

Верификация иерархического механизма Read-Copy Update ядра Linux

Lihao Liang

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

Paul E. McKenney

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

Daniel Kroening

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

Tom Melham

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

Abstract

Read-Copy Update (RCU) — это высокопроизводительный масштабируемый механизм синхронизации ядра операционной системы Linux, который позволяет выполнять нетребовательные к ресурсам запросы на чтение данных вместе с запросами на их изменение. Реализация качественного RCU для многоядерных систем является весьма сложной задачей. Учитывая распространённость Linux, даже самая редкая ошибка исходного кода реализации будет проявляться недопустимо часто. В связи с этим, строгая валидация сложных сценариев поведения RCU является критически важной. В связи с тем, что исчерпывающее тестирование данного механизма невозможно из-за экспоненциального роста числа сценариев тестирования, имеет смысл использовать метод формальной верификации.

Следует отметить, что прошлые попытки верификации RCU были направлены либо на более простые реализации, либо использовали языки моделирования, что требует процесса ручного перевода исходного текста ядра, который также подвержен ошибкам. Кроме этого, подобный перевод придется выполнять слишком часто, поскольку в реализацию RCU Linux регулярно вносятся правки. В этой статье мы опишем реализацию Tree RCU в ядре Linux, затем рассмотрим подход к построению модели верификации напрямую из исходного кода реализации и использованию верификатора CBMC для проверки ее инвариантов. По нашим сведениям, это

первая попытка верификации существенной части исходного кода RCU и важный шаг на пути интеграции процедуры формальной верификации в набор регрессионных тестов ядра Linux.

Categories and Subject Descriptors [D.2.4]: Программное обеспечение/Верификация программ—Верификация моделей; [D.1.3]: Многопоточное программирование—Параллельное программирование

Keywords Верификация программного обеспечения, Параллельные вычисления, Read-Copy Update, ядро Linux

1. Введение

Ядро операционной системы Linux широко используется во множестве вычислительных платформ, включая сервера, встроенные системы, бытовую технику и мобильные устройства (смартфоны). В течение последних 25 лет в ядре Linux было реализовано множество технологий, одной из которых является Read-Copy Update (RCU) [?].

RCU — это механизм синхронизации, который может использоваться взамен блокировок чтения-записи в тех случаях, когда число запросов на чтение значительно больше. Он позволяет выполнять нетребовательные к ресурсам запросы на чтение данных вместе с запросами на их изменение. Качественные реализации RCU для многоядерных систем должны обладать такими качествами, как отличная масштабируемость, высокая пропускная способность, низкие задержки, умеренный расход памяти, низкое энергопотребление и поддерживать «hotplug» операции процессора. В связи с этим реализация должна избегать кэш-промахов, избыточного использования блокировок, общих переменных, а также атомарных «read-modify-write» и «memory-barrier» инструкций. Наконец, реализация должна поддерживать

все многообразие целевых платформ и сценариев использования Linux [?].

В настоящее время RCU широко используется в сетевой, файловой подсистемах ядра Linux, а также в подсистеме работы с устройствами [? ?]. На данный момент в мире насчитывается более 75 миллионов Linux-серверов и 1.4 миллиардов мобильных Android-устройств. На практике это означает, что даже самая редкая ошибка исходного кода реализации будет проявляться недопустимо часто. В связи с этим, строгая валидация сложных сценариев поведения RCU является критически важной. В связи с тем, что исчерпывающее тестирование данного механизма невозможно из-за экспоненциального роста числа сценариев тестирования, имеет смысл использовать метод формальной верификации.

Основные усилия, направленные с верификацией многопоточного программного обеспечения, основаны на тестировании, но, к сожалению, на данный момент не существует эффективной методики тестирования такого ПО, способной проверить все возможные сценарии. Более того, некоторые ошибки, которые обнаруживаются в ходе тестирования, могут быть трудными для повторного воспроизведения, отладки и исправления. Многопоточная сущность RCU и огромное пространство возможных сценариев тестирования наводят на мысль об использовании методов формальной верификации, в частности, верификации моделей [?].

Следует отметить, что формальные методы уже применялись ранее для верификации некоторых частей дизайна RCU, например Tiny RCU [?], userspace RCU [?], sysidle [?] и взаимодействия между dyntick-idle и немаскируемыми прерываниями (NMI) [?]. Но эти попытки были направлены либо на верификацию примитивных реализаций RCU для одноядерных систем (Tiny RCU), либо использовали специализированные языки описания моделей, такие, как Promela [?]. Несмотря на то, что данные языки имеют ряд преимуществ, основным недостатком их использования в контексте ядра Linux является сложность ручной трансляции исходного кода. Иные исследователи использовали аппарат формальной логики для верификации простых реализаций RCU [? ?]. Несмотря на то, что данный подход является весьма интересным, он требует большего количества работы, нежели трансляция исходного кода.

Более того, цикл разработки ядра Linux составляет около 60 дней, и в течение каждого из них вносятся правки в RCU. В связи с этим всякий ручной труд по верификации должен будет повторяться шесть раз в год, чтобы формальные модели верификации RCU оставались актуальными. Из этого следует, что для того, чтобы процесс формальной верификации RCU мог быть включен в набор регрессионных тестов, используемые в нем

методы должны быть автоматизируемыми и масштабируемыми. В этой статье описывается процесс построения модели верификации напрямую из исходного кода реализации RCU в Linux и использование C Bounded Model Checker (CBMC) [?] для проверки ее инвариантных свойств. По нашим сведениям, это первая попытка автоматизированной верификации существенной части исходного кода RCU и важный шаг на пути интеграции процедуры формальной верификации в набор регрессионных тестов ядра Linux.

2. Background

2.1 Что такое RCU?

Read-copy update (RCU) — это механизм синхронизации, часто используемый вместо блокировок чтения-записи. RCU позволяет потокам-читателям выполняться одновременно с потоками-писателями, избегая использования блокировок чтения за счет управления жизненными циклами множества версий объекта чтения. В частности, данный механизм следит, чтобы объект, к которому обращается поток-читатель, не был удален в течение некоторого *grace*-периода после его изменения потоком-писателем. Суть метода состоит в том, чтобы разделить процесс обновления объекта на фазу удаления и освобождения, между которыми находится некоторый промежуток времени — *grace*-период [?]. В ходе фазы удаления выполняется удаление ссылок на объекты, доступных для потоков-читателей, сопровождающееся, возможно, заменой их новыми версиями.

Современные процессоры гарантируют, что операции чтения одичночных выравненных указателей являются атомарными, поэтому потоки-читатели могут получить доступ исключительно либо к старой или новой версии объекта чтения. Atomic-write semantics позволяет выполнять атомарные вставки, удаления и замены в связанных структурах данных. Это, в свою очередь, позволяет потокам-читателям отказаться от использования «дорогих» атомарных операций, избавиться от барьеров памяти и связанных с ними промахов кэша. Действительно, в самых оптимизированных конфигурациях Linux RCU, потоки-читатели могут выполнять точно такую же последовательность инструкций, какая использовалась бы в их однопоточной реализации, что обеспечивает их отличную производительность и масштабируемость.

Как показано на рисунке 1, *grace*-периоды в действительности нужны только для тех потоков-читателей, время выполнения которых накладывается на фазу удаления. Те из них, которые выполняются после удаления, не могут удерживать ссылки на удаленные объекты и поэтому не могут быть прерваны в ходе фазы освобождения.

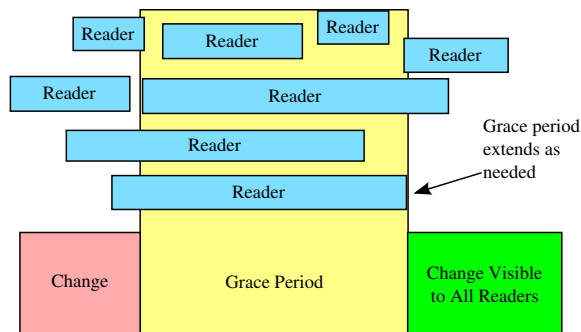


Рис. 1: RCU Concepts

2.2 Программный интерфейс RCU

Программный интерфейс RCU достаточно невелик и состоит всего из пяти операций: `rcu_read_lock()`, `rcu_read_unlock()`, `synchronize_rcu()`, `rcu_assign_pointer()`, and `rcu_dereference()` [?].

Критическая секция RCU-читателя начинается с `rcu_read_lock()` и заканчивается соответствующим `rcu_read_unlock()`. Вложенные критические секции чтения объединяются. Внутри критической секции запрещается блокирование данного потока. Данные, чтение которых осуществляется доступ внутри критической секции RCU, будут доступны до её окончания.

Функция `synchronize_rcu()` соответствует окончанию выполнения кода, обновляющего значение объекта, тем самым сигнализируя о начале фазы освобождения. Она блокирует поток-писатель до тех пор, пока все потоки-читатели не выйдут из своих критических RCU-секций. Отметим, что `synchronize_rcu()` не ожидает окончания критических секций, вход в которые был осуществлен позже её вызова.

Рассмотрим пример, приведенный на рисунке 2. Если вход в критическую секцию чтения функции `rcu_reader()` выполнится до вызова `synchronize_rcu()` в `rcu_updater()`, то выход из неё должен быть совершен до возврата из `synchronize_rcu()`, чтобы значение переменной `r2` было равно 0. Если же вход в неё произойдет после возврата из `synchronize_rcu()`, то значение `r1` будет равным 1.

Наконец, для присвоения нового значения указателю, защищенному RCU, потоки-писатели должны использовать `rcu_assign_pointer()`, которая возвращает новое значение. RCU-читатели могут использовать `rcu_dereference()` для чтения указателя, защищенного RCU, который впоследствии может быть безопасно разыменован. Возвращаемое ею значение является корректным лишь внутри критической секции. Функции `rcu_assign_pointer()` и `rcu_dereference()` используются в паре для того, чтобы убедиться, что если данный поток-читатель разыменовывает защищен-

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

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

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

...

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

Рис. 2: Verifying RCU Grace Periods

ный указатель на только что вставленный объект, операция разыменования вернет корректное значение, а не недоинициализированный мусор.

3. Реализация Tree RCU

Основное преимущество RCU заключается в том, что он позволяет ожидать выхода весьма большого числа потоков-читателей из своих критических секций без необходимости учета каждого из них: в ядрах с non-preemptible реализацией многопоточности их число ограничено количеством ядер процессора, в ядрах с preemptible реализацией — неограниченно вовсе. Несмотря на то, что примитивы чтения RCU обладают замечательными показателями производительности и масштабируемости, примитивы записи должны оттягивать фазу освобождения до тех пор, пока все потоки-читатели не выйдут из своих критических секций, за счет блокирования или регистрации callback'а, который должен быть вызван по истечении grace-периода. Производительность и масштабируемость RCU определяются эффективностью механизмов обнаружения окончания *grace*-периода. Например, простейшая реализация RCU может требовать, чтобы каждое ядро процессора использовало глобальную блокировку для каждого grace-периода, но этот подход существенно снизит производительность и масштабируемость. Для реальных систем, имеющих тысячи процессоров и управляемых Linux, данный подход неприменим. Этот факт послужил причиной создания Tree RCU.

3.1 Обзор

Будем рассматривать «стандартный» программный интерфейс RCU в комбинации с non-preemptible версии

ей ядра Linux, концентрируясь в основном на примитивах `rcu_read_lock()`, `rcu_read_unlock()` и `synchronize_rcu()`. Основная идея заключается в том, что примитивы чтения RCU являются частью ядра и поэтому в его non-preemptible конфигурациях не блокируются. Поэтому каждый раз, когда ядро процессора простаивает в состоянии бездействия или блокируется в процессе выполнения пользовательских программ, все критические секции чтения RCU, запущенные ранее на этом ядре, оказываются завершенными. Поэтому каждое из этих состояний называется *устойчивым состоянием*. Каждый такой переход через устойчивое состояние сигнализирует об окончании соответствующего grace-периода. Основная сложность заключается в том, чтобы определить момент, когда все необходимые устойчивые состояния были пройдены для данного grace-периода, сохранив при этом высокую производительность и масштабируемость.

Например, использование единой структуры данных для регистрации устойчивых состояний каждого ядра приводит к неприемлемо частому использованию блокировок на крупных системах, что в свою очередь приводит к снижению производительности. Для решения этой проблемы Tree RCU использует иерархическую организацию структур данных, каждый узел которой предназначен для учета устойчивых состояний отдельного ядра и предоставляет свою информацию более высоким уровням. По достижении корня дерева grace-период заканчивается и его информация распространяется по всем узлам-потомкам. Вскоре после того, как узлы получают данную информацию, происходит возврат из `synchronize_rcu()`.

В оставшейся части данного раздела мы рассмотрим реализацию Tree RCU в non-preemptible конфигурации ядра Linux версии 4.3.6. Вначале мы вкратце опишем реализацию примитивов чтения и записи, затем опишем иерархическую структуру данных, используемую для эффективного учета устойчивых состояний, и, наконец, рассмотрим, как RCU использует эту структуру данных для фиксации устойчивых состояний и grace-периодов без учета отдельных потоков-читателей.

3.2 Read/Write-Side Primitives

В non-preemptible версии ядра любая область его исходного кода, которая не использует добровольных блокировок, является неявной критической секцией чтения RCU. В связи с этим, реализации `rcu_read_lock()` и `rcu_read_unlock()` не должны выполнять никакой работы. Действительно, в production сборках ядра с выключенным режимом отладки, эта пара примитивов является пустышками.

В общем случае, когда используется несколько вычислительных ядер процессора, примитив записи `synchronize_rcu()` вызывает `wait_rcu_gp()`, которая является внутренней функцией, использующей механизм callback'ов

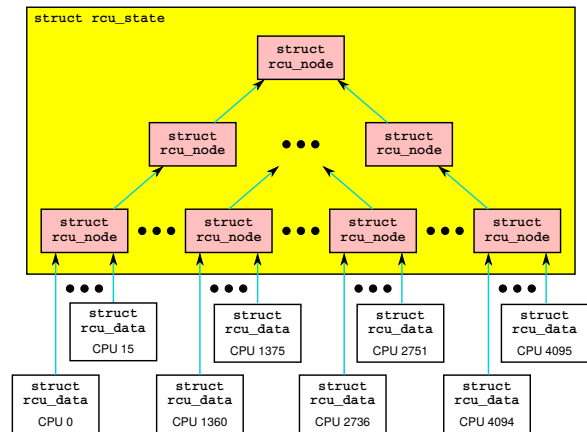


Рис. 3: Иерархия Tree RCU

для отложенного вызова `wakeme_after_rcu()` по окончании некоторого grace-периода. Как подсказывает название, данная функция повторно предназначена для повторного вызова `wait_rcu_gp()`, которая на этот раз ничего не делает, тем самым позволяя `synchronize_rcu()` вернуть управление в вызывающий поток.

3.3 Структуры данных Tree RCU

Глобальное состояние RCU записывается в структуру `rcu_state`, представляющую собой дерево структур `rcu_node` с арностью, равной 64 (32 на 32-битных системах). Каждый терминальный узел данного дерева может иметь ссылки на максимум 64 (32 на 32-битных системах) структуры RCU's global state is recorded in the `rcu_state` structure, which consists of a tree of `rcu_node` structures with a child count of up to 64 (32 in a 32-bit system). Every leaf node can have at most 64 `rcu_data`, каждая из которых представляет собой отдельное ядро процессора, как показано на рисунке 3. Каждая структура `rcu_data` ведет учет устойчивых состояний своего ядра, а `rcu_node`-дерево сначала используется для распространения информации об этих состояниях сначала в направлении корня, затем — для распространения информации о grace-периодах в направлении листьев. Информация об устойчивых состояниях передается на родительский уровень в тот момент времени, когда каждый узел-потомок каждого поддерева данного уровня уже передал её в корень этого поддерева. Эта сзема передачи информации позволяет существенно сократить частоту использования блокировок на верхних уровнях дерева. Например, рассмотрим стандартное `rcu_node` дерево для системы с 4,096 вычислительными ядрами, имеющее 256 терминальных узлов, 4 внутренних узлов и один корневой узел. В течение данного grace-периода, каждое ядро сообщит информацию о своем устойчивом состоянии в соответствующий терминальный узел, но при этом каждому

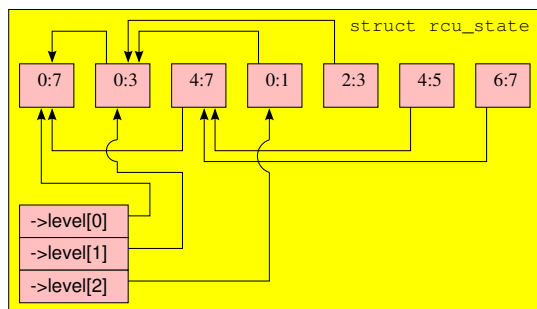


Рис. 4: Представление дерева структур `rcu_node` в виде массива

терминальному узлу будет соответствовать всего 16 сопериначающих ядер. Всего 256 ядер будут пытаться сообщить свои устойчивые состояния внутренним узлам, при этом всего информация 64 ядер дойдет до каждого из четырех внутренних узлов. Наконец, информация всего четырех ядер может дойти до корневого узла, что сказывается на его очень низкой частоте блокирования. Это позволяет использовать данную структуру на очень больших системах. В частности, актуальная реализация RCU ядра Linux поддерживает четырехуровневые деревья, что позволяет использовать до $64^4 = 16,777,216$ ядрами на 64-битных системах.¹

3.3.1 `rcu_state` Structure

Каждый реализация RCU имеет свою собственную структуру `rcu_state`. Структура `rcu_state` включает в себя массив структур `rcu_node`, логически организованных в виде дерева `struct rcu_node node[NUM_RCU_NODES]`, со структурами `rcu_data`, присоединенными к его терминальным узлам. Таким образом, обход этого дерева в ширину сводится линейному проходу по массиву. Еще один массив структур `rcu_node`, `*level[NUM_RCU_LVL]`, используется для указания на самый левый узел каждого уровня дерева, как показано на рисунке 4.

Структура `rcu_state` использует поля `->grpnum` и `->completed` типа **unsigned long** для учета grace-периодов. Поле `->grpnum` используется для отслеживания начала последнего grace-периода, в то время как `->completed` отслеживает окончание последнего grace-периода. Если значения данных полей одинаковы, то соответствующий RCU находится в состоянии по умолчанию. Если же значение `grpnum` больше, чем `completed`, то RCU находится в состоянии grace-периода. Все прочие комбинации недопустимы.

¹ В настоящее время четырехуровневые деревья используются при нагрузочном тестировании, а трехуровневые находят свое применение на промышленных 4096-ядерных системах.

3.3.2 `rcu_node` Structure

Дерево структур `rcu_node` регистрирует и распространяет информацию об устойчивых состояниях от терминальных узлов к корневому, а также распространяет информацию о grace-периодах в обратном направлении. Структура `rcu_node` использует спин-блокировку `->lock` для защиты своих полей. Поле `->parent` содержит указатель на структуру-родителя, при этом значение данного поля у корневого узла равно NULL. Значение поля `->level` соответствует, на котором находится данный узел в дереве, при этом уровень корневого узла равен нулю. Поле `->grpmask` описывает номер бита данного узла в значении поля `->qsmask` узла-родителя. Поля `->grplo` и `->grphi` соответствуют наименьшему и наибольшему порядковому номеру вычислительного ядра, учитываемого данной структурой.

Поле `->qsmask` указывает, какие из узлов-потомков еще не сообщили о своих устойчивых состояниях на данный момент времени. Как и в случае с `rcu_state`, структура `rcu_node` имеет поля `->completed` и `->grpnum`, имеющие такие же значения, как и у родительской структуры `rcu_state`, за исключением начала и конца каждого grace-периода, когда данные значения копируются из корневого узла. Значения этих полей могут быть равны друг другу, либо отличаться на единицу.

3.3.3 Структура `rcu_data`

Структура `rcu_data` используется для учета устойчивых состояний и вызова callback'ов связанного вычислительного ядра. Поскольку доступ к данной структуре осуществляется посредством связанного вычислительного ядра, нет необходимости выполнять синхронизацию. Как и в случае со структурой `rcu_state`, различные реализации RCU поддерживают различные виды структур `rcu_data`. Поле `->cpu` указывает на связанное вычислительное ядро, `->rsp` — на связанную структуру `rcu_state`, а `->mynode` ссылается на соответствующую терминальную структуру `rcu_node`. Значение поля `->grpmask` указывает на позицию структуры `rcu_data` в битовом поле `->qsmask` связанной структуры `rcu_node`.

Структура `rcu_data` содержит поле `->qs_pending`, указывающее, что RCU ожидает получения устойчивого состояния от связанного ядра, и поле `->passed_quiesce`, указывающее на то, что данное ядро уже прошло через устойчивое состояние. Кроме этого, данная структура также имеет поля `->grpnum` и `->completed`, значения которых могут отставать от соответствующих им полей структур `rcu_state` и `rcu_node` в нормальном режиме работы ядер процессора. С другой стороны, если ядра процессора являются заблокированными, их значения могут оставаться на один grace-период.

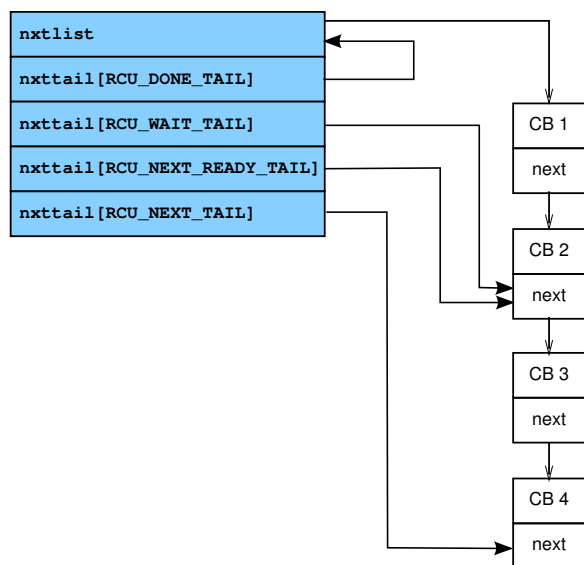


Рис. 5: Очередь callback'ов в rcu_data

от соответствующих значений полей структуры `rcu_node`.

Поля `->grpnum` и `->completed` структуры `rcu_state` содержат наиболее актуальные значения и используются для обновления соответствующих полей родительских структур `rcu_node`, что позволяет сравнивать значения данных полей со значениями этих же полей структур `rcu_node` для фиксации факта начала нового grace-периода. Эта схема позволяет вычислительным ядрам обнаруживать границы grace-периодов без использования блокировок. Структура `rcu_data` управляет RCU callback'ами с помощью структуры данных, известной как четырехсегментный список [?].

3.3.4 Callback'и RCU

Структура `rcu_data` управляет RCU callback'ами, используя указатель `->nxtlist`, указывающим на начало списка, и массив `->nxttail[]` указателей-на-конец, формирующий четырехсегментный список callback'ов [?], где каждый элемент массива `->nxttail[]` указывает на конец соответствующего сегмента, как показано на рисунке 5. Сегмент, оканчивающийся на `->nxttail[RCU_DONE_TAIL]` («RCU_DONE_TAIL сегмент»), содержит callback'и, связанные с предыдущим grace-периодом, и, поэтому, готовые к вызову. Сегменты `RCU_WAIT_TAIL` и `RCU_NEXT_READY_TAIL` содержат callback'и, ожидающие окончания текущего и следующего grace-периодов, соответственно. Наконец, сегмент `RCU_NEXT_TAIL` содержит callback'и, которые еще не были связаны с каким-либо grace-периодом. Поле `->qlen` выполняет учет общего числа callback'ов, а `->blimit` определяет максимальное число callback'ов, которые могут быть вызваны в данный момент времени, тем самым ограничивая размер ок-

на времени, используемого для их вызова, на длинных списках callback'ов.²

Возвращаясь к рисунку 5 отметим, что элемент массива `->nxttail[RCU_DONE_TAIL]` указывает на `->nxtlist`, что означает, что в данный момент ни один из callback'ов не готов к вызову. Элемент `->nxttail[RCU_WAIT_TAIL]` указывает на `->next`-указатель второго callback'a, что означает, что callback'и CB 1 и CB 2 ожидают окончания данного grace-периода. Элемент `->nxttail[RCU_NEXT_READY_TAIL]` указывает на тот же `->next`-указатель, что означает, что список не содержит callback'ов, связанных со следующим grace-периодом. Наконец, callback'и, расположенные между `->nxttail[RCU_NEXT_READY_TAIL]` и `->nxttail[RCU_NEXT_TAIL]` элементами (CB 3 и CB 4), еще не связаны ни с каким grace-периодом. Элемент `->nxttail[RCU_NEXT_TAIL]` всегда указывает либо на последний callback, либо, если весь список пустой, на `->nxtlist`.

Cache locality достигается за счет вызова callback'ов на тех вычислительных ядрах, которые их зарегистрировали. Например, примитив записи `RCU synchronize_rcu()` добавляет callback `wakeme_after_rcu()` в конец списка `->nxttail[RCU_NEXT_TAIL]` на данном вычислительном ядре (Раздел ??). По окончании данного grace-периода, которому соответствует изменение значения поля `->completed` структуры `rcu_data`, меньшего, чем соответствующее значение структуры `rcu_node`, они смещаются на один сегмент списка (с помощью `rcu_advance_cbs()`). Кроме этого, ядро процессора периодически объединяет сегменты `RCU_NEXT_TAIL` и `RCU_NEXT_READY_TAIL` путем вызова `rcu_accelerate_cbs()`. В некоторых специальных случаях, ядро выполняет объединение сегментов `RCU_NEXT_TAIL` и `RCU_WAIT_TAIL`, пропуская сегмент `RCU_NEXT_TAIL`. In a few special cases, the CPU merges the `RCU_NEXT_TAIL` segment into the `RCU_WAIT_TAIL` segment, bypassing the `RCU_NEXT_TAIL` segment. Эта оптимизация применяется в тех случаях, когда ядро начинает новый grace-период. Она не используется, когда ядро обнаруживает новый grace-период, поскольку этот период мог начаться до момента добавления callback'ов в сегмент `RCU_NEXT_TAIL`. Это особенность архитектуры неслучайна: требование обеспечения независимости работы вычислительных ядер (для избегания использования блокировок) является более важным, чем сокращение продолжительности grace-периодов. В тех редких случаях, когда требуются grace-периоды должны быть максимально короткими, требуется использовать `synchronize_rcu_`

²Вычислительные окружения реального времени, требующие выполнения более строгих ограничений времени вызова, должны использовать callback offloading, который находится вне контекста данной статьи.

`expedited()`. Эта функция имеет такую же семантику, как и `synchronize_rcu()`, но предпочитает уменьшение задержки остальным оптимизациям.

Каждый RCU callback представляет собой структуру `rcu_head`, имеющую поле `->next`, указывающее на следующий callback в списке, и поле `->func`, указывающее на функцию, подлежащую вызову по окончании предстоящего grace-периода.

3.4 Quiescent State Detection

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

The non-preemptible RCU-sched flavor's quiescent states apply to CPUs, and are user-space execution, context switch, idle, and offline state. Therefore, RCU-sched only needs to track tasks and interrupt handlers that are actually running because blocked and preempted tasks are always in quiescent states. Thus, RCU-sched needs only track CPU states.

3.4.1 Scheduling-Clock Interrupt

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

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

3.4.2 Context-Switch Handling

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

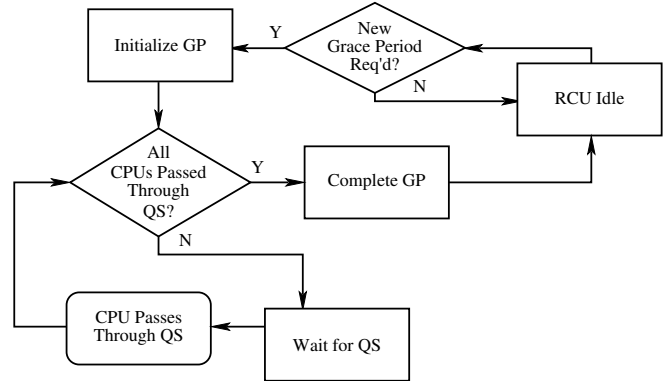


Рис. 6: Grace-Period Detection State Diagram

3.5 Grace Period Detection

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

3.5.1 Softirq Handler for RCU

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

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

Next, `rcu_check_quiescent_state()` checks whether `->qs_pending` indicates that RCU needs a quiescent state from this CPU. If so, it checks whether `->passed_quiesce` indicates that this CPU has in fact

passed through a quiescent state. If so, it invokes `rcu_report_qs_rdp()` to report that quiescent state up the combining tree.

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

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

3.5.2 Grace-Period Kernel Thread

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

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

To clean up after a grace period, `rcu_gp_kthread()` calls `rcu_gp_cleanup()` after setting the `rcu_state`

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

4. Verification Scenario

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

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

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

However, this critical-section exit has no immediate effect. Instead, a later context switch will invoke `rcu_note_context_switch()`, which in turn invokes `rcu_sched_qs()`, recording the quiescent state in the CPU's `rcu_sched_data` structure's `->passed_quiesce`

field. Later, a scheduling-clock interrupt will invoke `rcu_check_callbacks()`, which calls `rcu_pending()` and notes that the `->passed_quiesce` field is set. This will cause `rcu_pending()` to return `true`, which in turn causes `rcu_check_callbacks()` to invoke `rcu_process_callbacks()`. In its turn, `rcu_process_callbacks()` will invoke `raise_softirq(RCU_SOFTIRQ)`, which, once the CPU has interrupts, preemption, and bottom halves enabled, calls `rcu_process_callbacks()`.

As discussed in Section 3.5.1, RCU's softirq handler function `rcu_process_callbacks()` first calls `rcu_check_quiescent_state()` to report any recent quiescent states on the current CPU (CPU 0). Then it checks whether the CPU 0 has passed a quiescent state. Since a quiescent state has been recorded for CPU 0, `rcu_report_qs_rnp()` is invoked to traversal up the combining tree. It clears the first bit of the root `rcu_node` structure's `qsmask` field (recall that the RCU combining tree has only one level). Since the second bit for CPU 1 has not been cleared, the function returns.

Since `synchronize_rcu()` blocks in CPU 1, it will result in a context switch. This triggers a sequence of events similar to that described above for CPU 1, which results in the clearing of the second bit of the root `rcu_node` structure's `->qsmask` field, the value of which is now 0, indicating the end of the current grace period. CPU 1 therefore invokes `rcu_report_qs_rsp()` to awaken the grace-period kthread, which will clean up the ended grace period, and, if needed, start a new one (Section 3.5.2).

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

5. Modeling RCU for CBMC

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

In the remainder of this section we describe how to construct a model from the source code of the Tree RCU implementation in the Linux kernel version 4.3.6, which can be verified by CBMC. Model construction entailed

stubbing out calls to other parts of the kernel, removing irrelevant functionality (such as idle-CPU detection), removing irrelevant data (such as statistics), and adding preprocessor directives to conditionally inject bugs (described in Section 6.1). The Linux kernel environment and the majority of these changes to the source code are made through macros in separate files that can be reused across different versions of the Tree RCU implementation. The biggest change in the source files is to use arrays to model per-CPU data, which could potentially be scripted. The resulting model is C code with assertions that can be also run as a user program, which provides important validation of the model itself.

Initialization

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

Per-CPU Variables and State

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

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

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

Update-Side API `synchronize_sched()`

Because our model omits CPU hotplug and callback handling, we cannot use Tree RCU's normal callback mechanisms to detect the end of a grace period. We

³<http://www.cprover.org/cbmc/>

therefore use a global variable `wait_rcu_gp_flag`, which is initialized to 1 in `wait_rcu_gp()` before the grace period. Because `wait_rcu_gp()` blocks, it can result in a context switch, the model invokes `rcu_note_context_switch()`, followed by a call to `rcu_process_callbacks()` to inform RCU of the resulting quiescent state. When the resulting quiescent states propagate to the root of the combining tree, the grace-period kthread is awakened. This kthread then invokes `rcu_gp_cleanup()`, the modeling of which is described below. Then `rcu_gp_cleanup()` calls `rcu_advance_cbs()`, which invokes `pass_rcu_gp()` to clear the `wait_rcu_gp_flag` flag. The `__CPROVER_assume(wait_rcu_gp_flag == 0)` in `wait_rcu_gp()` prevents CBMC from continuing execution until `wait_rcu_gp_flag` is equal to 0, thus modeling the needed grace-period wait.

Scheduling-Clock Interrupt and Context Switch

The `rcu_check_callbacks()` function detects idle execution, usermode execution, and to invoke RCU core processing in response to state changes. Because we model neither idle nor usermode execution, the only state changes are quiescent states and the beginnings and ends of grace periods. We therefore dispense with `rcu_check_callbacks()` (Section 3.5.1). Instead, we directly call `rcu_note_context_switch()` just after releasing a CPU, which in turn calls `rcu_sched_qs()` to record the quiescent state. Finally, we call `rcu_process_callbacks()`, which notes grace-period beginnings and ends and reports quiescent states up RCU's combining tree.

Grace-Period Kthread

As discussed in Section 3.5.2, `rcu_gp_kthread()` invokes `rcu_gp_init()`, `rcu_gp_fqs()`, and `rcu_gp_cleanup()` to initialize, wait for, and clean up after each grace period, respectively. To reduce the size of the formula generated by CBMC, instead of spawning a separate thread, we directly call `rcu_gp_init()` from `rcu_spawn_gp_kthread` and `rcu_gp_cleanup()` from `rcu_report_qs_rsp()`. Because we model neither idle nor usermode execution, we need not call `rcu_gp_fqs()`.

Kernel Spin Locks

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

Limitations

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

6. Experiments

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

6.1 Bug-Injection Scenarios

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

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

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

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

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

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

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

Bug 5 In this bug-injection scenario, CPUs remain aware of the need for quiescent states. However, CPUs are prevented from recording their quiescent states, thus preventing grace periods from ever completing. To accomplish this, we modify function `rcu_sched_qs()` to return immediately (line 246 in `tree.c`), so that quiescent states are not recorded. Grace periods therefore never complete, which constitutes a liveness violation similar to Bug 2.

Bug 6 In this bug-injection scenario, CPUs are aware of the need for quiescent states, and they also record them locally. However, they are prevented from reporting them up the `rcu_node` tree, which again prevents grace periods from ever completing. This bug modifies function `rcu_report_qs_rnp()` to return immediately (line 2227 in `tree.c`). This prevents RCU from walking up the `rcu_`

node tree, thus preventing grace periods from ending. This is again a liveness violation similar to Bug 2.

Bug 7 Where Bug 6 prevents quiescent states from being reported up the `rcu_node` tree, this bug-injection scenario causes quiescent states to be reported up the tree prematurely, before all the CPUs covered by a given subtree have all reported quiescent states. To this end, in `rcu_report_qs_rnp()`, we remove the `if`-block checking for `rnp->qsmask != 0 || rcu_preempt_blocked_readers_cgp(rnp)` (line 2251 in `tree.c`). Then the tree-walking process will not stop until it reaches the root, resulting in too-short grace periods. This is therefore a safety violation similar to Bug 1.

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

6.2 Validating the RCU Model in User-Space

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

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

As expected, for scenario Prove, the user program ran to completion successfully in all runs. For Prove-GP, it was able to detect the end of a grace period by triggering an assertion violation in all the runs. For Bug 1, an assertion violation was triggered in 559 out of 1000 runs. For Bugs 2–6, the user program timed out in all the runs, thus a grace period did not complete. For Bug 7 with one reader thread, the testing harness failed to trigger an assertion violation. However, we were able to observe a failure in 242 out of 1000 runs with two reader threads.

6.3 Getting CBMC to work on Tree RCU

We have found that getting CBMC to work on our RCU model is non-trivial due to Tree RCU’s complexity combined with CBMC’s bit-precise verification. In fact, early attempts resulted in SAT formulas that were so large that CBMC ran out of memory. After the optimizations described in the remainder of this section, the largest formula contained around 90 million variables and 450 million clauses, which enabled CBMC to run to completion.

First, instead of placing the scheduling-clock interrupt in its own thread, we invoke functions `rcu_note_`

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

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

`context_switch()` and `rcu_process_callbacks()` directly, as described in Section 5. Also, we invoke `__note_gp_changes()` from `rcu_gp_init()` to notify each CPU of a new grace period, instead of invoking `rcu_process_callbacks()`.

Second, the support for linked lists in CBMC version 5.4 is limited, resulting in unreachable code in CBMC’s symbolic execution. Thus, we stubbed all the list-related code in our RCU model, including those for callback handling.

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

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

Finally, since our test harness only requires one `rcu_node` structure and two `rcu_data` structures, we can use 32-bit encodings for `int`, `long`, and pointers by using the command line option `--ILP32`. This reduces CBMC’s formula size by half compared to the 64-bit default.

6.4 Results and Discussion

Table 2 presents the results of our experiments applying CBMC version 5.4 to verify our RCU model. Scenario Prove verifies our RCU model without bug injection over Sequential Consistency (SC). We also exercise the model over the weak memory models TSO and PSO in scenarios Prove-TSO and Prove-PSO, respectively. Scenario Prove-GP performs the same reachability check as in Section 6.2

over SC. We perform the same reachability verification over TSO and PSO in scenarios Prove-GP-TSO and Prove-GP-PSO, respectively. Scenarios Bug 1–7 are the bug-injection scenarios discussed in Section 6.1, and are verified over SC, TSO and PSO. Columns 2–4 give the number of constraints (symbolic program expressions and partial orders), variables, and clauses of the generated formula. The next three columns give the maximum (virtual) memory consumption, solver runtime, and total runtime of our experiments. The final column gives the verification result.

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

In our experiments, CBMC returned all the expected results except for Bug 7, for which it failed to report a violation of the assertion `assert(r2 == 0 || r1 == 1)` with one RCU reader thread running over SC. This failure was due to the approximation of the scheduling-clock interrupt by a direct function call, as described in Section 5. However, CBMC did report a violation of the assertion either when two RCU reader threads were present or when run over TSO or PSO. All of these cases decrease determinism, which in turn more faithfully model non-deterministic scheduling-clock interrupts, allowing the assertion to be violated.

CBMC took more than 9 hours to verify our model over SC (scenario Prove). The resulting SAT formulas have more than 5m constraints, 30m variables and 149m clauses, and occupy 23 GB of memory. The formulas for scenarios Prove-TSO and Prove-PSO are about 40% larger than the scenario Prove. They have more than 40m variables and 200m clauses, and took more than 11 hours and 33 GB memory to solve. Although this verification consumed considerable memory and CPU, it verified all possible executions and reorderings permitted by TSO and PSO, a tiny subset of which are reached by the `rcutorture` test suite.

CBMC proved that grace periods can end (i.e., `assert(0)` is violated), over SC (Prove-GP), TSO (Prove-GP-TSO), and PSO (Prove-GP-PSO). The sizes of resulting formulas

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

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

Таблица 2: Experimental Results of CBMC

and memory consumption are similar to those of the three Prove scenarios. However, it took CBMC only about 4, 13, and 8.5 hours to find an violation of `assert(0)` in Prove-GP, Prove-GP-TSO, and Prove-GP-PSO, respectively.

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

Figures 7–8 compare the formula size between SC, TSO and PSO. Comparison of runtime and memory can be found in Figures 10 and 11. As we can see, the runtime and memory overhead for the TSO and PSO variants of a given experiment are quite similar. The overheads of TSO are slightly higher than those of PSO in all bug-injection scenarios except for Bug 7 on which PSO had longest runtime. However, the overhead of TSO and PSO

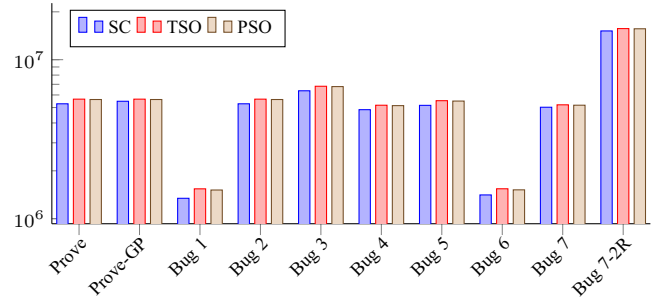


Рис. 7: Number of Constraints in the SAT Formulas

is significantly larger than that of SC, with up to 340% (Bug 6 runtime) and 50% (Bug 1 memory) increases. The runtime was 5–19 hours and memory consumption exceeded 31 GB in all scenarios except Bug 1 and 6. The numbers of variables and clauses are 37m–49m and 188m–245m, respectively, around 130% greater than SC.

The two-reader variant of Bug 7 has by far the longest runtime, consuming more than 19 hours and 78 hours over SC and TSO, respectively, comparing to 9 hours

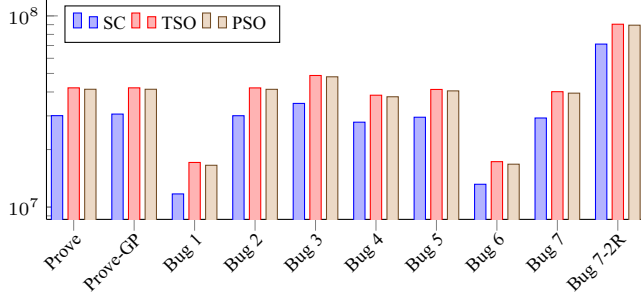


Рис. 8: Number of Variables in the SAT Formulas

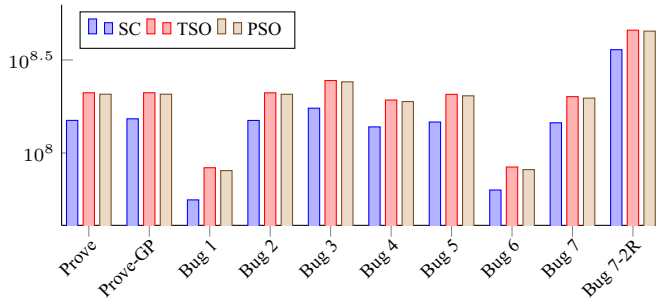


Рис. 9: Number of Clauses in the SAT Formulas

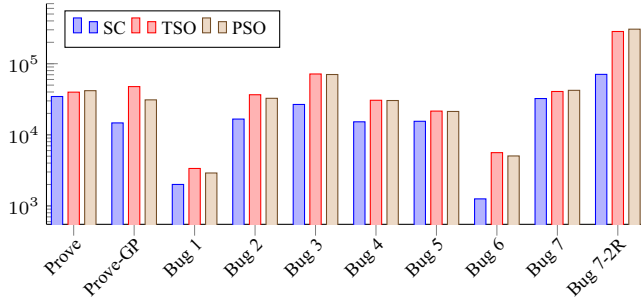


Рис. 10: Total Runtime in Seconds

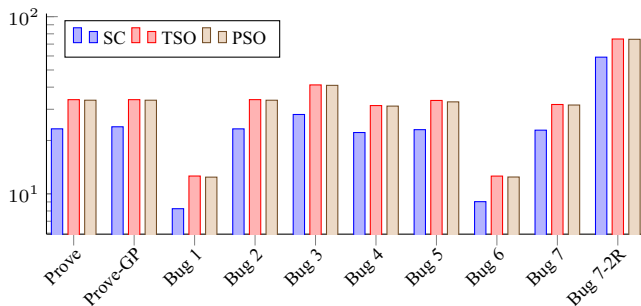


Рис. 11: Maximum Memory Consumption in Gigabytes

and 11 hours with one reader. It also consumed about 75 GB memory, more than double the one-reader variant. For PSO, with two reader threads CBMC's solver ran out of memory after 85 hours whereas with one reader it completed in less than 12 hours. The increased overhead is due to the additional RCU reader's call to `rcu_process_callbacks()`. This in turn results in more than a 125% increase in the number of constraints, variables, and clauses. For example, the two-reader TSO formula has triple the constraints and double the variables and clauses of the one-reader case.

7. Related Work

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

These efforts require an error-prone translation from C to SPIN's modeling language, and therefore are not appropriate for regression testing. By contrast, our work constructs an RCU model directly from its source code from the Linux kernel, and verifies it using automated verification tool.

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

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

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

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

Gotsman et al. [?] use an extended concurrent separation logic to formalise the concept of grace period and prove an abstract implementation of RCU over SC. Tassarotti et al. [?] use GPS, a recently developed program logic for the C/C++11 memory model, to carry out a formal proof of a simple implementation of user-level RCU for a singly-linked list assuming "release-acquire" semantics, which is weaker than SC but stronger than memory models used

by real-world RCU implementations. These formal proofs were performed manually on simple implementations of RCU. By contrast, our work applies an automated verification tool with a test harness to verify the grace-period property of a real-world implementation of RCU over SC, TSO, and PSO.

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

8. Conclusion

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

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

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

This work demonstrates the nascent ability of SAT-based formal-verification tools to handle real-world production-quality synchronization primitives, as exemplified by Linux-kernel Tree RCU on weakly ordered TSO and PSO systems. Although modeling weak ordering incurs a significant performance penalty, this penalty is not excessive. We therefore hypothesize that use of these tools for highly concurrent multithreaded software will reach mainstream within 3-5 years, especially given recent rates of improvement.