystems of the Linux kernel, making the precise modeling of RCU's environment arduous. Moreover, the lockless design of its fastpaths, and the fact that it needs to operate in heavily concurrent environments, renders the modeling and verification process even more challenging. The relatively short release cycle of the Linux kernel (currently, there is a new release every approximately 2 months), the number of changes that are involved in each release, and the increasing complexity of the kernel's code call for thorough and automatic testing. In fact, the Linux code base already contains a fair number of regression test suites, including a so-called torture test suite for its RCU component [6]. 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 bug-finding techniques, such as software model checking, that are able to operate on as big a percentage of the *actual code* of the Linux kernel as possible.

This article reports on the use of stateless model checking (also known as systematic concurrency testing) for testing the core of Tiny RCU and Tree RCU, both being RCU implementations used in the Linux kernel.

First, after a brief introduction to RCU (Sect. 2) and stateless model checking (Sect. 3), we demonstrate how we used the tool Nidhugg [7] to verify the *grace-period guarantee*, which is the basic guarantee that RCU offers, for Tiny RCU, an implementation of RCU for uniprocessor systems (Sect. 4).

Next, after describing the implementation of Tree RCU (Sect. 5), we show how we obtained a reusable model for the kernel's environment (Sect. 6), necessary for the verification procedure. Using this model, as well as the source code from five different kernel versions directly, we verified both a part of the *publish–subscribe guarantee* (Sect. 7) and the *grace-period guarantee* (Sect. 8) for Tree RCU, the main RCU implementation used in the Linux kernel. Our effort concentrated on particular kernel configurations (non-preemptible builds), but we also investigated the effects that weak memory models (TSO and PSO) may have on RCU's operation.

In order to strengthen our verification claim, we injected concurrency bugs similar to ones that have been reported 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 also able to demonstrate that a submitted patch, intended to impose a locking design, in reality fixed a much more serious bug that was responsible for failures observed in production systems some years back, a fact that was previously unknown. We report on this issue and present the exact conditions under which this bug occurs (Sect. 9).

Finally, we discuss some limitations of our approach, as well as the main lessons learned from the verification procedure (Sect. 10). As demonstrated by our results, our technique handles real code employed in today's production systems in an efficient and scalable way, especially compared to other tools (Sect. 11), raising hopes regarding the inclusion of a stateless model checking tool such as Nidhugg in the standard testing infrastructure of the Linux kernel.

This article is the journal version of a conference paper [8] that has appeared in the proceedings of SPIN 2017. Compared with that paper, this article contains the following additional material:

- Stateless model checking is described in more detail, giving insights on how tools such as Nidhugg operate.
- More intuition on the kernel's environment modeling is provided, via the case study of Tiny RCU.
- The publish–subscribe guarantee is described and verified.
- The grace-period guarantee is verified also under the PSO memory model.
- More details regarding the reproduced bug are given.
- The limitations of our approach are discussed, and more details regarding the model have been added.

2 Read–copy update (RCU)

Read–copy update is a synchronization mechanism invented by McKenney and Slingwine [4,5] that is a part of the Linux kernel since 2002. The key feature of RCU is the good scalability it provides by allowing concurrent reads and updates. While this may seem counter-intuitive or even 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, which are the ones we focus on this work, RCU imposes zero overhead to readers.

2.1 How RCU works

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 [4].

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.

3. Wait until all pre-existing readers complete their RCU read-side critical section, so that there is 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.

Note that steps 2 (the removal phase) and 4 (the reclamation phase) in the above procedure are not necessarily performed by the same thread.

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 1 (*Quiescent state*) 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 the Linux kernel, this is checked with the tool sparse [9].) Note that different RCU flavors have different sets of quiescent states.

Definition 2 (*Grace period*) 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 GP, it would have to complete before the end of GP. 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.[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 function `rcu_read_lock()` and end with `rcu_read_unlock()`. These functions do not block or spin, and in non-preemptible kernels, they are effectively no-ops.

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

void reader(void)            void updater(void)
{                            {
  rcu_read_lock();             WRITE_ONCE(x, 1);
  r_x = READ_ONCE(x);          synchronize_rcu();
  r_y = READ_ONCE(y);          WRITE_ONCE(y, 1);
  rcu_read_unlock();         }
}
```

Fig. 1 RCU's grace-period guarantee litmus test

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) before that grace-period ends. This very fact allows RCU verification to be focused; every correct implementation has to adhere to the following rule:

> If any statement in a given RCU read-side critical section CS precedes a grace period GP, then all statements (including memory operations) in CS 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.

In order to see what this guarantee really implies, consider the code fragment in Fig. 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. The description of how this guarantee is achieved though is deferred to Sects. 4 and 5.

Publish–subscribe guarantee This guarantee is used in order to coordinate read-side accesses to data structures. The publish–subscribe mechanism is used for data insertion into data structures (e.g., lists), without disrupting possible concurrent readers. Since updaters run concurrently with readers, this mechanism should ensure two things: first, that updaters will have completed all initialization operations before publishing a data structure, and second, that readers will not see uninitialized data. Note that the latter may occur even if the

[1] Requirements for RCU appear at: https://www.kernel.org/doc/Documentation/RCU/.

```
Initially: struct foo {int a; int b;} *gp = NULL;
```

```
void publisher(void)
{
  struct foo *p;

  p = kmalloc(...);
  p->a = 42;
  p->b = 42;
  rcu_assign_pointer(gp, p);
}
```

```
void subscriber(void)
{
  struct foo *p;

  rcu_read_lock();
  p = rcu_dereference(gp);
  if (p) {
    /* access p->a, p->b */
    rcu_read_unlock();
    return;
  }
  rcu_read_unlock();
  return;
}
```

Fig. 2 RCU's publish–subscribe guarantee litmus test

Initially: `x = y = 0`
spawn threads p and q;

p	q
`x := 1;`	`x := 2;`
`y := 1`	`y := 2`

join p and q;
`assert(x == y)`

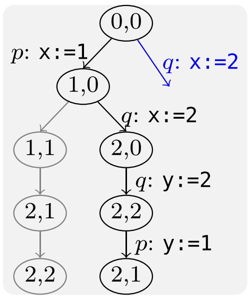

Fig. 3 A concurrent program (its correctness property as assertion) and two of its interleavings. Shaded nodes are states that are forgotten when exploring the second interleaving. The blue edge shows the first step of the next interleaving that could be explored, if needed

updaters publish a data structure after all initializations have completed and are visible to other CPUs (e.g., by a compiler that does value-speculation optimizations, or by a CPU that reorders dependent loads).

In order to achieve this, RCU offers two primitives: `rcu_assign_pointer()` and `rcu_dereference()`. The primitive `rcu_assign_pointer()` has similar semantics to C11's `memory_order_release` operation. In effect, it is similar to an assignment but also provides additional ordering guarantees. The second primitive, `rcu_dereference()`, on the other hand, can be considered as a subscription to a value of a specified pointer and 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 to provide the aforementioned guarantee.

Consider the code fragment in Fig. 2, which constitutes a classic publish–subscribe scenario. In this example, it is guaranteed that the subscriber will not see uninitialized values for the field values of `p`, i.e., that the outcome:

```
p->a != 42 || p->b != 42                                  (2)
```

is impossible.

3 Stateless model checking

Stateless model checking (SMC) [10], also known as systematic concurrency testing, is a testing and verification technique with low memory requirements that is applicable to programs with executions of finite length. Stateless model checking tools explore the executions of a program without explicitly storing the executions they have previously visited. The technique has been successfully implemented in tools such as VeriSoft [11], CHESS [12], Concuerror [13], Nidhugg [7], rInspect [14], CDSChecker [15], and RCMC [16].

At least conceptually, the technique that SMC tools employ is very simple. Given an entry point to a concurrent program, whose code possibly contains some assertions that express its correctness properties, an SMC tool takes control of the scheduler and systematically explores all different ways that its threads can be interleaved.

For example, consider the program shown in Fig. 3 in which a *main* thread spawns two concurrent threads, p and q, which issue write operations on two different shared variables `x` and `y` whose initial value is 0. From the initial state $(0, 0)$, an SMC tool could start by exploring the interleaving where the two steps of thread p are performed first, followed by the two steps of thread q, thereby reaching state $(2, 2)$ that satisfies the assertion. The second interleaving which is explored could be the one in which thread p is preempted after its first step, and thread q executes at that point. In this interleaving, execution reaches final state $(2, 1)$ in which the assertion is violated. At this point, the exploration has detected a concurrency error and can stop. In contrast, had the assertion been e.g., `assert(abs(x - y) < 2)`, which would not be violated by the program, then the exploration would need to examine all interleavings that reach different states. This is a general phenomenon in stateless model checking: errors are typically detected relatively fast, but verification needs to explore the complete search space and thus can take considerably longer than bug finding.

To combat the combinatorial explosion in the number of interleavings that need to be examined in order to maintain full coverage of all program behaviors, SMC tools use *partial-order reduction* [17–19] 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) algorithms capture dependencies (conflicts) between steps of concurrent threads, while the program is running [20,21]. Each interleaving which is explored is used to identify dependent operations and program 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 effects of architectural or programming language memory models [7,14–16] in addition to scheduling non-determinism. Nidhugg [7], for example, the tool we have employed, 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 partially, POWER. Nidhugg works on the level of LLVM intermediate representation and, at the time when this worked was performed, employed an effective dynamic partial-order algorithm called source-DPOR [21,22]. We note in passing that Nidhugg has since been extended with a wider selection of DPOR algorithms: optimal-DPOR [22] and optimal-DPOR with observers [23].

However, in stateless model checking, all tests need to be *data-deterministic* and *finite*. Data determinism means that, in a given state, a given execution step must always lead the system to the same new state, i.e., 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). Finiteness means that, for all test cases, there must be a bound $n \in \mathbb{N}$ such that all executions of the program terminate within n execution steps. In Nidhugg, loops that may in principle execute for an unbounded number of times (e.g., spin loops for locks) are either automatically transformed to `assume()` statements or need to become bounded by using an appropriate `unroll=`n option, which makes Nidhugg not consider executions in which any loop performs more than n iterations.

4 Stateless model checking Tiny RCU

Now that we have laid down the basic foundations of RCU and stateless model checking, let us proceed with the verification of Tiny RCU [24], an implementation of RCU designed to run on uniprocessor systems, targeting mostly embedded applications. Although this is not the first verification attempt for Tiny RCU (cf. Sect. 11), it serves as a good first step toward understanding how a simple RCU implementation looks like and how it can be verified, while also providing a very good intuition on how the Linux kernel's environment can be modeled.

4.1 Tiny RCU implementation

Tiny RCU is only offered for non-preemptible uniprocessor systems.[2] This means that the definition of a grace period for Tiny RCU can be formulated as follows [24]:

> Whenever the sole CPU of the system passes through a quiescent state, a grace period has elapsed.

This greatly simplifies the design of Tiny RCU's implementation and renders the memory footprint of Tiny RCU much smaller than Tree RCU's, since Tiny RCU requires much simpler RCU-related data structures.

Now the only question that needs to be answered is *how* RCU knows that a processor has passed through a quiescent state. For that, Tiny RCU relies on context switches, scheduling-clock interrupts and idle mode, just like Tree RCU (albeit in a slightly different way). For example, every time the sole CPU takes a scheduling-clock interrupt, RCU checks whether the CPU is in a quiescent state (e.g., in user-mode execution) and, if that is the case, marks the pending callbacks of the CPU (that have been registered via `call_rcu()`) as ready to invoke. These will be executed at some later point (not within the interrupt handler), when it is safe to do so. Of course, this is just a high-level view of how callbacks are handled, and a lot of details are omitted (e.g., how RCU knows which callbacks are ready to be invoked). We will refrain from presenting these details here and defer the discussion regarding the callback handling mechanism to Sect. 5.

Instead, let us focus on `synchronize_rcu()`, the implementation of which, unlike Tree RCU, is not based on callbacks. First, it is important to note that, in general, it is illegal to invoke `synchronize_rcu()` within RCU read-side critical sections, as in that case an updater would wait for itself to finish its read-side critical section, which would in turn lead to a deadlock. Consequently, and by also taking into account the fact that read-side critical sections cannot be preempted, every time `synchronize_rcu()` is called, the CPU is in a quiescent state. This in turn means that `synchronize_rcu()` can actually return immediately. In fact, this is the way this function is implemented[3] for Tiny RCU in recent kernels (following release v4.9.6).

In a similar way, it is easy to see why `rcu_read_lock()` and `rcu_read_unlock()` are effectively no-ops for Tiny RCU, since no particular actions are needed from the readers' side, apart from using `rcu_assign_pointer()` and `rcu_dereference()`, of course.

4.2 Kernel environment modeling

Now that we laid out the basic details of Tiny RCU's implementation, a question that naturally arises is how one can model the various subsystems that RCU interacts with, in order to verify that implementation.

Suppose that we want to use a litmus test similar to the one of Fig. 1 as part of the verification of the grace-period guaran-

[2] There also used to be a version of Tiny RCU for preemptible kernels [25].

[3] Disregarding some deadlock checks that are performed.

tee for Tiny RCU. For that, we need at least one reader and one updater (two distinct threads) that will run on the sole CPU of the system. However, Nidhugg, given two threads, will systematically explore all possible interleavings between these two threads, disregarding the mutual exclusion the processor of the system imposes. Thus, we need to somehow restore this mutual exclusion between the threads. The solution is, of course, to use a mutex in order to emulate a CPU: a thread has to acquire the CPU's lock in order to run on the CPU, and it releases the lock when it (voluntarily) yields. Similarly, functions that call the scheduler (e.g., `cond_resched()`) can be modeled as having a thread drop the CPU's lock and then (possibly) re-acquire it. Then, Nidhugg will take care of exploring all the possible schedulings between the two threads, while respecting the mutual exclusion enforced by the CPU. As far as other kernel primitives are concerned, the definition of these were either copied directly from the kernel or emulated where copying was not possible (e.g., `cond_resched()`).

Of course, there are other things that one should take care of (e.g., interaction with dyntick-idle mode,[4] interrupts). However, the purpose of this section is only to establish a basic intuition regarding how the kernel's environment can be modeled, and not describe that modeling in full detail; this is deferred for Sect. 6.

4.3 Results

Using the model we constructed, we verified the grace-period guarantee for Tiny RCU, based on litmus tests similar to the one of Fig. 1. Due to the simplicity of Tiny RCU's implementation, Nidhugg only needed 0.08 s to run our tests. We only used kernel v3.19 for our tests, since Tiny RCU is not so interesting compared to Tree RCU, and its implementation does not change so often.

In addition, we tried other scenarios where bugs were injected in the code (e.g., having the reader to yield in the middle of its critical section), to determine whether Nidhugg would be able to detect those bugs, and, as expected, the answer was affirmative. Nidhugg provided us with the relevant traces that triggered these bugs.

As a last note, although we did not perform any tests that involved callback handling for Tiny RCU, doing so would not be hard; in fact, we did so for Tree RCU, but more on that in the next sections.

[4] The Linux kernel's dynticks-idle mode [26] (a.k.a. NO_HZ) is a mode in which a CPU is idle and scheduling-clock ticks are turned off, in order to promote energy saving.

5 Tree RCU implementation

The Linux kernel offers many different RCU implementations, each one serving a different purpose. The first Linux kernel RCU implementation 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 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 [26] CPUs. It can easily scale to thousands of CPUs, while Classic RCU could only scale to several hundred. Apart from the original Tree RCU implementation, several flavors of Tree RCU are provided [27], for example:

RCU-sched, where anything that disables preemption acts as an RCU read-side critical section. This is useful if code segments with preemption disabled need to be treated as explicit RCU readers.

RCU-bh, where RCU read-side critical sections disable softirq processing. This is useful if grace periods need to complete even when softirqs monopolize one or more of the CPUs (e.g., if the code is subject to network-based denial-of-service attacks).

Sleepable RCU (SRCU), which is a specialized RCU version that permits general sleeping in RCU read-side critical section.

In this article, we focus on the original Tree RCU implementation, which is the same as RCU-sched in non-preemptible builds.

Below we present a high-level explanation of Tree RCU along with some implementation details, a brief overview of its data structures, and some use cases that are helpful in understanding how RCU's fundamental mechanisms are actually implemented.

5.1 High-level explanation

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

Tree RCU avoids this performance and scalability bottleneck 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

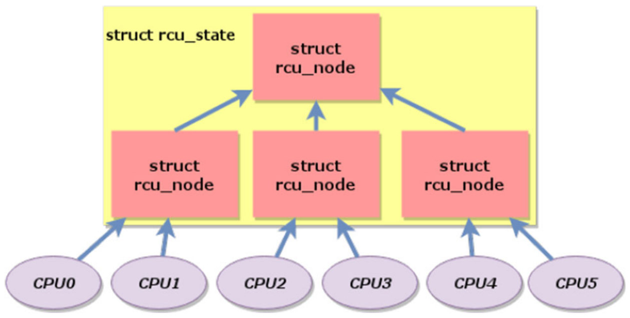

Fig. 4 Tree RCU node hierarchy (adapted from [28])

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. A simple node hierarchy for a 6-CPU system is presented in Fig. 4.

As can be seen in the figure, CPUs 0 and 1 will acquire the lower-left node's lock, CPUs 2 and 3 will acquire the lower-middle node's lock, and CPUs 4 and 5 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 maximum number of CPUs contending for a leaf-node's lock. Default value is 16.
`CONFIG_RCU_FANOUT`: Controls 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.

5.2 Data structures

Let us now present three major data structures (`rcu_data`, `rcu_node`, and `rcu_state`) of Tree RCU's implementation.

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` structure, which includes, 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 Boolean variable indicating whether this CPU has passed through a quiescent state for this grace period;
- a pointer to this CPU's leaf of hierarchy; and
- the mask that will be applied to the leaf's mask (`grpmask`).

Of course, when a CPU registers a callback, this is also stored in the respective per-CPU data structure.

Then, 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:

- 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`);
- a pointer to the node's parent;
- the mask that will be applied to parent node's mask (`grpmask`); and
- the number of the lowest and the highest CPU or group for this node.

Lastly, the RCU global state and 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 other `Kconfig` options. (Note that small systems have a hierarchy consisting of a single `rcu_node`.) The `rcu_state` structure contains, among others:

- the node hierarchy;
- a pointer to the per-CPU `rcu_data` variable;
- the current grace-period number; and
- 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 often the discovery of bugs that led to such changes in the source code.

Finally, we have already mentioned that Classic RCU had a suboptimal 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; and
- 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.

5.3 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 hotplugs [29], CPUs in user-land, and CPUs that fail to report a quiescent state to RCU within a reasonable amount of time. In the next subsections, we will discuss some use cases of RCU, as well as the interaction of RCU with the described data structures, and the functions involved.

5.3.1 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. More information regarding the callback list and its structure can be found in RCU's documentation [30].

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 Linux 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[5]) and will start a grace period if there is need for one (e.g., if there is no grace period in progress and this 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.

The `synchronize_rcu()` function, which is implemented on top of `call_rcu()` in Tree 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.

5.3.2 Starting a grace period

The `rcu_start_gp()` function 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 first 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. However, 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 `rcu_process_callbacks()`).

The grace-period kthread resolved many races present in older kernels. For example, races occurring when CPUs requiring a new grace period were trying to directly initialize the node hierarchy, something that can potentially lead to bugs; see Sect. 9.

5.3.3 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 function `rcu_check_callbacks()`, and from the `ksoftirq/n` kthreads. However, the fact that a CPU has passed through a quiescent state does not

[5] `ksoftirq/n` kthreads are special per-CPU kernel threads that run when the machine is under heavy softirq load.

mean that RCU knows about it. Besides, this fact has been recorded in the respective per-CPU `rcu_data` structure and not in the node hierarchy. Thus, 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.

5.3.4 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()`, a function whose duties include:

- Awakening the RCU grace-period kthread (by invoking the `rcu_start_gp()` function), in order 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 the function `rcu_check_quiescent_state()`, which in turn invokes `note_gp_changes()`. The latter advances this CPU's callbacks 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 `rcu_report_qs_rdp()`, 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 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; see Sect. 9.

5.3.5 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 (`dynticks_nesting`) and increments a per-CPU counter (`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, which increments `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 so that it can be safely determined if a CPU is (or has been, at some point) in a quiescent state for this grace period. 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 (see Sect. 5.3.7).

5.3.6 Interrupts and dynticks-idle mode

When a CPU enters an interrupt handler, the function `rcu_irq_enter()` is invoked. This function increments the value of `dynticks_nesting` and, if the prior value was zero (i.e., the CPU was in dynticks-idle mode), also increments the `dynticks` counter. When a CPU exits an interrupt handler, `rcu_irq_exit()` decrements `dynticks_nesting`, and if the new value is zero (i.e., 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.

5.3.7 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 will try to force quiescent states on CPUs that have yet to report one. More specifically, the grace-period kthread will invoke `rcu_gp_fqs()`, which 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 whether 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.

6 Kernel environment modeling

In this section, we present the way we scaffolded a non-preemptible Linux kernel symmetric multiprocessing (SMP) environment. For this, we had to disable some timing-based warnings and stub out some primitives used in functions that were not included in our tests (e.g., RCU-expediting related functions). 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.

6.1 Modeling an SMP platform

Modeling CPUs 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 by a thread, then this thread is running on the respective CPU.

We assume that all CPUs are online, that there are no CPU hotplugs, and that the full dynticks system (tickless operation) is disabled (`CONFIG_NO_HZ_FULL=n`). All CPUs are initially idle, and when a thread wishes to acquire/release a CPU, it acquires/releases the CPU's lock and exits/enters idle mode (if necessary).

We also need to emulate per-CPU variables. In the kernel, these variables are created using special compiler/linker directives, along with some preprocessor directives. However, since these variables require significant runtime support, we used arrays to emulate them, with each array entry representing the respective CPU's copy of a per-CPU variable.

Lastly, since a thread needs to have knowledge regarding the CPU it runs on, we implemented two macros (`set_cpu()` and `get_cpu()`), which manipulate a thread-local variable indicating the CPU on which a thread runs. The CPU 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.

Emulating 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, the CPU's lock must be already held. 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.

We also need to model scheduling-clock interrupts (on which RCU relies heavily) and `rcu_check_callbacks()` function. But, as mentioned, stateless model checking is performed on deterministic programs, meaning that timing-based actions cannot be included in our tests. However, the exact time an interrupt occurs is not so important; what interests us is the implications a timing interrupt might have at a certain point of a program's execution given a concurrency context. Consequently, our version of the interrupt handler only invokes `rcu_check_callbacks()` and then, if an RCU softirq is raised, the `rcu_process_callbacks()` function will be called. Of course, we could have just called the function `rcu_process_callbacks()` directly, but in the Linux kernel this function is not invoked unconditionally, and we wanted our model to be as precise 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, to mark the passing through a quiescent state.

A better way to model this function would probably have been to drop the current CPU's lock, acquire the lock of a *random* CPU, and then check that no assertion is violated for *every* possible CPU choice. However, doing this requires support for data non-determinism, at least in the form of some suitable built-in (like, for example, the `VS_toss(n)` built-in that the VeriSoft tool provided). Alas, Nidhugg currently does not provide such support. This also explains why we have not modeled a preemptible kernel's environment.

This modeling, however, does not affect the correctness of our modeling for non-preemptible builds. The threads in our tests represent the CPUs of a system and not kthreads. Indeed, since a kthread cannot be preempted within its RCU read-side critical section (and resume on a different CPU), we do not care about the specific kthread that enters/exits a critical section or services a softirq; we merely care about the CPU on which these actions are performed. In that sense, our tests are CPU-centered and not kthread-centered.

Lastly, we note that we stubbed the `resched_cpu()` function; since this is used in scenarios, we did not include in our tests (e.g., RCU CPU stalls).

6.2 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 some `#include` directives and also provided some definitions based on specific `Kconfig` options. These include CPU-related definitions (e.g., `NR_CPUS`), RCU-related definitions (e.g., `CONFIG_RCU_BOOST`) that are normally configured at compile time, special compiler directives, tracing functions, etc. Some debugging facilities in the code, like the `BUG_ON()` macro (which panics the kernel) and its relatives (e.g., `WARN_ON()`, which conditionally logs a message in the kernel logs) were replaced by `assert()` statements. Note that we only stubbed primitives irrelevant to our tests (e.g., primitives used in grace-period expediting functions) and provided our own definitions for some other primitives in order for them to work with our modeling of the CPUs and interrupts. Memory barriers are provided for a TSO (x86), a PSO-like, and a POWER configuration. However, the POWER configuration was not used in the verification of the grace-period guarantee, due to Nidhugg's lack of support for atomic operations under POWER.

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

6.3 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_t` data type definition directly from the Linux kernel, this is not the case for atomic operations like `atomic_read()`, `atomic_set()`, since their implementation is architecture dependent. In order to emulate those, we used some `GCC` language extension [31] supported by `clang` [32], the compiler that produces the `LLVM IR` code that Nidhugg analyzes.

Spinlocks and mutexes Because we wanted an architecture-independent implementation that is supported by Nidhugg, we used `pthread_mutex`es for the emulation of kernel spinlocks and mutexes. 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 also had to model wait queues.

Since a thread waiting on a completion is put on a wait queue until some condition is satisfied, we used spin loops in order to emulate this waiting behavior. Nidhugg automatically transforms all spin loops to `__VERIFIER_assume()` statements where, if the condition does not hold, the execution blocks indefinitely [33]. 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, in order to report a quiescent state to RCU) could have been invoked before the thread released the CPU's lock. However, if the thread waiting on the completion variable is not the only thread running on the specific CPU, this is unnecessary; these functions can be called from other threads running on the same CPU as well.

7 Verifying the publish–subscribe guarantee

Now that we have explained how the kernel's environment can be modeled, let us first discuss the verification of the publish–subscribe guarantee. We tackled only a part of this guarantee, namely the `rcu_assign_pointer()`.

Since the primitives relevant for this guarantee are part of RCU's API and are used by different RCU flavors, their implementation does not change as often as the core of RCU. We thus verified the guarantee only for Linux kernel v3.19.

The publish–subscribe guarantee can be undermined by both the compiler and the CPU. Since Nidhugg cannot detect compiler-induced errors, we verified the guarantees provided by `rcu_assign_pointer()` only as far as the CPU is concerned. To do so, we used the model we constructed in Sect. 6.

The litmus test we used is based on the test of Fig. 2 and involves only a subscriber thread and a publisher thread. The test is configured to run under SC, TSO (x86), and POWER, depending on the preprocessing options used. The actual definition for `rcu_assign_pointer()` is copied from the kernel for the respective architecture.

To generate a buggy test, we pass to Nidhugg the `-DORDERING_BUG` preprocessor option, which when used replaces `rcu_assign_pointer()` with plain assignments. Under a memory model that reorders stores (e.g., POWER), this can lead to bugs. Thus, using a simple `BUG_ON()` statement we were able to check whether the subscriber can see uninitialized values.

The results we got are not surprising. In the case of POWER, the `rcu_assign_pointer()` primitive was absolutely necessary, whereas it was not required under SC or TSO (at least from a hardware perspective) because store-store reordering is not allowed in these memory models. This is why `rcu_assign_pointer()` boils down to a plain compiler barrier in the case of x86 in the kernel.[6] The respective tests finished in just 0.08 s (compilation and transformation time included) for all configurations.

As a final comment, note that we have verified (from a hardware perspective) that `rcu_assign_pointer()` has a correct implementation in the Linux kernel for TSO and POWER. To fully verify the publish–subscribe guarantee (again from a hardware perspective), we would also need to verify `rcu_dereference()`'s implementation, but that would require support from Nidhugg for an architecture that allows dependent-load reordering (e.g., DEC Alpha).

8 Verifying the grace-period guarantee

Next, we will verify the grace-period guarantee of Tree RCU for a non-preemptible Linux kernel environment, using the model we created in Sect. 6. We have applied this model to five different Linux kernels (v3.0, v3.19, v4.3, v4.7, and v4.9.6) and were able to verify that the actual RCU code satisfies the GP guarantee under SC, TSO, and PSO, using a litmus test similar to that of Fig. 1.

[6] With the exception of the Pentium Pro family of processors.

All experiments have been run on a 64-bit desktop with an Intel Core i7-3770 processor @ 3.40 GHz and 16 GB of RAM running Arch Linux 4.10.13-1-ARCH. We used Nidhugg's `--unroll` option with an appropriate value in order to put a bound on loops (e.g., server loops in RCU's grace-period kthread) that are unbounded.

8.1 Test configuration

Let us first briefly discuss our modeling of the Linux kernel. All our experiments focused on the RCU-sched flavor of Tree RCU (see Sect. 5).

First of all, we model a system with two CPUs, 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). 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. 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, we have sprinkled calls to `do_IRQ()` in various points of the test code, which 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 enable both of these scenarios.

8.2 Test runs

After running the tests with Nidhugg, the tool reports that the verification procedure is successful for all five kernel versions. Moreover, the process is quite fast. As shown on the first row of Table 1, the verification of the GP guarantee under SC requires less than 18 min (for kernels v4.3 and v4.9.6) and less than 10 min for each of the three other kernels. Another set of runs, shown in Table 2, verifying this guarantee under the TSO memory model does not require considerably more time. In contrast, verification of the GP guarantee under Nidhugg's PSO model takes considerably longer (up to 2.5 h for kernels v4.3 and v4.9.6; cf. Table 3); for reasons we will explain later. Still, for all five kernel versions, Nidhugg tells us that there is no possible thread or memory model interleaving that violates the GP guarantee in Tree RCU's implementation.

Table 1 Results for Tree RCU litmus test on five Linux kernel versions (time in seconds) under SC

Preprocessor options	v3.0		v3.19		v4.3		v4.7		v4.9.6	
	Time	Traces	Time	Traces	Time	Traces	Time	Traces	Time	Traces
–	**227.37**	**19,398**	**594.33**	**24,760**	**1041.85**	**28,996**	**411.25**	**11,076**	**1033.12**	**28,996**
-DASSERT_0	2.09	145	1.47	37	1.70	29	1.93	29	2.06	29
-DFORCE_FAILURE_1	2.11	146	1.51	41	1.99	33	2.09	33	2.13	33
-DFORCE_FAILURE_2	0.43	4	0.63	3	0.80	3	1.18	3	1.14	3
-DFORCE_FAILURE_3	27.60	2372	399.89	13,264	334.13	8114	290.50	8114	307.72	8114
-DFORCE_FAILURE_4	1.35	84	3.01	79	1.87	24	3.09	43	2.99	43
-DFORCE_FAILURE_5	58.15	4888	1.18	9	1.22	9	1.57	9	1.60	9
-DFORCE_FAILURE_6	1.14	1	3.34	2	5.28	2	10.40	2	10.80	2
-DLIVENESS_CHECK_1-DASSERT_0	24.91	2024	10.73	608	11.27	488	10.15	488	10.77	488
-DLIVENESS_CHECK_2-DASSERT_0	50.28	3888	10.46	608	12.30	516	11.55	516	11.80	516
-DLIVENESS_CHECK_3-DASSERT_0	26.38	2184	12.48	688	11.37	488	11.66	532	11.89	532

Table 2 Results for Tree RCU litmus test on five Linux kernel versions (time in seconds) under TSO

Preprocessor options	v3.0		v3.19		v4.3		v4.7		v4.9.6	
	Time	Traces	Time	Traces	Time	Traces	Time	Traces	Time	Traces
–	**260.38**	**19,398**	**651.32**	**24,760**	**1147.20**	**28,996**	**416.03**	**11,076**	**1125.30**	**28,996**
`-DASSERT_0`	2.30	145	1.66	37	1.77	29	2.09	29	2.16	29
`-DFORCE_FAILURE_1`	2.27	146	1.67	41	1.97	33	2.26	33	2.32	33
`-DFORCE_FAILURE_2`	0.38	4	0.66	3	1.03	3	1.22	3	1.27	3
`-DFORCE_FAILURE_3`	31.02	2372	433.38	13,264	316.14	8114	329.10	8114	338.26	8114
`-DFORCE_FAILURE_4`	1.46	84	3.13	79	2.08	24	3.17	43	3.26	43
`-DFORCE_FAILURE_5`	64.80	4888	0.98	9	1.39	9	1.66	9	1.65	9
`-DFORCE_FAILURE_6`	1.41	1	3.33	2	5.13	2	10.34	2	10.93	2
`-DLIVENESS_CHECK_1-DASSERT_0`	26.42	2024	11.32	608	10.88	488	11.64	488	11.85	488
`-DLIVENESS_CHECK_2-DASSERT_0`	52.37	3888	11.13	608	11.95	516	12.49	516	13.18	516
`-DLIVENESS_CHECK_3-DASSERT_0`	28.98	2184	13.63	688	11.23	488	12.75	532	13.27	532

Table 3 Results for Tree RCU litmus test on five Linux kernel versions (time in seconds) under PSO

Preprocessor options	v3.0		v3.19		v4.3		v4.7		v4.9.6	
	Time	Traces	Time	Traces	Time	Traces	Time	Traces	Time	Traces
–	**2510.13**	**89,038**	**2116.82**	**25,824**	**8346.02**	**47,556**	**2718.69**	**15,876**	**9081.39**	**47,556**
`-DASSERT_0`	10.79	436	2.32	40	2.78	29	3.06	29	3.25	29
`-DFORCE_FAILURE_1`	10.75	437	2.64	44	3.36	33	3.69	33	3.87	33
`-DFORCE_FAILURE_2`	0.40	4	0.59	3	0.79	3	0.95	3	1.00	3
`-DFORCE_FAILURE_3`	732.84	28,700	1433.08	15,658	1470.49	8114	1523.11	8114	1602.66	8114
`-DFORCE_FAILURE_4`	2.61	84	8.61	79	5.21	24	11.21	43	11.80	43
`-DFORCE_FAILURE_5`	1502.28	59,596	1.15	9	1.74	9	1.94	9	2.02	9
`-DFORCE_FAILURE_6`	1.13	1	3.42	2	5.17	2	9.97	2	10.58	2
`-DLIVENESS_CHECK_1-DASSERT_0`	567.05	23,132	28.45	712	31.56	488	32.54	488	35.86	488
`-DLIVENESS_CHECK_2-DASSERT_0`	1044.58	41,100	28.26	712	35.19	516	36.38	516	38.82	516
`-DLIVENESS_CHECK_3-DASSERT_0`	584.59	23,976	33.48	792	31.56	488	37.00	532	39.21	532

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 GP 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.

Figure 5 contains information about the bug injections keyed by the define macro that enables each test. Note that, for the `FORCE_FAILURE_6` test, since a multi-level hierarchy for a system with more than 16 CPUs is created, an `--unroll=19` option has been used and `CONFIG_NR_CPUS` has been set appropriately, while for all other tests an `--unroll=5` option has been used. All tests had the desired outcome, something that increases our confidence in our modeling and the verification result for the GP guarantee of Tree RCU's implementation that we report.

8.3 Results and discussion

In Tables 1, 2, and 3, the "Time" columns represent the total wall-clock time in seconds (compilation and transformation time are included). As can be seen in Tables 1 and 2, the number of traces explored under SC and TSO are exactly the same and there is very little overhead when going from SC to TSO, which shows the power of stateless model checking with source-DPOR [21] and chronological traces [7]. The reason for the same total number of explored executions is twofold. RCU's source code does not have many opportunities for store-load reorderings, in the first place. Second, it also contains a lot of memory fences which prevent these reorderings from happening. But we also note that even if such reorderings were possible, all bug injections here do not rely on the employed memory model; instead, they violate the assertions algorithmically. As expected, since the model checking is stateless, the memory requirements are very low, especially considering the size of the source code under test: $\sim$35 MB for SC and TSO and $\sim$105 MB for PSO. In the case of `FORCE_FAILURE_6`, approximately three times more memory is required for all memory models, due to the higher unroll value.

The most interesting row in all tables is the first one, shown in bold. In all these cases, Nidhugg needs to explore the complete set of traces in order to verify that the GP guarantee indeed holds for Tree RCU's implementation. In rows with failure injections, exploration stops as soon as the failure is detected. How fast this happens depends on the order in which traces are explored. In some cases failures are detected immediately (in the first few traces and in less than 3 s) and in other cases only after many traces have been explored.

It can be observed that the number of explored traces varies between different kernel versions. With the exception of PSO, there are fewer traces explored in kernel v3.0 than in v3.19 and v4.3, due to the absence of the grace-period kthread in the first; this thread contains infinite loops which generate many races that Nidhugg tries to reverse. Note that in v3.0 the `-DFORCE_FAILURE_3` and `-DFORCE_FAILURE_5` injections are liveness checks due to the absence of the grace-period kthread. In kernel v4.7 the explored traces decrease dramatically due to the replacement of `rcu_gp_kthread_wake()` with `swake_up()` in `rcu_report_qs_rsp()`. The former performed a check which read a variable that was written by the grace-period kthread (among others) and generated far too many races. In kernel v4.9.6, however, this change was reverted and the explored traces are the same as those of kernel v4.3. Overall, it is obvious that irrespective of the kernel's growth in size, Nidhugg provides an efficient and scalable way to test such a big code base since the number of traces that need to be explored only depends on races on shared variables and not on the general complexity or size of the source code.

We also conducted some tests for a PSO-like architecture, shown in Table 3. We say "PSO-like" for two reasons: (a) because the Linux kernel does not support any PSO architectures[7] (so we had to provide architecture-specific definitions ourselves), and (b) because while Nidhugg does support a PSO memory model, this model is slightly stronger than the one used by real PSO CPUs. More specifically, Nidhugg does not provide a store-store fence (which would have similar semantics to the Linux kernel's `smp_wmb()`), thus forcing us to model the latter as a full memory barrier, and also forbids the reordering of atomic instructions with stores (in contrast to SPARC-PSO CPUs), rendering our model stronger than required. Nevertheless, all the tests had the expected outcome, and the results are consistent with the ones for SC and TSO. However, Nidhugg requires significantly more time (up to nine times) when operating under PSO, despite the fact that the traces are increased by a smaller factor (1.5–4). The reason for this is that, in order to emulate the PSO memory model, Nidhugg maintains one store buffer per global mem-

[7] SPARC CPUs are used in TSO mode.

-DASSERT_0: An `assert(0)` statement is inserted after `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 can end 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 `synchronize_rcu()`. 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 node mask (`->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 it. 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) have to be in dynticks-idle mode (or the CPU must have passed through a quiescent state at some point, since 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 bit of the respective node's mask 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 `->qsmask` bit, which implies that this CPU reported a quiescent state for the new grace period. This results in test failure.

-DFORCE_FAILURE_6: Essentially, what this injection does is delete the `if` statement checking whether a node's mask is zero 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 iterate at least as many times as the number of CPUs used plus one. So, we used an `--unroll=19` option for this case.

-DLIVENESS_CHECK_1: This eliminates the need for a CPU to pass through a quiescent state by setting `rdp->qs_pending` to zero in `__note_gp_changes()`. This function updates the per-CPU `rcu_data` structure and, 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()`.

-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 their passing through a quiescent state in the respective `rcu_data` structures, something that 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 grace periods cannot complete. This injection also needs to be used together with `-DASSERT_0` to discover the liveness violation.

Fig. 5 Description of the bug injections we used, identified by the preprocessor option that enables them

ory location used in the program, which imposes a constant, yet non-negligible overhead.

9 Presenting the cause of an older kernel bug

In Sect. 5.3.4, 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 [34,35], which caused the advancement of callbacks whose grace period had not yet expired. Below we will create a test case that exposes such a scenario, but this test case will also demonstrate *that the problem is actually deeper*: these unsynchronized accesses also lead to too-short grace periods.

```
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;
}
```

Fig. 6 Snippet of the `rcu_process_gp_end()` function

To construct a test case that exposes this issue, we started by looking at the `rcu_process_gp_end()` function, since the issue was related to it. Figure 6 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 whether 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 whether this could potentially lead to a CPU starting a new grace period without having noticed that the last grace period has 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. With these clues, we constructed a simple test which proved that these unsynchronized accesses can lead to too-short grace periods. The test 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 which, of course, is a violation of the grace-period guarantee. A sequence of events (produced by Nidhugg) which exposes this bug is shown in Fig. 7. Observe that these events form a quite involved thread interleaving.

Interestingly enough, just 2 days before the patch that fixed this bug, commit `83f5b01ffbba` fixed an alleged long-grace-period race between grace-period forcing and initialization [36], which was supposedly responsible for the failures observed at the time in multi-node hierarchies. However, we constructed a test case which showed that an interleaving such as the one described in the commit log is impossible, and other variations of our test case also did not expose any bug as well. Ultimately, it was confirmed to us (Paul E. McKenney, personal communication; also [37]) that the analysis presented in the commit log was wrong, and that these two issues were related (also see [38]): commit `d09b62dfa336` serendipitously fixed the bug commit `83f5b01ffbba` was supposed to fix.

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

– In contrast to what the commit log states [36], this bug does not rely on interactions with the node hierarchy; it existed in both single-node and multi-level hierarchies. (A slightly different test case with the respective `Kconfig` options set appropriately was created for multi-level hierarchies.)
– Nidhugg reports that this bug is not present in kernel v3.0, which means that it was indeed fixed. In v3.0, `rcu_start_gp()` calls `__rcu_process_gp_end()`, thus

(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, 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.

Fig. 7 Sequence of events resulting in an RCU tree bug

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., v2.6.31.1.
- 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). 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 as well.
- Nidhugg produced the violating sequence of events in only 0.56 s (compilation and transformation time included) and used 30.85 MB of memory in total.

10 Further discussion

10.1 Threats to validity

As mentioned in Sect. 3, stateless model checking requires that test cases are data-deterministic and finite: the former requirement implies that our approach can only detect concurrency bugs, while the latter implies that our results are valid only up to the bound up to which all loops have been unrolled. In addition, Nidhugg operates at the level of LLVM-IR; it is conceivable that the Clang compiler hides some bugs in the translation from C to LLVM-IR, or that the back-end compiler exposes some other bugs when compiling the LLVM-IR to native code. Naturally, Nidhugg cannot detect all these. Finally, we note that Nidhugg itself is not verified, and a bug in its implementation could cause it to miss some bugs in RCU's code as well.

A second set of threats to validity stems from limitations related to the testing procedure and infrastructure. Our efforts concentrated on specific kernel builds, which led to specific RCU configurations, and our verification results hold only for these particular configurations. As a side note, we mention that some of these restrictions were partly imposed by Nidhugg and its limitations. For example, the fact that we only handle preemptible kernels is because Nidhugg does not support data non-determinism. (Refer to the "Scheduling" paragraph of Sect. 6 for more explanation of this limitation.)

As far as the grace-period guarantee is concerned, we focused on 2-CPU systems, single-node hierarchies (which occur when `NR_CPUS` < 16), non-preemptible kernels, and single-grace-period test cases. Although our model allows for other configurations (i.e., it is tunable), and we did try other configurations as parts of our tests (see Sects. 9 and 10.2), our primary testing infrastructure was the one described in Sect. 8. Having said that, we note that the code paths activated in a single-node hierarchy are the same no matter the exact number of CPUs (i.e., there is no conditional execution depending on the value of `NR_CPUS`).

As far as the publish–subscribe guarantee is concerned, we only verified a part of it. A detailed description is provided in Sect. 7.

Finally, the last set of threats to validity is due to limitations related to the modeling of the kernel's environment. Apart from errors that might exist in our modeling, the way some primitives were modeled could have affected the outcome of the verification procedure. One such case is the modeling of scheduling-clock interrupts and the `do_IRQ()` function. Although we tried different modelings for this function and we chose the one described in Sect. 6 for efficiency reasons (see Sect. 10.2), it could be argued that this renders our model less precise. That is true; however, given enough computational resources, different modeling of other components, or intervening in the kernel's code, the respective needs of each verification scenario can be met.

10.2 Lessons learned

The verification process itself was very educational. We gained useful experiences regarding the construction of the model of the Linux kernel, the model itself, and on how to deal with the combinatorial explosion in the number of interleavings that a stateless model checking tool needs to explore.

Arguably, the most valuable lesson learned was the way a Linux kernel model can be constructed. Initially, the way an SMP system should be emulated was not obvious, and the construction of the model had to be precise. Both of these posed two non-trivial challenges; with the kernel occupying more than 15MLOC, the isolation and testing of only the ingredients we cared about was of extreme importance. Still, we managed to use the source code from the various kernel versions directly, and the constructed model is reusable, meaning that it can be used for further RCU testing, and perhaps for testing other components of the Linux kernel as well.

Of course, confining the state space was not in any way an easy task as well. First of all, as far as the model is concerned, the most important design decision we had to make was the way the interrupts are modeled. Initially, we tried to emulate interrupts with per-CPU threads invoking the interrupt handler repeatedly, but unfortunately this approach rendered the state space extremely large. Apart from this, plenty of other design choices were made and most of them are described in Sect. 6. As far as the verification of Tree RCU is concerned, multiple different configurations were tried and did not affect the outcome. We chose the one mentioned in Sect. 8.1 because the state space was considerably smaller. The reason for that, although not obvious from the beginning, is that the updater and the grace-period kthread are mutually exclusive and take advantage of each other's context switches. In addition, 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, we wanted our model to be as precise as possible, so we did not resort to such approximations.

11 Related work

Previous work on RCU verification includes the expression of RCU's formal semantics in terms of separation logic [39] and the verification of user-space RCU in a logic for weak memory [40]. A virtual architecture to model out-of-order memory accesses and instruction scheduling has been proposed [41], and a verification of user-space RCU has been done using the SPIN model checker [42]. Alglave et al. verified that RCU's *actual* kernel code preserves data consistency of the object it is protecting [43] using CBMC [44], i.e., they verified the combination of `rcu_assign_pointer()` and `rcu_dereference()`. Subsequently, McKenney [45] verified the grace-period guarantee for Tiny RCU. Finally, mutation testing strategies have been applied to RCU's code [46] as well.

Concurrently with our work, Liang et al. used CBMC to verify the grace-period guarantee for Tree RCU [47]. 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, and this has some implications in the verification procedure. The most basic one is that bugs in the callback handling mechanism (e.g., a bug similar to the one we reproduced in Sect. 9) cannot be exposed. Considering the fact that RCU's update side primitives are based on callback handling, this limitation is a serious one. For example, primitives like `call_rcu()` were not included in their tests, and `synchronize_rcu()`'s implementation (which, in reality, is based on `call_rcu()`) had to be emulated. This in turn means that only the underlying grace-period mechanism was modeled, and 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 their 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, Liang et al. [47] report that CBMC needs more than 11 h and 34 GB of memory in order to claim successful verification for Tree RCU in kernel v4.3 under TSO. In contrast, as we report in this article, Nidhugg only needs 19 min and 102 MB of memory for the same task. More generally, our results are orders of magnitude better, which we attribute to the different algorithms that the two tools employ.

On the other hand, 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 the verification of the grace-period guarantee of Tree RCU for the configurations we tested. Some supporting evidence for this claim offers the fact that the bug injections we listed in Sect. 8 are a proper superset of those identified by CBMC.[8] Furthermore, because our approach does include callback handling, we were able to reproduce an older, real kernel bug that was caused by premature callback advancements, which could potentially lead to too-short grace periods that violate the GP guarantee. As explained, this bug cannot be reproduced with CBMC, due to its limited support for lists.

12 Concluding remarks

In this article, 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, using the stateless model checking tool Nidhugg, we managed to verify both a part of RCU's API (publish–subscribe guarantee) which is used by different RCU flavors, and the grace-period guarantee, the most basic guarantee that RCU offers for the main implementation used in the Linux kernel, namely, Tree RCU.

More specifically, we verified the grace-period guarantee for five different Linux kernel versions, under both a sequentially consistent and weak (TSO and PSO) memory models. For all our tests, we used the source code from the Linux kernel directly, with only a handful of changes, which can be and have been scripted.

[8] Injections `-DFORCE_FAILURE_1` and `-DFORCE_FAILURE_4` are not considered by Liang et al. [47]; the latter due to not modeling the dynticks-idle mode.

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 an older kernel [35] resolved a much more complex issue that was in effect a concurrency 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 have matured to the point that they can be used to test *real* code from today's production systems with large code bases. The small time and memory consumption of our tests, especially considering the size and the dynamic nature of the code base 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, although the GP guarantee is the most significant correctness property of RCU, there 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 architectural memory models (e.g., POWER) or, naturally, the recently proposed Linux kernel memory model [48]. In addition, we could construct tests that include quiescent-state forcing, grace-period expediting, and CPU hotplugs. The same applies for the full dynticks mode which was fully merged in the kernel only relatively recently. Last but not least, the scalability of our results renders the construction of test cases and techniques aiming at the thorough testing of the preemptible Tree RCU extremely interesting as well.

Acknowledgements Open access funding provided by Max Planck Society. We are 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 article would not have been possible without Nidhugg; we thank its main developer, Carl Leonardsson, for the hard work he has put into it. Finally, we thank the anonymous reviewers of our SPIN 2017 paper and of this article for their helpful comments and feedback.

Compliance with ethical standards

Data availability The Nidhugg tool is available at https://www.github.com/nidhugg/nidhugg. The code we used for our experiments and scripts to reproduce the results we report are available at https://www.github.com/michaliskok/rcu.

References

1. Callaham, J.: Google says there are now 1.4 billion active Android devices worldwide (2015). http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide
2. Prakash, A.: Linux now runs on all of the top 500 supercomputers (2017). https://itsfoss.com/linux-runs-top-supercomputers/
3. Weinberger, M.: For the first time ever, Microsoft will distribute its own version of Linux (2018). http://www.businessinsider.de/microsoft-azure-sphere-is-powered-by-linux-2018-4?r=US&IR=T
4. McKenney, P.E., Slingwine, J.D.: Read–copy update: using execution history to solve concurrency problems. In: Parallel and Distributed Computing and Systems, pp. 509–518. Las Vegas, NV (1998). http://www.rdrop.com/~paulmck/scalability/paper/rclockpdcsproof.pdf
5. McKenney, P.E., Walpole, J.: What is RCU, fundamentally? (2007). http://lwn.net/Articles/262464/
6. RCU torture test operation (2017). https://www.kernel.org/doc/Documentation/RCU/torture.txt
7. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. Acta Inf. **54**(8), 789–818 (2017). https://doi.org/10.1007/s00236-016-0275-0
8. Kokologiannakis, M., Sagonas, K.: Stateless model checking of the Linux kernel's hierarchical read–copy–update (Tree RCU). In: Proceedings of International SPIN Symposium on Model Checking of Software, SPIN 2017. ACM, New York (2017). https://doi.org/10.1145/3092282.3092287
9. Sparse—a semantic parser for C. https://sparse.wiki.kernel.org
10. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 147–186. ACM, New York (1997). https://doi.org/10.1145/263699.263717
11. Godefroid, P.: Software model checking: the VeriSoft approach. Form. Methods Syst. Des. **26**(2), 77–101 (2005). https://doi.org/10.1007/s10703-005-1489-x
12. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, pp. 267–280. USENIX Association, Berkeley (2008). http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf
13. Christakis, M., Gotovos, A., Sagonas, K.: Systematic testing for detecting concurrency errors in Erlang programs. In: 6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013), pp. 154–163. IEEE Computer Society, Los Angeles (2013). https://doi.org/10.1109/ICST.2013.50
14. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI 2015, pp. 250–259. ACM, New York (2015). https://doi.org/10.1145/2737924.2737956

15. Norris, B., Demsky, B.: A practical approach for model checking C/C++11 code. ACM Trans. Program. Lang. Syst. **38**(3), 10:1–10:51 (2016). https://doi.org/10.1145/2806886
16. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. PACMPL **2**(POPL), 17:1–17:32 (2018). https://doi.org/10.1145/3158105
17. Valmari, A.: 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, pp. 491–515. Springer, London (1991). http://dl.acm.org/citation.cfm?id=647736.735461
18. Peled, D.: All from one, one for all: On model checking using representatives. In: Proceedings of the 5th International Conference on Computer Aided Verification, LNCS, pp. 409–423. Springer, London (1993). http://dl.acm.org/citation.cfm?id=647762.735490
19. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Ph.D. Thesis, University of Liège (1996). https://doi.org/10.1007/3-540-60761-7. Also, volume 1032 of LNCS, Springer
20. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 110–121. ACM, New York (2005). https://doi.org/10.1145/1040305.1040315
21. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 373–384. ACM, New York (2014). https://doi.org/10.1145/2535838.2535845
22. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: a foundation for optimal dynamic partial order reduction. J. ACM **64**(4), 25:1–25:49 (2017). https://doi.org/10.1145/3073408
23. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: Tools and Algorithms for the Construction and Analysis of Systems—24th International Conference, LNCS, vol. 10806, pp. 229–248. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_14
24. McKenney, P.E.: RCU: The Bloatwatch edition (2009). http://lwn.net/Articles/323929/
25. McKenney, P.E.: rcu: Remove TINY_PREEMPT_RCU (2013). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=127781d1ba1ee5bbe1780afa35dd0e71583b143d
26. NO_HZ: Reducing Scheduling-clock Ticks (2017). https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt
27. What is RCU?—Linux kernel documentation. https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt
28. McKenney, P.E.: Hierarchical RCU (2008). http://lwn.net/Articles/305782/
29. CPU Hotplug in the Kernel (2016). https://www.kernel.org/doc/Documentation/core-api/cpu_hotplug.rst
30. RCU Linux kernel documentation. https://www.kernel.org/doc/Documentation/RCU/
31. Built-in Functions for Memory Model Aware Atomic Operations. https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html
32. LLVM Atomic Instructions and Concurrency Guide (2017). http://llvm.org/docs/Atomics.html#libcalls-atomic
33. Beyer, D.: Rules for 4th international competition on software verification (2015). https://sv-comp.sosy-lab.org/2015/rules.php
34. rcu: Clean up locking for `->completed` and `->gpnum` fields (2009). https://lkml.org/lkml/2009/10/30/212
35. Rcu: fix synchronization for `rcu_process_gp_end()` uses of `->completed` counter (2009). https://lkml.org/lkml/2009/11/4/69
36. Rcu: fix long-grace-period race between forcing and initialization (2009). https://lkml.org/lkml/2009/10/28/196
37. McKenney, P.E.: Verification challenge 6: Linux-kernel tree RCU (2017). https://paulmck.livejournal.com/46993.html
38. McKenney, P.E.: Hunting heisenbugs (2009). http://paulmck.livejournal.com/14639.html
39. Gotsman, A., Rinetzky, N., Yang, H.: Verifying concurrent memory reclamation algorithms with Grace. In: Programming Languages and Systems, LNCS, vol. 7792, pp. 249–269. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-37036-6_15
40. Tassarotti, J., Dreyer, D., Vafeiadis, V.: Verifying read–copy–update in a logic for weak memory. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 110–120. ACM, New York (2015). https://doi.org/10.1145/2737924.2737992
41. Desnoyers, M., McKenney, P.E., Dagenais, M.R.: Multi-core systems modeling for formal verification of parallel algorithms. SIGOPS Oper. Syst. Rev. **47**(2), 51–65 (2013). https://doi.org/10.1145/2506164.2506174
42. Desnoyers, M., McKenney, P.E., Stern, A.S., Dagenais, M.R., Walpole, J.: User-level implementations of read–copy update. IEEE Trans. Parallel Distrib. Syst. **23**(2), 375–382 (2012). https://doi.org/10.1109/TPDS.2011.159
43. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Computer Aided Verification, LNCS, vol. 8044, pp. 141–157. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-39799-8_9
44. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 2988, pp. 168–176. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-24730-2_15
45. McKenney, P.E.: Verification challenge 4: Tiny RCU (2015). http://paulmck.livejournal.com/39343.html
46. Ahmed, I., Groce, A., Jensen, C., McKenney, P.E.: How verified is my code? falsification-driven verification. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, pp. 737–748. IEEE Computer Society, Washington, DC (2015). https://doi.org/10.1109/ASE.2015.40
47. Liang, L., McKenney, P.E., Kroening, D., Melham, T.: Verification of the tree-based hierarchical read–copy update in the Linux kernel (2016). http://arxiv.org/abs/1610.03052
48. Alglave, J., Maranget, L., McKenney, P.E., Parri, A., Stern, A.S.: Frightening small children and disconcerting grown-ups: concurrency in the Linux kernel. In: Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, pp. 405–418. ACM, New York (2018). https://doi.org/10.1145/3173162.3177156