Models of RCU Read-Side Critical Sections and Grace Periods

Paul E. McKenney
paulmck@linux.vnet.ibm.com

December 7, 2015

1 Introduction

RCU is a synchronization mechanism that is used heavily in the Linux kernel, perhaps most notably as a high-performance replacement for reader-writer locking for linked structures [4]. In this use case, *RCU read-side critical sections* are delimited by `rcu_read_lock()` and `rcu_read_unlock()`. The `rcu_read_lock()` and `rcu_read_unlock()` primitives are extremely lightweight, having exactly zero overhead in server-class builds of the Linux kernel (`CONFIG_PREEMPT=n`). Of course, this means that RCU updaters cannot exclude RCU readers, which means that updaters must take care to avoid disrupting readers. Updaters avoid read-side disruption via use of `synchronize_rcu()`, which waits for all pre-existing readers to complete. The period of time required for all pre-existing readers to complete is termed a *grace period*.

As noted earlier, RCU is used primarily for linked data structures. However, this paper will use simple litmus tests to simplify demonstration of the relationships between RCU read-side critical sections and grace periods, as exemplified by Figure 1. This litmus test demonstrates that an RCU read-side critical section cannot completely overlap a grace period, which is to be expected given that a grace period must wait for all pre-existing RCU readers to complete [3]. The diagram below the litmus test illustrates this: If Thread 1's read see Thread 0's write, then Thread 0's RCU read-side critical section cannot extend beyond the end of Thread 1's grace period, and therefore Thread 1's write cannot affect the value read by Thread 0's read.

Thread 0	Thread 1
`rcu_read_lock();`	`r1 = READ_ONCE(b);`
`r1 = READ_ONCE(a);`	`synchronize_rcu();`
`WRITE_ONCE(b, 1);`	`WRITE_ONCE(a, 1);`
`rcu_read_unlock();`	

`BUG_ON(0:r1 == 1 && 1:r1 == 1);`
(Cycle prohibited)

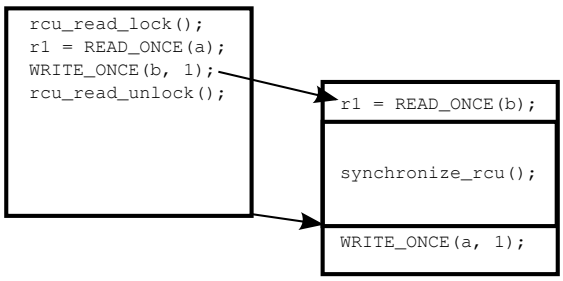

Figure 1: Sample RCU Release-Acquire Litmus Test

2 Modeling RCU

@@@ Roadmap

2.1 Consequences of Fundamental RCU Guarantee

As noted in the previous section, RCU's fundamental guarantee is that any given RCU read-side critical section cannot completely overlap any given RCU grace period. This guarantee can be restated as follows:

1. If any part of a given RCU read-side critical section precedes the beginning of a given RCU grace period, then that entire critical section must precede the end of that grace period. This rule is illustrated by the litmus test in Figure 5 on page 7, as are the next two rules.

2. If any part of a given RCU read-side critical section follows the end of a given RCU grace period, then that entire critical section must follow the beginning of that grace period.

3. It is possible for an RCU grace period to completely overlap a given RCU read-side critical section, so that the grace period starts before the critical section begins and ends after the critical section ends. However, as a consequence of the previous two rules, it is not possible for an RCU read-side critical section to completely overlap a given grace period.

4. If a given RCU read-side critical section is constrained to precede the end of a given RCU grace period, then any other RCU read-side critical section preceding the given critical section in program order is also constrained to precede the end of that same grace period.[1] This rule and the next rule are illustrated by the litmus test in Figure 10 on Page 10.

5. If a given RCU read-side critical section is constrained to follow the beginning of a given RCU grace period, then any other RCU read-side critical section following the given critical section in program order is also constrained to follow the beginning of that same grace period.[2]

6. RCU read-side critical sections impose no ordering other than that specified by the preceding items. In particular, in a program that had no RCU grace periods, RCU read-side critical sections would have no effect, as illustrated by the litmus tests in Figure 8 on Page 9.

7. In the absence of RCU read-side critical sections, RCU grace periods have the same ordering properties as do full memory barriers (`smp_mb()` for the Linux kernel, or `f[mb]` in LISA). That said, if a cycle in a given litmus test is forbidden without RCU readers, adding `rcu_read_lock()` and `rcu_read_unlock()` (`f[lock]` and `f[unlock]` in LISA) to existing processes cannot cause that cycle to be allowed.

The only way to detect the relationship between a given RCU read-side critical section and a given grace period is by means of accesses to shared variables within the critical section and surrounding the grace period.

This suggests a modeling strategy of discarding any execution that fails to adhere to this restated guarantee.

One way to detect that a given RCU read-side critical section starts after a given RCU grace period starts is so have the grace period start by executing a memory barrier (`smp_mb()`, in Linux kernel parlance), followed by an assignment to variable specific to that grace period (`WRITE_ONCE(gpstart0, 1)`, in Linux kernel parlance). The critical section could then begin by reading from this variable, and, if that read returned the value one, executing a memory barrier. The combination of these two memory barriers ensures that if the RCU read-side critical section

[1] This constraint does not apply to code outside of an RCU read-side critical section. That said, all implementations I am aware of would also constrain *all* prior code, whether within an RCU read-side critical section or not.

[2] Similar to the previous rule, this has no effect on code not within an RCU read-side critical section, but, again similar to the previous rule, all known implementations constrain all code.

starts after the grace period does, then the code in the critical section will see the results of all code preceding that grace period, as required.

Similarly, one way to detect that a given RCU read-side critical section ends before a given RCU grace period ends is so have the grace period end by executing an assignment to variable specific to that grace period (`WRITE_ONCE(gpend0, 1)`, in Linux kernel parlance), followed by a memory barrier (`smp_mb()`, in Linux kernel parlance). The critical section could then end by reading from this variable, and, if that read returned the value one, executing a memory barrier *before* the read.[3] The combination of these two memory barriers ensures that if the RCU read-side critical section ends before the grace period does, then the code following that grace period will see the results of all code with that critical section, as required.

Thread 0	Thread 1
`/* rcu_read_lock(); */`	`r1 = READ_ONCE(b);`
`r10 = READ_ONCE(gpstart0);`	`/* synchronize_rcu(); */`
`if (r10)`	`smp_mb();`
`  smp_mb();`	`WRITE_ONCE(gpstart0, 1);`
`/* rcu_read_lock(); */`	`smp_mb();`
`r1 = READ_ONCE(a);`	`WRITE_ONCE(gpend0, 1);`
`WRITE_ONCE(b, 1);`	`smp_mb();`
`/* rcu_read_unlock(); */`	`/* synchronize_rcu(); */`
`if (!future_value(r11))`	`WRITE_ONCE(a, 1);`
`  smp_mb();`	
`r11 = READ_ONCE(gpend0);`	
`/* rcu_read_unlock(); */`	

```
BUG_ON(0:r1 == 1 && 1:r1 == 1 && (r10 != 0 || r11 != 1));
```
(Cycle prohibited)

Figure 2: Detecting Grace Period Beginning and End

Figure 2 shows a straightforward (if wildly counter-intuitive) conversion of the code in Figure 1 to this model. Thread 1's `synchronize_rcu()` is replaced with the code between the two comments, which executes a memory barrier, writes to `gpstart0`, executes another memory barrier, writes to `gpend0`, and finally executes one last memory barrier.

[3] No, that is *not* a typo, the memory barrier absolutely must be executed before the read that determines whether or not the barrier is required. And no, it is not sufficient to just unconditionally execute the memory barrier, as that will result in oversynchronization of the RCU read-side critical section with subsequent code.

Thread 0's `rcu_read_lock()` is replaced by the code between the two comments, which reads `gpstart0`, and if the read returns the value one, executes a memory barrier. In the case where the value one is returned, the following write to `b` is guaranteed not to affect Thread 1's prior load, as required.

Similarly, Thread 0's `rcu_read_unlock()` is replaced by an `if` statement that checks the future value of `r11` using a highly convenient but entirely mythical `future_value()` function, and if that value is to be zero, executes a memory barrier. Either way, the `r11` is then read from `gpend0`. In the case where zero is read, the prior read from `a` is guaranteed not to see Thread 1's later store, as required.

Of course, Thread 1's code sequence is not actually providing RCU's grace-period guarantee. There is nothing to stop Thread 0's write to `b` from preceding Thread 1's read, nor is there anything to stop Thread 0's read from `a` from following Thread 1's write. For example, either the compiler or the CPU could reorder Thread 0's read from `a` with its write to `b`. Then Thread 0 could execute up throught its write to `b`, Thread 1 could execute in its entirety, and finally Thread 0 could continue its execution to completion. This would trigger the `BUG_ON()` in Figure 1, even though RCU would prevent that outcome. This possibility is handled by an additional clause in the `BUG_ON()` in Figure 2. This clause suppresses the `BUG_ON()` if Thread 0's read-side critical section completely overlapped with Thread 1's grace period, that is, if `r10` is zero and `r11` is one.

This is a common modeling technique. Where a real implementation would be required to prevent something from happening, a model can instead discard any execution where that something is seen to occur.

However, it is necessary to implement the counter-temporal `future_value()` primitive. This is discussed in the next section.

2.2 Counter-Temporal Models???

Just to be clear, a real implementation of RCU (or, for that matter, of pretty much anything else) cannot be counter-temporal, at least given our current un-

derstanding of physics.[4] However, counter-temporal *models* of RCU can function quite well. For example, the model can enumerate candidate scenarios or executions, and then discard any that violate the counter-temporal constraints. This is exactly the same technique described in the previous section to handle RCU's fundamental guarantee.

2.3 Counter-Temporal Model Description

	Grace Period	RCU Read-Side Critical Section
1.	$\text{remove}_0 = 1$;	
2.	smp_mb();	
3.	$\text{gpstart}_0 = 1$;	$\text{r1}_i = \text{gpstart}_i$;
4.		if (r1_i)
5.		smp_mb();
		...
6.		rcu_read_lock();
7.		$\text{r2} = \text{remove}_0$;
8.	...	$\text{r3} = \text{reclaim}_0$;
9.		rcu_read_unlock();
		...
10.		$\text{p}_i = \text{random}()$;
11.		if ($!\text{p}_i$)
12.		smp_mb();
13.		$\text{r4}_i = \text{gpend}_i$;
14.	$\text{gpend}_0 = 1$;	
15.	smp_mb();	
16.	$\text{reclaim}_0 = 1$;	

Constraints:
BUG_ON(!r2 && r3 && p_i == r4_i && (r1_i || $!\text{r4}_i$));

Figure 3: Counter-Temporal Model of RCU

Figure 3 shows a rough model of how RCU grace periods and RCU read-side critical sections interact. This requires a backwards-in-time flow of data through $r4_i$, which is handled via a *prophesy variable* p_i. This prophesy variable is randomly generated, and part of the assertion checks that the prophesy was correct, so that any execution involving an incorrect prophesy is discarded. The `random()` function can be replaced by an unordered store from an additional thread.

This table models a grace period in lines 2-15 of the second column and an RCU read-side critical section in lines 6-9 of the third column. Note that the `rcu_read_lock()` and `rcu_read_unlock()` do not actually do anything, and are included strictly for readability. Given that the point of this model is to be independent of any particular implementation, that implementation is represented by the "..." in the grace-period column.

The assignment to the "remove" array elements on line 1 represents the removal actions that normally precede a grace period: In conventional RCU-protected linked data structures, this assignment models removing an element from the structure, thereby making it inaccessible to readers. Similarly, the assignment to the "reclaim" array element on line 16 represents the reclamation actions that normally follow the grace period: In conventional RCU-protected linked data structures, these assignments model freeing the data element that was removed prior to the grace period. In other RCU use cases, this assignment represents whatever action is taken preceding and following the grace period.

The assignments to the *auxilliary variables* "gpstart" (line 3) and "gpend" (line 14) array elements model the beginning and end, respectively, of the grace period. The `smp_mb()` statements force full ordering at the boundaries of the grace period. In the model, another `smp_mb()` is required somewhere between lines 3 and 14.

Turning now to the model of the RCU read-side critical section in the third column, let's start at the `rcu_read_lock()` statement on line 6, which in the model as in the program represents the start of the RCU read-side critical section. The assignments to the "r2" and "r3" variables capture the state of the modifications before and after the grace period, respectively. Finally, both in the model and in the code, the `rcu_read_unlock()` represents the end of the RCU read-side critical section. Of course,

[4] Yes, there was that incident a few years ago involving neutrinos, but that turned out to instead be measurement error.

RCU read-side critical sections are not permitted to span grace periods, hence the last clause of the constraint, which disables the `BUG_ON()` statement for any execution where this occurs.

Now, if any part of this RCU read-side critical section unambiguously precedes a given grace period (in this case, there is of course only grace period 0), all statements within that critical section must happen before the reclamation following that grace period. This is modeled by the `smp_mb()` on line 12 (conditioned on p_i). The constraint "$p_i == r4_i$" comes into play here, ensuring that any executions with an incorrect prophesy disable the `BUG_ON()` statement. Note that lines 10-13 are set up for an arbitrarily large number of grace periods, when we in fact have only one in this example.

Similarly, if this RCU read-side critical section unambiguously follows a given grace period, all the statements within that critical section must see all statements preceding that grace period. This is modeled by the `smp_mb()` on line 5 (conditioned on $r1_i$). Now, if $r1_i$ indicates that this RCU read-side critical section will see a given grace period's reclamation, then the `smp_mb()` must be executed, ensuring that it also sees anything preceding that grace period.

The conditional execution of the `smp_mb()` full memory barriers must not be thought of as a singular event. In the grace-period-before case, the only real requirement is that there be a full memory barrier somewhere after the start of a given grace period and the start of any RCU read-side critical section that sees that grace period's reclamation phase. Similarly, in the grace-period-after case, the only real requirement is that there be a full memory barrier somewhere before the end of a given grace period and the end of any RCU read-side critical section that precedes that grace period's removal phase. A single `smp_mb()` might suffice for an arbitrarily large number of grace periods (as is the case in practice with entry to or exit from idle in the Linux kernel), and on the other hand each of several grace periods might each have its own `smp_mb()` to ensure proper ordering with a given RCU read-side critical section. These considerations explain the "..." before the `rcu_read_lock()` and after the `rcu_read_unlock()`: These represent the arbitrary amount of time and processing that might occur between the RCU read-side critical section and any needed memory barriers.

This gets more complex for threads that have multiple RCU read-side critical sections, which can of course interact differently with the grace period. It gets even more complex given multiple RCU grace periods and multiple RCU read-side critical sections.

2.4 Handling Multiple Grace Periods

In the worst case, each grace period might need to update a set of begin/end variables for each thread. However, let's start with the Linux kernel, which serializes grace periods, so that all CPUs agree that a given grace period ends before its successor grace period starts. Of course, serialization is normally an great way to destroy any semblance of performance and scalability, but the Linux kernel compensates for serialization with a technique called *batching*. The insight behind batching is that a single grace period can handle an arbitrarily large number of prior requests. In fact, in the Linux kernel, it is not unusual for a single grace period computation to satisfy more than a thousand requests [2].

Because of this serialization, it suffices for each grace period to maintain a single set of variables, regardless of how many threads, other grace periods, or RCU read-side critical sections there are.

2.5 Handling Multiple Read-Side Critical Sections

In general, each read-side critical section must interact with each grace period, with the interaction taking place at the `rcu_read_lock()` and the `rcu_read_unlock()`. But this is not sufficient in general. If there are statements before a given RCU read-side critical section, interacting solely at the site of the `rcu_read_lock()` would result in oversynchronization because there would always be a full memory barrier at that location any time that the RCU read-side critical section followed the start of any grace period. It is therefore necessary to insert checks before the `rcu_read_lock()` as well as at the `rcu_read_lock()`.

Similarly, it is necessary to insert checks after the `rcu_read_unlock()` as well as at the `rcu_read_unlock()`.

Fortunately, we can again specialize the model to the class of RCU implementations in the Linux kernel, in which a given grace period interacts with each thread at a given point in the code, using either an inter-processor interrupt or a pre-existing quiescent state. This means that these checks can simply be interleaved with the code; there is no need to produce permutations of the code and then interleave the checks.

Please note that this approach ignores the possibility of code reordering due to compiler optimizations. Because compiler optimizations are ignored, there will be some litmus tests for which the auxiliary-variable approach gives the wrong answer.

2.6 Handling Nested RCU Read-Side Critical Sections

A nested set of RCU read-side critical sections may simply be flattened to a single large RCU read-side critical section. For example, the inner `rcu_read_lock()` and `rcu_read_unlock()` invocations can simply be commented out when translating to auxiliary-variable form.

In the first version, the remaining RCU read-side critical section must span the entire thread unless the litmus test has no grace periods.

2.7 Omitting Checks

Only threads that have RCU read-side critical sections need check for grace periods.

RCU read-side critical sections in a given thread need not check for grace periods in that same thread. Given that the first version only allows RCU read-side critical sections that span an entire thread, the only case where a grace period and a read-side critical section can appear in a single thread is the deadlock case where the grace period appears within the RCU read-side critical section, as shown in Figure 4.

Checks need not be inserted before operations that do not affect memory or memory ordering.

```
 1 LISA LISA1R1Gdeadlock
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 }
 6  P0            ;
 7  f[lock]       ;
 8  w[once] x0 1  ;
 9  f[sync]       ;
10  w[once] x1 1  ;
11  f[unlock]     ;
12 exists (x0=1 /\ x0=1)
```

Figure 4: RCU Self-Deadlock Litmus Test

Checks against prior RCU grace periods need only be inserted up to the beginning of the last RCU read-side critical section. Similarly, checks against subsequent RCU grace periods need only be inserted after the end of the first RCU read-side critical section. In the first version, this means that there will be one set of checks for prior RCU grace periods at the beginning of a thread containing an RCU read-side critical section and another set of checks for subsequent RCU grace periods at the end of that thread.

Processes that are completely spanned by a single RCU read-side critical section only read a given prophesy variable once, and therefore do not need to check consistency across multiple reads. Such processes need only check the accuracy of that one prophesy against the read of the corresponding `gpend` variable.

2.8 Generating Prophesies

As noted in Section 2.3, line 10 of the counter-temporal model shown in Figure 3 generates a random number generator to provide a prophesy for the value of $r4_i$. Recall that this is required so that line 12's memory barrier can be executed before $r4_i$ is fetched, but only in cases when the value that is to be fetched will turn out to be zero.

Although it should be no problem generating random numbers in a modeling tool, there is a simpler way. The trick is to create an auxiliary thread that stores the value zero to each prophesy variable, as in "$p_i = 0$". Because this auxiliary thread will have no memory-ordering directives of any kind, the modeling

```
 1 LISA LISA1R1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 }
 6  P0            | P1            ;
 7  f[lock]       | r[once] r1 x1 ;
 8  w[once] x0 1  | f[sync]       ;
 9  w[once] x1 1  | r[once] r2 x0 ;
10  f[unlock]     |               ;
11 exists (1:r1=1 /\ 1:r2=0)
```

Figure 5: Canonical RCU Litmus Test (1R1G Ø)

tool will automatically analyze all possible orders of stores and all possible combinations of propagation speeds to the reading thread.

However, if the litmus test has multiple RCU read-side critical sections, it is also necessary that the assignment be momentary, so that the "$p_i = 0$" is followed by "$p_i = 1$". This allows one thread preceding a grace period to execute the required memory, while also allowing another thread following that same grace period to avoid doing so.

If all grace periods are executed by a single thread, then they are strongly ordered. Therefore, the pair of stores to a given grace period's prophesy variables can be separated from the other pairs with a full memory barrier (LISA `f[mb]`).

3 Litmus Tests and Lessons Learned

This section takes a tour through a loosely related set of RCU litmus tests. Section 3.1 looks at tests with one reader and one grace period, Section 3.2 looks at tests with several readers but no grace period, Section 3.3 looks at tests with two readers and one grace period, Section 3.4 looks at tests with three readers and one grace period, Section 3.5 looks at tests with two readers and two grace periods, Section 3.6 looks at tests with three readers and two grace periods, and finally, Section 3.7 discusses solver effectiveness.

```
 1 LISA LISA1R1n1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 }
 6  P0            | P1            ;
 7  f[lock]       | r[once] r1 x1 ;
 8  w[once] x0 1  | f[sync]       ;
 9  f[lock]       | r[once] r2 x0 ;
10  w[once] x1 1  |               ;
11  f[unlock]     |               ;
12  f[unlock]     |               ;
13 exists (1:r1=1 /\ 1:r2=0)
```

Figure 7: Canonical RCU Litmus Test With Nesting (1R1n1G Ø)

3.1 Litmus Tests With One Reader and One Grace Period

The canonical RCU litmus test, named 1R1G, is shown in Figure 5, which is an RCU-mediated form of message passing. Because P0's RCU read-side critical section cannot span P1's grace period, both of P0's stores must happen either before P1's second read or after P1's first read. Therefore, if P1's read from `x0` returns the value one, P1's read from `x1` must do so as well. In other words, the cycle connecting P0's `x1` write to P1's `x1` read to P1's `x0` write to P0's `x0` read and back to P0's `x1` write is prohibited. For later litmus tests, "cycle prohibited" (or Ø in tables) indicates that the counter-intuitive cycle cannot happen, and "cycle allowed" indicates that it can.

Figure 6 on page 8 shows this same litmus test, but converted to auxiliary-variable form. This conversion replaces `rcu_read_lock()` (AKA `f[lock]`), `rcu_read_unlock()` (AKA `f[unlock]`), `synchronize_rcu()` (AKA `f[sync]`) with the sequences of loads, stores, memory barriers, comparisons, branches, and labels required to emulate RCU. The added code is flagged with comments, as is the extent of the RCU read-side critical sections. Additional terms are also added to the `exists` clause. The resulting code corresponds closely to that shown in Figure 3.

Figure 7 adds a nested RCU read-side critical section. Because nested RCU read-side critical sections are flattened, this is equivalent to the litmus test shown in Figure 5.

```
 1 LISA LISA1R1G-Auxiliary
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 gpstart01=1;
 6 proph01=1;
 7  0:r1001=1; 1:r1001=1;
 8  0:r101200=1;
 9 }
10  P0                        | P1                  | P2                ;
11  (* f[lock] *)             | r[once] r1 x1       | w[once] proph01 0 ;
12  (* preamble 1 *)          | (* GP 1 *)          | w[once] proph01 1 ;
13  r[once] r101000 gpstart01 | f[mb]               |                   ;
14  b[] r101000 GPSS01010     | w[once] gpstart01 0 |                   ;
15  f[mb]                     | f[mb]               |                   ;
16  (* mov r100901 1 *)       | w[once] gpend01 1   |                   ;
17  GPSS01010:                | f[mb]               |                   ;
18  (* end preamble 1 *)      | (* end GP 1 *)      |                   ;
19  w[once] x0 1              | r[once] r2 x0       |                   ;
20  w[once] x1 1              |                     |                   ;
21  (* f[unlock] *)           |                     |                   ;
22  (* postamble 1 *)         |                     |                   ;
23  r[once] r101200 proph01   |                     |                   ;
24  b[] r101200 GPES01010     |                     |                   ;
25  f[mb]                     |                     |                   ;
26  (* mov r100801 1 *)       |                     |                   ;
27  GPES01010:                |                     |                   ;
28  r[once] r101100 gpend01   |                     |                   ;
29  (* end postamble 1 *)     |                     |                   ;
30 exists ((1:r1=1 /\ 1:r2=0)
31  /\ 0:r101100=0:r101200
32  /\ (0:r101000=0 \/ 0:r101100=0))
```

Figure 6: RCU Canonical Litmus Test: Auxiliary-Variable Form (1R1G Ø)

```
 1 LISA LISA4R2t
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 }
 6  P0            | P1            ;
 7  f[lock]       | f[lock]       ;
 8  w[once] x0 1  | r[once] r1 x1 ;
 9  f[unlock]     | f[unlock]     ;
10  f[lock]       | f[lock]       ;
11  w[once] x1 1  | r[once] r2 x0 ;
12  f[unlock]     | f[unlock]     ;
13 exists (1:r1=1 /\ 1:r2=0)
```

Figure 8: RCU Readers Impose No Ordering (4R2t)

```
 1 LISA LISA2R1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 }
 7  P0            | P1            | P2            ;
 8  f[lock]       | r[once] r1 x1 | f[lock]       ;
 9  r[once] r2 x0 | f[sync]       | r[once] r2 x2 ;
10  w[once] x1 1  | w[once] x2 1  | w[once] x0 1  ;
11  f[unlock]     |               | f[unlock]     ;
12 exists (1:r1=1 /\ 2:r2=1 /\ 0:r2=1)
```

Figure 9: Grace Period Outnumbered (2R1G)

3.2 Litmus Test With No Grace Period

It is important to note that RCU readers have absolutely no ordering properties except in conjunction with grace periods. This can be seen in Figure 8, which features two threads each with two RCU read-side critical sections. Because RCU readers have no ordering properties, both the compiler and the CPU are within their rights to reorder both P0's stores and P1's loads, so that the `exists` clause on line 13 can and does trigger.

Interestingly enough, because there are no grace periods, the translation to auxiliary-variable form simply comments out all the RCU read-side primitives. As a result, this is the only litmus test for which auxiliary-variable analysis runs as fast as does `herd7`'s native analysis.

3.3 Litmus Tests With Two Readers and One Grace Period

If RCU updaters excluded RCU readers in a manner similar to reader-writer locking, then any cycle including an updater and a pair of readers would be prohibited. However, this is not the case. RCU readers can run concurrently with RCU grace periods, the only restriction being that a given reader cannot completely overlap a given grace period. This is shown in Figure 9. Here P0's write to `x1` happens before P1's grace period (given `1:r1=1` in the `exists` clause) and P2's read from `r2` happens after the grace period (given `2:r2=1` in the `exists` clause). It is tempting to assume that P0's read from `x0` must therefore precede P2's write, however, both the compiler and the CPU can reorder both P0's and P2's accesses. Of course, the fundamental property of RCU prohibits P0's accesses (specifically its read from `x0`) from being reordered to follow the grace period, and similarly, P2's accesses (specifically its write to `x0`) cannot be reordered to precede the grace period. However, both process's `x0` accesses can be reordered to execute concurrently with the grace period, which is enough to allow P0's load to see P2's store, completing the cycle.

One way to think of this is that each RCU grace period imposes a delay, and that each RCU read-side critical section can be reordered to almost cancel out one grace period's worth of delay. This leads to a simple rule: If the number of grace periods in a given cycle is greater than or equal to the number of RCU read-side critical sections, that cycle is prohibited. This rule actually works in a number of situations and is explored further in Section 4.1.

However, this rule is not without exceptions. One exception occurs when a given process contains more than one of the RCU read-side critical sections in the cycle, as is the case in Figure 10. Because P1's read from `x1` saw P0's write, we know that P0's second RCU read-side critical section cannot be reordered to follow the grace period. However, it is also the case that no prior RCU read-side critical section may be reordered to follow the grace period, which means that P1's read from `x0` is guaranteed to see P0's write, so that the cycle is prohibited.[5] It is therefore not

[5] That said, auxiliary-variable analysis gets the right answer

```
 1 LISA LISA2Rt1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 }
 6  P0            | P1           ;
 7  f[lock]       | r[once] r1 x1 ;
 8  w[once] x0 1  | f[sync]       ;
 9  f[unlock]     | r[once] r2 x0 ;
10  f[lock]       |               ;
11  w[once] x1 1  |               ;
12  f[unlock]     |               ;
13 exists (1:r1=1 /\ 1:r2=0)
```

Figure 10: Grace Period Outnumbered, But Readers Share Process (2Rt1G Ø)

```
 1 LISA LISA2qR1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 }
 7  P0              | P1               | P2           ;
 8  f[lock]         | r[acquire] r1 x1 | f[lock]      ;
 9  r[once] r2 x0   | f[sync]          | r[once] r2 x2 ;
10  w[release] x1 1 | w[once] x2 1     | w[once] x0 1  ;
11  f[unlock]       |                  | f[unlock]     ;
12 exists (1:r1=1 /\ 2:r2=1 /\ 0:r2=1)
```

Figure 11: Grace Period Outnumbered, But Release and Acquire (2qR1G Ø)

sufficient to count RCU read-side critical sections.

But it turns out that the counting rule has exceptions even if each process has no more than one RCU read-side critical section. One example of this is shown in Figure 11, where P0's write-release synchronizes with P1's read-acquire. This release-acquire pairing ensures that all of P0's RCU read-side critical section precedes P1's grace period, in turn ensuring that P0's read from `x0` must precede P2's write, which must follow the beginning of the grace period. The cycle is thus prohibited, despite there being more RCU read-side critical sections than grace periods. Although one of the great strengths of RCU is that it interoperates well with other synchronization mechanisms, this interoperation can make it more difficult to correctly analyze RCU-related litmus tests.

Nor are release-acquire pairs the only way to prohibit cycles. Figure 12 shows that a memory barrier will serve just as well. This should be no surprise, given that a write-release can be emulated by placing a full memory barrier before the write, and that a read-acquire can be emulated by placing an RCU grace period after the read. One can easily argue that a grace period is overkill for this purpose, but it does work. In short, the litmus test in this figure should prohibit all cycles that are prohibited by that of Figure 11.

The cycle is also prohibited by a memory barrier in P2, as shown in Figure 13. In this case, the memory barrier acts by forcing all of P2's RCU read-side critical section to follow P1's grace period. Because RCU's fundamental guarantee prevents any part of P0's critical section from being ordered after the end of P1's grace period, P0's read from `x0` must preceded P2's write.

here by accident, given that it does not yet fully account for processes containing multiple RCU read-side critical sections.

```
 1 LISA LISA2Re1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 }
 7  P0            | P1           | P2           ;
 8  f[lock]       | r[once] r1 x1 | f[lock]      ;
 9  r[once] r2 x0 | f[sync]       | r[once] r2 x2 ;
10  f[mb]         | w[once] x2 1  | w[once] x0 1  ;
11  w[once] x1 1  |               | f[unlock]     ;
12  f[unlock]     |               |               ;
13 exists (1:r1=1 /\ 2:r2=1 /\ 0:r2=1)
```

Figure 12: Grace Period Outnumbered, But Memory Barrier I (2Re1G Ø)

```
 1 LISA LISA2Rf1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 }
 7  P0            | P1           | P2           ;
 8  f[lock]       | r[once] r1 x1 | f[lock]      ;
 9  r[once] r2 x0 | f[sync]       | r[once] r2 x2 ;
10  w[once] x1 1  | w[once] x2 1  | f[mb]         ;
11  f[unlock]     |               | w[once] x0 1  ;
12                |               | f[unlock]     ;
13 exists (1:r1=1 /\ 2:r2=1 /\ 0:r2=1)
```

Figure 13: Grace Period Outnumbered, But Memory Barrier II (2Rf1G Ø)

```
 1 LISA LISA2Rei1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 }
 7  P0            | P1            | P2            ;
 8  f[lock]       | r[once] r1 x1 | f[lock]       ;
 9  w[once] x1 1  | f[sync]       | r[once] r2 x2 ;
10  f[mb]         | w[once] x2 1  | w[once] x0 1  ;
11  r[once] r2 x0 |               | f[unlock]     ;
12  f[unlock]     |               |               ;
exists (1:r1=1 /\ 2:r2=1 /\ 0:r2=1)
```

Figure 14: Grace Period Outnumbered Despite Memory Barrier I (2Rei1G)

```
 1 LISA LISA2Rfi1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 }
 7  P0            | P1            | P2            ;
 8  f[lock]       | r[once] r1 x1 | f[lock]       ;
 9  r[once] r2 x0 | f[sync]       | w[once] x0 1  ;
10  w[once] x1 1  | w[once] x2 1  | f[mb]         ;
11  f[unlock]     |               | r[once] r2 x2 ;
12                |               | f[unlock]     ;
13 exists (1:r1=1 /\ 2:r2=1 /\ 0:r2=1)
```

Figure 15: Grace Period Outnumbered Despite Memory Barrier II (2Rfi1G)

At this point, it might be tempting to hypothesize that a modified counting analysis would work, whereadding either a release-acquire pair or a memory barrier to any of the RCU read-side critical sections excludes that RCU read-side critical section from the count. Unfortunately, the litmus test in Figure 14 thoroughly invalidates that hypothesis. Although P0 does contain a memory barrier, the fact that the order of P0's accesses to `x0` and `x1` have been reversed completely nullifies the effect of that memory barrier. P0's write to `x1` must precede P1's grace period, P0's write to `x0` is free to run concurrently with that grace period. The memory barrier therefore fails to prohibit the cycle.

Figure 15 shows that the same thing holds true for a memory barrier inserted into a reordered P2. It is not enough to have a memory barrier: The memory barrier must be properly positions amongst properly ordered accesses, otherwise the cycle will still be allowed.

3.4 Litmus Tests With Three Readers and One Grace Period

If cycles through one grace period and two read-side critical section are good, then cycles through one grace period and *three* read-side critical sections must be even better, and such a cycle is shown in Figure 16. Here, P0's write to `x2` and P2's write to `x3` can run concurrently with P1's grace period, but P3's write to `x0` can actually precede that grace period. The cycle is therefore not only allowed, but with up to almost two grace period's worth of time to spare.

As one might expect, adding a release-acquire pair does not necessarily prohibit the cycle, as can be seen in Figure 17. This release-acquire pair, which connects P1 and P2, does force P2's store to `x3` to follow P1's grace period, which in turn means that P3's read from `x3` also follows the grace period. However, both the compiler and the CPU can reorder P3's store to `x0` to run concurrently with P1's grace period. Therefore, despite the added release-acquire pair, the cycle is allowed.

One obvious thing to try is to make a release-acquire pair connect not consecutive processes in the cycle, but instead have the pair skip one of the processes. Figure 18 takes just this approach, with a release in P0 synchronizing with an acquire in P2. This release-acquire pair forces the whole of P0's RCU read-side critical section to precede P1's grace period. Unfortunately, it has little effect on the ordering of P2's critical section with respect to the grace period. After all, all that the release-acquire pair can do is to force P2's code to follow P0's store to `v0`, and that store could execute concurrently with P1's grace period. Therefore, all that the release-acquire pair has done for P2 is to force P2's code to execute after the start of P1's grace period, which RCU's fundamental guarantee was already doing. Therefore, the cycle is still allowed, despite the skip-process release-acquire pair.

It is clear that having a release-acquire pair skip the grace-period process isn't as helpful as one might

```
 1 LISA LISA3R1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 x3 = 0;
 7 }
 8  P0            | P1            | P2            | P3            ;
 9  f[lock]       | r[once] r1 x1 | f[lock]       | f[lock]       ;
10  r[once] r2 x0 | f[sync]       | r[once] r2 x2 | r[once] r3 x3 ;
11  w[once] x1 1  | w[once] x2 1  | w[once] x3 1  | w[once] x0 1  ;
12  f[unlock]     |               | f[unlock]     | f[unlock]     ;
13 exists (1:r1=1 /\ 2:r2=1 /\ 3:r3=1 /\ 0:r2=1)
```

Figure 16: Grace Period Triply Outnumbered (3R1G)

```
LISA LISA3R1Gq
{
x0 = 0;
x1 = 0;
x2 = 0;
x3 = 0;
}
 P0            | P1               | P2               | P3            ;
 f[lock]       | r[once] r1 x1    | f[lock]          | f[lock]       ;
 r[once] r2 x0 | f[sync]          | r[acquire] r2 x2 | r[once] r3 x3 ;
 w[once] x1 1  | w[release] x2 1  | w[once] x3 1     | w[once] x0 1  ;
 f[unlock]     |                  | f[unlock]        | f[unlock]     ;
exists (1:r1=1 /\ 2:r2=1 /\ 3:r3=1 /\ 0:r2=1)
```

Figure 17: Grace Period Triply Outnumbered, Despite Release and Acquire (3R1Gq)

```
 1 LISA LISA3qRq1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 x3 = 0;
 7 v0 = 0;
 8 }
 9  P0              | P1            | P2               | P3            ;
10  f[lock]         | r[once] r1 x1 | f[lock]          | f[lock]       ;
11  r[once] r2 x0   | f[sync]       | r[acquire] r4 v0 | r[once] r3 x3 ;
12  w[once] x1 1    | w[once] x2 1  | r[once] r2 x2    | w[once] x0 1  ;
13  w[release] v0 1 |               | w[once] x3 1     | f[unlock]     ;
14  f[unlock]       |               | f[unlock]        |               ;
15 exists (1:r1=1 /\ 2:r2=1 /\ 3:r3=1 /\ 0:r2=1 /\ 2:r4=1)
```

Figure 18: Grace Period Triply Outnumbered, Despite Skip-Process Release and Acquire (3qRq1G)

```
 1 LISA LISA3Rqq1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 x3 = 0;
 7 v0 = 0;
 8 }
 9  P0             | P1              | P2             | P3               ;
10  f[lock]        | r[once] r1 x1   | f[lock]        | f[lock]          ;
11  r[once] r2 x0  | f[sync]         | r[once] r2 x2  | r[acquire] r4 v0 ;
12  w[once] x1 1   | w[once] x2 1    | w[once] x3 1   | r[once] r3 x3    ;
13  f[unlock]      | w[release] v0 1 | f[unlock]      | w[once] x0 1     ;
14                 |                 |                | f[unlock]        ;
15 exists (1:r1=1 /\ 2:r2=1 /\ 3:r3=1 /\ 3:r4=1 /\ 0:r2=1)
```

Figure 19: Grace Period Triply Outnumbered, But Non-Grace-Period Skip-Process Release and Acquire (3Rqq1G ∅)

like. What if we instead skip one of the RCU readers? Figure 19 tries this out, and we finally have a litmus test with one grace period and three readers where the cycle is prohibited! P1's write-release of `v0` synchronizes with P2's read-acquire, which forces the entirety of P2's RCU read-side critical section to follow the end of P1's grace period. Because P0's load from `x0` cannot follow the end of the grace period, that load cannot return the value that P3 stores, thereby prohibiting the cycle.

For litmus tests containing one grace period and three RCU read-side critical sections, each in its own process, it appears that the rule is that a skip-process release-acquire pair is required to prohibit the cycle, and that even this is not always sufficient.

However, appearances can be deceiving, as can be seen in the litmus test shown in Figure 20. In this litmus test, a release-acquire pair connects the adjacent processes P2 and P3. P2's write-release, in combination with RCU's fundamental property, force the entirety of P2's RCU read-side critical section to follow P1's grace period. In addition, the fact that P2's write-release synchronizes with P3's read-acquire means that the entirety of P3's critical section also follows P1's grace period. Because RCU's fundamental guarantee also ensures that P0's read from `x0` precedes the end of P1's grace period, P0's read cannot return the value written by P3. This in turn means that the cycle is prohibited, despite the fact that there is only one release-acquire pair that connects consecutive processes in the cycle.

Then again, perhaps Figure 20 is merely the exception that proves the rule.

But if so, how do we explain Figure 21, which shows a different way to prohibit the cycle with but one release-acquire pair connecting consecutive processes? The determination as to whether or not a given cycle is prohibited clearly depends on fine details of everything making up that cycle.

3.5 Litmus Tests With Two Readers and Two Grace Periods

Figure 22 shows several grace periods having two readers and two grace periods each. The cycles in all three litmus tests are prohibited, as expected given that there are as many grace periods as there are readers. The cycles are prohibited when both grace periods are run by the same process (second test) and when both critical sections are run by the same process (third test).

In the third test, there are two independent cycles, one through P0's first critical section and P1, and the other through P0's second critical section and P2. The first version of the auxiliary-variable analysis would be expected to mis-analyze this litmus test, given that this analysis does not yet properly account for multiple critical sections in the same process. In reality, the analysis is far worse than that, as it hits a stack overflow.

```
 1 LISA LISA3Rq1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 x3 = 0;
 7 v0 = 0;
 8 }
 9  P0            | P1            | P2              | P3               ;
10  f[lock]       | r[once] r1 x1 | f[lock]         | f[lock]          ;
11  r[once] r2 x0 | f[sync]       | r[once] r2 x2   | r[acquire] r3 x3 ;
12  w[once] x1 1  | w[once] x2 1  | w[release] x3 1 | w[once] x0 1     ;
13  f[unlock]     |               | f[unlock]       | f[unlock]        ;
14 exists (1:r1=1 /\ 2:r2=1 /\ 3:r3=1 /\ 0:r2=1)
```

Figure 20: Grace Period Triply Outnumbered, But Release and Acquire I (3Rq1G Ø)

```
 1 LISA LISAq3R1G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 x3 = 0;
 7 }
 8  P0              | P1            | P2            | P3              ;
 9  f[lock]         | r[once] r1 x1 | f[lock]       | f[lock]         ;
10  r[acquire] r2 x0 | f[sync]       | r[once] r2 x2 | r[once] r3 x3   ;
11  w[once] x1 1     | w[once] x2 1  | w[once] x3 1  | w[release] x0 1 ;
12  f[unlock]       |               | f[unlock]     | f[unlock]       ;
13 exists (1:r1=1 /\ 2:r2=1 /\ 3:r3=1 /\ 0:r2=1)
```

Figure 21: Grace Period Triply Outnumbered, But Release and Acquire II (q3R1G Ø)

```
 1 LISA LISA2R2G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 x3 = 0;
 7 }
 8  P0            | P1            | P2            | P3            ;
 9  f[lock]       | r[once] r1 x1 | r[once] r1 x2 | f[lock]       ;
10  r[once] r2 x0 | f[sync]       | f[sync]       | r[once] r3 x3 ;
11  w[once] x1 1  | w[once] x2 1  | w[once] x3 1  | w[once] x0 1  ;
12  f[unlock]     |               |               | f[unlock]     ;
13 exists (1:r1=1 /\ 2:r2=1 /\ 3:r3=1 /\ 0:r2=1)
```

```
 1 LISA LISA2R2Gt
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 }
 7  P0            | P1            | P2            ;
 8  f[lock]       | r[once] r1 x1 | f[lock]       ;
 9  r[once] r2 x0 | f[sync]       | r[once] r2 x2 ;
10  w[once] x1 1  | f[sync]       | w[once] x0 1  ;
11  f[unlock]     | w[once] x2 1  | f[unlock]     ;
12 exists (1:r1=1 /\ 2:r2=1 /\ 0:r2=1)
```

```
 1 LISA LISA2Rt2G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 x3 = 0;
 7 }
 8  P0            | P1            | P2            ;
 9  f[lock]       | r[once] r1 x1 | r[once] r3 x3 ;
10  w[once] x0 1  | f[sync]       | f[sync]       ;
11  w[once] x1 1  | r[once] r2 x0 | r[once] r4 x2 ;
12  f[unlock]     |               |               ;
13  f[lock]       |               |               ;
14  w[once] x2 1  |               |               ;
15  w[once] x3 1  |               |               ;
16  f[unlock]     |               |               ;
17 exists (1:r1=0 /\ 1:r2=1 /\ 2:r3=0 /\ 2:r1=1)
```

Figure 22: Litmus Tests With Two Readers and Two Grace Periods Ø

```
 1 LISA LISA3R2G
 2 {
 3 x0 = 0;
 4 x1 = 0;
 5 x2 = 0;
 6 x3 = 0;
 7 }
 8  P0            | P1            | P2            | P3            ;
 9  f[lock]       | r[once] r1 x1 | f[lock]       | f[lock]       ;
10  r[once] r2 x0 | f[sync]       | r[once] r2 x2 | r[once] r3 x3 ;
11  w[once] x1 1  | f[sync]       | w[once] x3 1  | w[once] x0 1  ;
12  f[unlock]     | w[once] x2 1  | f[unlock]     | f[unlock]     ;
13 exists (1:r1=1 /\ 2:r2=1 /\ 3:r3=1 /\ 0:r2=1)
```

Figure 23: Litmus Tests With Three Readers and Two Grace Periods (3R2G)

3.6 Litmus Tests With Three Readers and Two Grace Periods

Figure 23 shows a litmus test with three readers and two grace periods, but with both grace periods on P1. Given that there are more readers than grace periods, and given that there is no other ordering, the cycle should be allowed. Unfortunately, neither analysis method handles this case correctly. The speed and accuracy of analysis is covered by the next section.

3.7 Solver Effectiveness

The `herd7` solver [1] correctly solves the canonical litmus test shown in Figure 5 in less than ten milliseconds and also correctly solves the auxiliary-variable form in about 30 milliseconds.

3.7.1 Modified Cat Grace Definition

Table 1 shows the results of a `linux.cat` file that replaces:

```
let grace = (sync;com+)+;(sandwich;com+)+
```

with:

```
let grace = (sandwich;com+);(sync;com+)+
```

As you can see from the table, this change corrects all errors in the initial cat file, but at the expense of introducing a similar set of alternative errors. This data supports the hypothesis that the RCU grace-period relationship cannot be directly represented by the cat-file language. The next section therefore evaluates an approach that introduces auxiliary variables, in effect, providing an implementation of RCU in the LISA language.

3.7.2 First Version Auxiliary Translation

Table 2 summarizes the results of running a number of RCU-related litmus tests. The "LISA" column shows the results of `linux.cat` analysis of the litmus tests, the "Hand" column shows the results for hand-coded auxiliary-variable analysis, and finally "Aux" shows the results of automatically translated auxiliary-variable analysis. The advantage of `linux.cat` analysis is blazing speed: None of the scenarios took more than 40 milliseconds. The corresponding penalty is mis-analysis for several of the

Litmus Test		LISA	Luc1	Luc2
1R1Gdeadlock	!			0.00
1R1G	Ø	0.00	0.00	0.00
1R1n1G	Ø	0.00	0.00	0.01
2qR1G	Ø	0.00	0.01	0.01
2R1G		0.00	0.90	0.00
2R2G	Ø	0.01	0.01	0.02
2R2Gt	Ø	0.00	0.01	0.01
2Re1G	Ø	0.00	0.00	0.00
2Rei1G		0.00	0.00	0.01
2Rf1G	Ø	0.01	0.01	0.00
2Rfi1G		0.00	0.01	0.00
2Rt1G	Ø	0.00	0.00	0.00
2Rt2G	Ø	0.02	0.02	0.02
3G	o			0.00
3qRq1G		0.03	0.05	0.04
3R1Gq		0.01	0.01	0.02
3R1G		0.01	0.01	0.01
3R2G		0.01	0.02	0.02
3Rq1G	Ø	0.02	0.01	0.01
3Rqq1G	Ø	0.04	0.02	0.04
4R2t		0.00	0.00	0.00
LISADL1	!			0.00
LISAnoDL0	!			0.00
LISAnoDL1	!			0.00
LISAnoDL2	!			0.00
q3R1G	Ø	0.02	0.01	0.02

Ø: Cycle prohibited
o: Cycle required
!: Deadlock
*.**: Test crashed
Red cell: Incorrect analysis or crash

Table 1: Modified Cat Grace Definition

Litmus Test		LISA	Hand	Aux
1R1G	Ø	0.00	0.03	0.03
1R1n1G	Ø	0.00		0.04
2qR1G	Ø	0.00		1.04
2R1G		0.00	0.91	1.02
2R2G	Ø	0.01	173.59	205.90
2R2Gt	Ø	0.00		98.48
2Re1G	Ø	0.00	0.99	1.11
2Rei1G		0.00	0.98	1.06
2Rf1G	Ø	0.01	0.96	1.11
2Rfi1G		0.00	0.97	1.09
2Rt1G	Ø	0.00		0.16
2Rt2G	Ø	0.02		*.**
3qRq1G		0.03		66.82
3R1G		0.01		29.04
3R1Gq		0.01		28.98
3Rq1G	Ø	0.02		29.62
3Rqq1G	Ø	0.04		64.73
3R2G		0.01		*.**
4R2t		0.00		0.00
q3R1G	Ø	0.02		29.09

Ø: Cycle prohibited
*.**: Test crashed
Red cell: Incorrect analysis or crash

Table 2: Modeling Results Summary

scenarios, though it can be argued that these mis-analyzed scenarios are rather unlikely to turn up in practice. The two auxiliary-variable columns boast much better accuracy of analysis, at least for those scenarios that successfully completed.[6] The corresponding penalty is a performance degradation ranging upwards of five orders of magnitude.

In addition, the initial version of the auxiliary-variable cannot handle backwards branches that cause a given RCU grace period or RCU read-side critical section to be executed twice.[7] It is likely that the native `linux.cat` analysis would not have a problem with this sort of backwards branch.

3.7.3 Second Version Auxiliary Translation

The second version of the auxiliary-variable translation script moves the prophesy-variable checks from LISA code to the `exists` clause, with results shown in Table 3. The results are decidedly mixed, however, it appears that the change benefits litmus tests with larger numbers of threads containing RCU read-side critical sections and hurts litmus tests with larger numbers of grace periods.

Interestingly enough, the hand-coded 2R2G litmus test does much better than the scripted version, which indicates that further optimization is eminently possible.

However, the 2Rt2G test with two read-side critical sections in one process and two grace periods each in their own process, which encountered a stack overflow on version one, completes successfully, albeit consuming more than two hours of CPU time and almost 3GB of memory. Furthermore, the 3R2G test with three RCU read-side critical sections and two grace periods, which also encountered a stack overflow on version one, also completes successfully, albeit consuming more than ten hours of CPU time and 2GB of memory. This gives some reason to believe that version two is overall better than is version

[6] 2Rt2G failed with a stack overflow after running for more than an hour and consuming well in excess of 4.5GB of memory. Given that its P0 contains 97 lines of non-comment code, this failure might not be all that surprising. 3R2G also failed with a stack overflow.

[7] This shortcoming is problematic for more general analysis tools such as `cbmc`.

Litmus Test		LISA	Hand	Aux	Chg
1R1G	Ø	0.00	0.04	0.04	-33.3%
1R1n1G	Ø	0.00		0.03	+25.0%
2qR1G	Ø	0.00		0.81	+22.1%
2R1G		0.00	0.91	0.80	+21.6%
2R2G	Ø	0.01	171.00	336.67	-63.5%
2R2Gt	Ø	0.00		155.04	-57.4%
2Re1G	Ø	0.00	1.00	0.87	+21.6%
2Rei1G		0.00	0.98	0.85	+19.8%
2Rf1G	Ø	0.01	0.99	0.85	+23.4%
2Rfi1G		0.00	0.98	0.86	+21.1%
2Rt1G	Ø	0.00		0.26	-62.3%
2Rt2G	Ø	0.01		8879.71	+∞%
3qRq1G		0.03		49.12	+26.5%
3R1G		0.01		20.64	+28.9%
3R1Gq		0.01		20.22	+30.2%
3Rq1G	Ø	0.02		21.27	+28.2%
3Rqq1G	Ø	0.03		47.98	+25.9%
3R2G		0.01		42686.62	+∞%
4R2t		0.00		0.00	0.0%
q3R1G	Ø	0.02		20.62	+29.1%

Ø: Cycle prohibited
*.**: Test crashed
Red cell: Incorrect analysis or crash

Table 3: Modeling Results Summary, Version Two

one. It also militates in favor of carefully splitting the work between not only the litmus-test code and the `exists` clause, but also between the `linux.cat` file, as was in fact suggested by Jade Alglave. This raises the question of exactly how to go about taking this apparently good advice, a topic that is taken up by the next section.

3.7.4 Third Version Auxiliary Translation

Slow though it was, the second version failed to handle litmus tests where a given thread might have more than one RCU read-side critical section, or, indeed, even one RCU read-side critical section with additional code outside of that critical section. Fixing this requires two writes to each ghost variable, which results in further slowdowns.

Fortunately, herd7 provides a "`-speedcheck true`" option that provides abbreviated checks. This abbreviated checking either (more) quickly determines if there are no executions, or (more) quickly locates a single execution that violates the `exists` clause.

Table 4 compares the two ways of running `herd7`.

3.7.5 Fourth Version Auxiliary Translation

It only makes sense to do two writes to the prophesy variables if there is at least one process that has two or more RCU read-side critical sections, or if there is at least one process that has one RCU read-side critical section with some code outside of that RCU read-side critical section. Therefore, for most litmus tests, only one write to the prophesy variable is required. Because decreasing the number of writes greatly decreases execution time, it makes sense to optimize for this common case. The results are shown in Table 5.

3.7.6 Solver Effectiveness Summary

An accurate analysis method with adequate performance is clearly much to be desired. However, one of the lessons of this section is that any accurate analysis method must take into account fine details of the litmus tests.

Litmus Test	Slow	Fast	Speedup
LISA1R1G	0.10	0.01	10.00
LISA1R1Gdeadlock	0	0	0.00
LISA1R1n1G	0.10	0.01	10.00
LISA1Rib1G	28.88	3.19	9.05
LISA1Rr1G	3.07	0.39	7.87
LISA1Rx1GM	5.59	0.63	8.87
LISA1xR1GM	1.26	0.14	9.00
LISA2qR1G	4.32	0.28	15.43
LISA2R1G	4.11	0.27	15.22
LISA2Re1G	4.69	0.28	16.75
LISA2Rec1G	10.90	0.90	12.11
LISA2Rei1G	4.58	0.29	15.79
LISA2Rf1G	4.57	0.28	16.32
LISA2Rfi1G	4.61	0.28	16.46
LISA2Rftx1GM	162.37	20.85	7.79
LISA2Rt1G	1.15	0.15	7.67
LISA2Rtx1GM	63.65	8.20	7.76
LISA2Rtx1GM-C	61.68	8.11	7.61
LISA2Rtxf1GM	331.88	47.40	7.00
LISA3G	0.00	0.00	0.00
LISA3qRq1G	374.01	19.70	18.99
LISA3R1G	152.57	8.91	17.12
LISA3R1Gq	150.20	8.91	16.86
LISA3Rq1G	156.00	9.08	17.18
LISA3Rqq1G	345.37	19.70	17.53
LISA4R2t	0.00	0.00	0.00
LISADL1	0.00	0.00	0.00
LISADL2ND	0.00	0.00	0.00
LISAnoDL0	0.00	0.00	0.00
LISAnoDL1	0.00	0.00	0.00
LISAnoDL2	0.00	0.00	0.00
LISAnoDL3-B	0.00	0.00	0.00
LISAq3R1G	154.58	8.97	17.23

Table 4: Modeling Results Summary, Version Three

Litmus Test	3 Fast	4 Fast	Speedup
LISA1R1G	0.01	0.01	1.00
LISA1R1Gdeadlock	0	0	0.00
LISA1R1n1G	0.01	0.01	1.00
LISA1Rib1G	3.19	3.13	1.02
LISA1Rr1G	0.39	0.36	1.08
LISA1Rx1GM	0.63	0.60	1.05
LISA1xR1GM	0.14	0.13	1.08
LISA2qR1G	0.28	0.11	2.55
LISA2R1G	0.27	0.11	2.45
LISA2Re1G	0.28	0.12	2.33
LISA2Rec1G	0.90	0.40	2.25
LISA2Rei1G	0.29	0.12	2.42
LISA2Rf1G	0.28	0.12	2.33
LISA2Rfi1G	0.28	0.12	2.33
LISA2Rftx1GM	20.85	19.38	1.08
LISA2Rt1G	0.15	0.14	1.07
LISA2Rtx1GM	8.20	7.73	1.06
LISA2Rtx1GM-C	8.11	7.72	1.05
LISA2Rtxf1GM	47.40	44.54	1.06
LISA3G	0.00	0.00	0.00
LISA3qRq1G	19.70	8.25	2.39
LISA3R1G	8.91	2.57	3.47
LISA3R1Gq	8.91	2.64	3.38
LISA3Rq1G	9.08	2.63	3.45
LISA3Rqq1G	19.70	5.79	3.40
LISA4R2t	0.00	0.00	0.00
LISADL1	0.00	0.00	0.00
LISADL2ND	0.00	0.00	0.00
LISAnoDL0	0.00	0.00	0.00
LISAnoDL1	0.00	0.00	0.00
LISAnoDL2	0.00	0.00	0.00
LISAnoDL3-B	0.00	0.00	0.00
LISAq3R1G	8.97	2.62	3.42

Table 5: Modeling Results Summary, Version Four

4 Bogus Optimizations

This section looks at a couple of attractive but ultimately bogus approaches to optimization.

4.1 RCU Grace-Period Relationship is Solved by Counting

Figure 1 is the base member of a class of litmus tests that compare the numbers of RCU read-side critical sections and RCU grace periods. The next member is shown in Figure 24. The cycle in this litmus test is allowed: Although Thread 0's read from `a` follows Thread 2's grace period and Thread 1's write to `c` precedes that same grace period, the two RCU read-side critical sections can overlap, allowing Thread 1's read from `b` to see Thread 1's write. This is illustrated by the figure beneath the litmus test, which shows how the statements in the RCU read-side critical sections may be reordered and how delays may be inserted to allow the cycle to occur. Of course, chaining additional RCU read-side critical sections would also allow the cycle.

However, if Thread 2 executes not one but two consecutive grace periods as shown in Figure 25, the cycle is prohibited. This may seem quite strange, as repeating most memory-barrier instructions in this manner would have no effect. However, due to the peculiar relationship between RCU read-side critical sections and RCU grace periods, repetition really does make a difference for RCU. This can be seen by applying the grace-period relationships:

1. Because `2:r1==1`, a portion of Thread 1's read-side critical section precedes Thread 2's first grace period. Therefore, all of Thread 1's critical section must precede the end of Thread 2's first grace period.

2. Similarly, because `0:r1==1`, a portion of Thread 0's read-side critical section follows Thread 2's second grace period. Therefore, all of Thread 0's critical section must follow the beginning of Thread 2's second grace period.

3. Combining these two ordering constraints, we see that all of Thread 1's RCU read-side critical section must precede that of Thread 0. Therefore, Thread 1's read from `b` cannot follow Thread 0's write, which means that we cannot have `1:r1==1`.

In short, adding the second `synchronize_rcu()` to Thread 2 prohibits the cycle. This situation underscores a major difference between RCU grace periods and memory-barrier instructions.

Similarly, if an additional grace period is chained in a separate thread while keeping a single reader, the cycle is prohibited, as shown in Figure 26. The diagram again illustrates the fact that the cycle cannot happen. Chaining additional grace periods would continue to prohibit the cycle.

As shown in Figure 27 having at least as many grace periods as readers prohibits the cycle. The graphic illustrates this, again inserting carefully chosen but entirely legal delays and reordering. Thread 1's RCU read-side critical section must end before Thread 2's grace period, which, combined with RCU's memory-barrier properties, means that Thread 0's RCU read-side critical section must begin before Thread 3's grace period. This in turn implies that Thread 0's critical section must end before the end of Thread 3's grace period, which ensures that Thread 0's read cannot return the value written by Thread 3's write, which prohibits the cycle. As before, chaining on more grace periods continues to prohibit the cycle. Generalizing, if there are at least as many grace periods as read-side critical sections in the cycle, the cycle will be prohibited.

However, this general rule assumes that the chain consists of a single run of RCU read-side critical sections followed by a single run of grace periods, where each pair of consecutive grace periods are partitioned by happens-before relationships, as has been the case in all the examples up to this point. In contrast, if we insert an RCU read-side critical section between the two grace periods in Figure 27, resulting in the situation shown in Figure 28, then the resulting cycle is allowed. The reason for this is that the interposed RCU read-side critical section in Thread 3 allows the two grace periods overlap, which in turn allows Thread 0's and Thread 1's RCU read-side critical sections to overlap, in turn allowing the cycle.

This can also be seen in the graphic representation.

Adding a second pair of grace periods split by an RCU read-side critical section, as shown in Figure 29, again prohibits the cycle, as can be seen in the illegible diagram. Adding another RCU read-side critical section between Thread 6 and Thread 7 would allow the cycle, which would again be prohibited by adding another grace period.

Thus, the relationship between RCU read-side critical sections and grace periods act like nested parentheses, in other words, the grammar of combinations is context free. My conjecture is that using the model shown in Figure 3 will allow a simple constraint system to nevertheless handle these litmus tests, at the cost of adding $O(N * M)$ variables and constraints, where N is the number of RCU read-side critical sections and M is the number of grace periods.

Alternatively, given a cycle of N RCU read-side critical sections and M grace periods, the cycle is allowed whenever $N \leq M$.

Unfortunately, there is no law against RCU read-side critical sections containing memory barriers or release-acquire relationships. Either case will defeat the simple counting rule.

Furthermore, a normal constraint solver need only find the first cycle, then stop. A counting constraint solver must check each cycle it finds, and must ignore irrelevant cycles. This in turn means that further processing must avoid finding old irrelevant cycles. In theory, this is just a simple matter of software, but in practice this greatly increases the computational complexity.

It is quite possible that further thought will uncover a way to nevertheless make use of this counting rule, but this is not currently at all obvious. In the meantime, it is a handy way to manually check whether or not a model is doing the right thing for a large RCU litmus test.

4.2 Ignoring Irrelevant Interaction

It is tempting to assert that a given RCU read-side critical section need only interact with threads containing grace periods that access variables that are also accessed by that RCU read-side critical section. This would be very convenient, as it would greatly reduce the number of prophesy variables and memory-barrier checks. Unfortunately, this simply does not work in all cases. To see this, consider Figure 29. Here Thread 6's RCU read-side critical section must interact with Thread 4's grace period, despite there being no accesses in common.

5 Opportunities for Optimization

The current auxiliary-variable translation generates branches over branches due to the fact that there is only an equality comparison. An inequality comparison would eliminate these extra instructions. Although it would be possible to reverse the sense of the prophesy variables, that would simply move the branch-over-branch from the prophesy check to the memory-barrier check. Another possibility would be to move the prophesy check to the event condition, but this would require an additional pair of register for each prophesy check in each thread, which is not likely to be an improvement. A not-equals comparison would therefore be helpful.

There are cases where the code knows that the current execution is useless, but there appear to be no way to inform the constraint solver of this fact. If there was a way to inform the constraint solver of doomed executions, this could be used to check the prophesy and potentially to check for violations of the grace-period property.[8]

All the members of a given set of grace-period checks could use the same memory-barrier instruction, as is done in the hand-generated tests. If this is thought to produce significant savings, a future version of the translation tool will take this approach.

It might be possible to economize on the check code. One possibility is to place it only at the beginning and end of each RCU read-side critical section. In cases where there is code preceding the first or following the last RCU read-side critical section, it might also be necessary to place checks at the beginning or end of the process. That said, this won't help litmus

[8] No RCU read-side critical section is permitted to span a grace period.

tests in which each RCU read-side critical section completely fills its process.

Another approach would be to generate multiple auxiliary-variable litmus tests, each with a different relationship of grace periods and threads, then run each of them separately. If any showed that a cycle was allowed, then the overall test would need to be considered to allow that cycle. Instead of one heavyweight test, large litmus tests would generate a very large number of much smaller tests.

References

[1] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 40–40, New York, NY, USA, 2014. ACM.

[2] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, pages 297–310. USENIX Association, June 2003.

[3] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23:375–382, 2012.

[4] Paul E. McKenney. Structured deferral: synchronization via procrastination. *Commun. ACM*, 56(7):40–49, July 2013.

Thread 0	Thread 1	Thread 2
`rcu_read_lock();`	`rcu_read_lock();`	`r1 = READ_ONCE(c);`
`r1 = READ_ONCE(a);`	`r1 = READ_ONCE(b);`	`synchronize_rcu();`
`WRITE_ONCE(b, 1);`	`WRITE_ONCE(c, 1);`	`WRITE_ONCE(a, 1);`
`rcu_read_unlock();`	`rcu_read_unlock();`	

`BUG_ON(0:r1 == 1 && 1:r1 == 1 && 2:r1 == 1);`
(Cycle allowed)

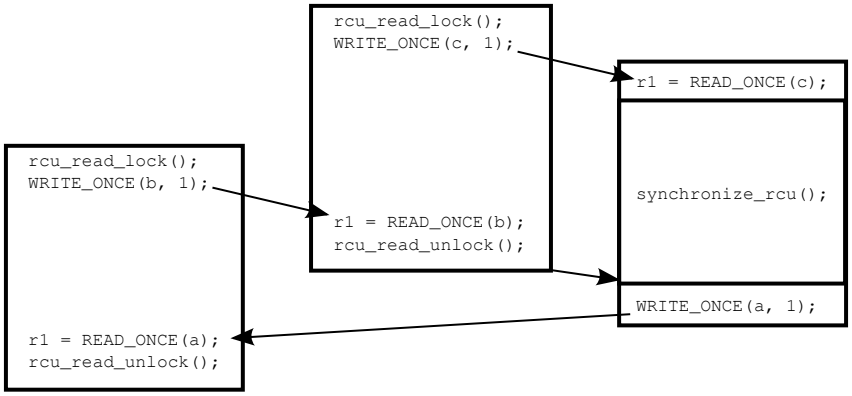

Figure 24: Two Readers, One Grace Period

Thread 0	Thread 1	Thread 2
rcu_read_lock();	rcu_read_lock();	r1 = READ_ONCE(c);
r1 = READ_ONCE(a);	r1 = READ_ONCE(b);	synchronize_rcu();
WRITE_ONCE(b, 1);	WRITE_ONCE(c, 1);	synchronize_rcu();
rcu_read_unlock();	rcu_read_unlock();	WRITE_ONCE(a, 1);

BUG_ON(0:r1 == 1 && 1:r1 == 1 && 2:r1 == 1);
(Cycle prohibited)

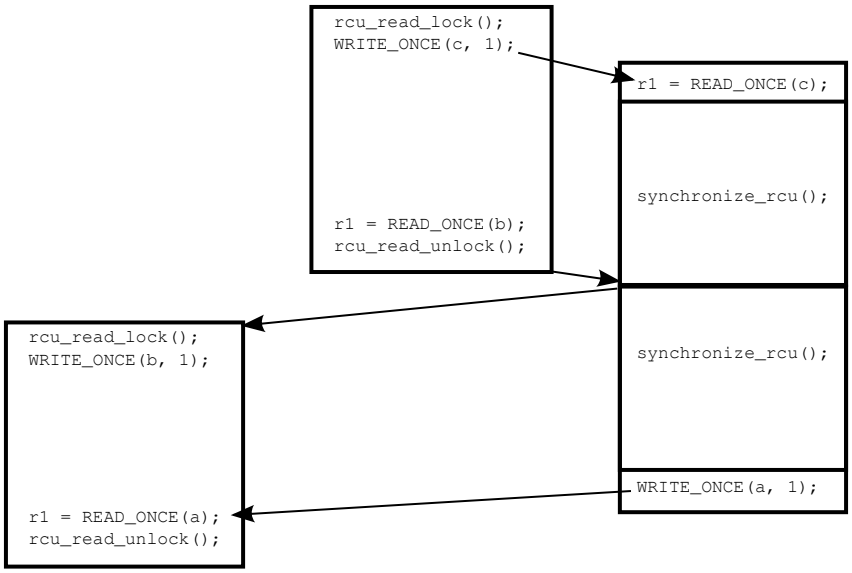

Figure 25: Two Readers, Two Consecutive Grace Periods

Thread 0	Thread 1	Thread 2
`rcu_read_lock();`	`r1 = READ_ONCE(b);`	`r1 = READ_ONCE(c);`
`r1 = READ_ONCE(a);`	`synchronize_rcu();`	`synchronize_rcu();`
`WRITE_ONCE(b, 1);`	`WRITE_ONCE(c, 1);`	`WRITE_ONCE(a, 1);`
`rcu_read_unlock();`		

`BUG_ON(0:r1 == 1 && 1:r1 == 1 && 2:r1 == 1);`
(Cycle prohibited)

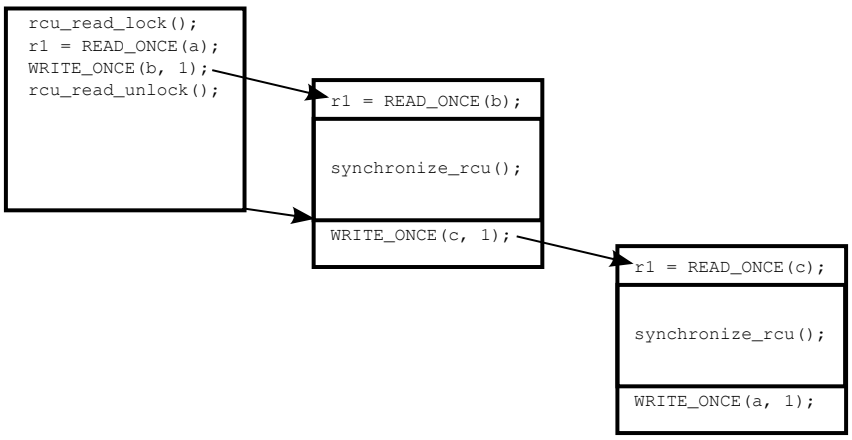

Figure 26: One Reader, Two Grace Periods

Thread 0	Thread 1	Thread 2	Thread 3
`rcu_read_lock();`	`rcu_read_lock();`	`r1 = READ_ONCE(c);`	`r1 = READ_ONCE(d);`
`r1 = READ_ONCE(a);`	`r1 = READ_ONCE(b);`	`synchronize_rcu();`	`synchronize_rcu();`
`WRITE_ONCE(b, 1);`	`WRITE_ONCE(c, 1);`	`WRITE_ONCE(d, 1);`	`WRITE_ONCE(a, 1);`
`rcu_read_unlock();`	`rcu_read_unlock();`		

`BUG_ON(0:r1 == 1 && 1:r1 == 1 && 2:r1 == 1 && 3:r1 == 1);`
(Cycle prohibited)

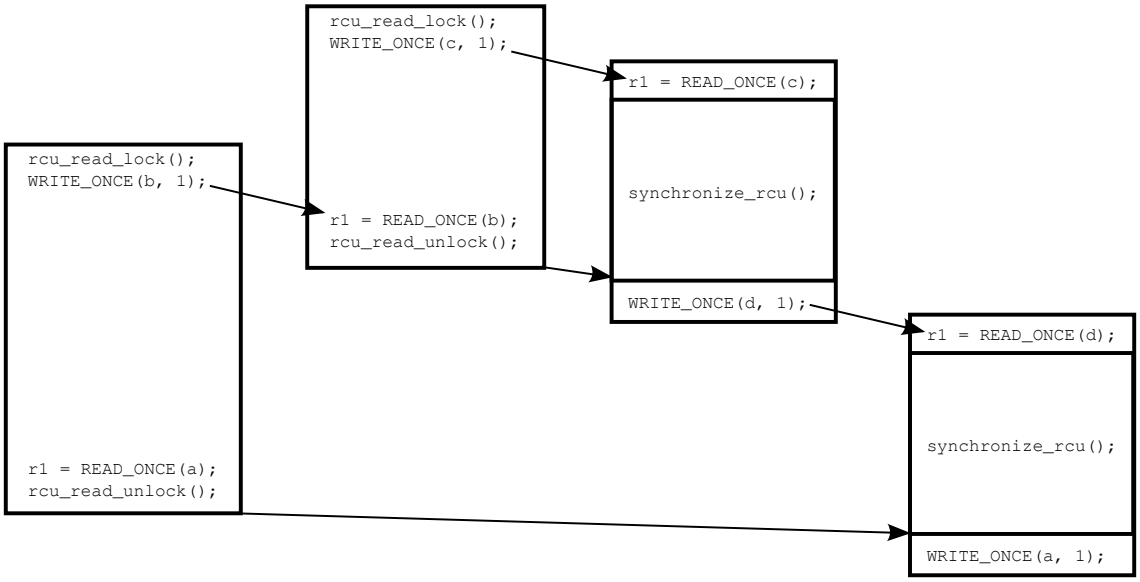

Figure 27: Two Readers, Two Grace Periods

Thread 0	Thread 1	Thread 2	Thread 3
`rcu_read_lock();`	`rcu_read_lock();`	`r1 = READ_ONCE(c);`	`rcu_read_lock();`
`r1 = READ_ONCE(a);`	`r1 = READ_ONCE(b);`	`synchronize_rcu();`	`r1 = READ_ONCE(d);`
`WRITE_ONCE(b, 1);`	`WRITE_ONCE(c, 1);`	`WRITE_ONCE(d, 1);`	`WRITE_ONCE(e, 1);`
`rcu_read_unlock();`	`rcu_read_unlock();`		`rcu_read_unlock();`

Thread 4			
`r1 = READ_ONCE(e);`			
`synchronize_rcu();`			
`WRITE_ONCE(f, 1);`			

`BUG_ON(0:r1 == 1 && 1:r1 == 1 && 2:r1 == 1 && 3:r1 == 1 && 4:r1 == 1);`
(Cycle allowed)

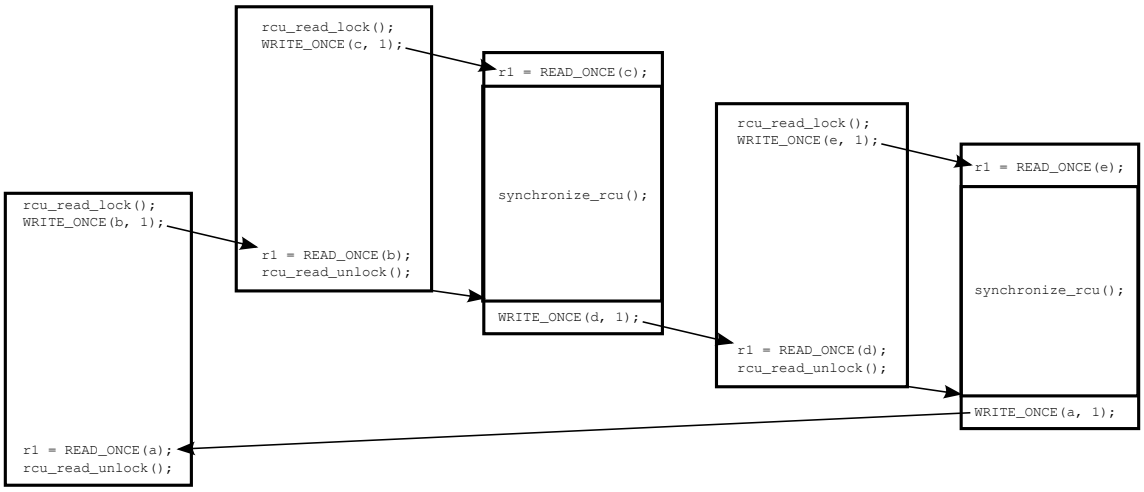

Figure 28: Two Readers, One Composite Grace Period

Thread 0	Thread 1	Thread 2	Thread 3
`rcu_read_lock();`	`rcu_read_lock();`	`r1 = READ_ONCE(c);`	`rcu_read_lock();`
`r1 = READ_ONCE(a);`	`r1 = READ_ONCE(b);`	`synchronize_rcu();`	`r1 = READ_ONCE(d);`
`WRITE_ONCE(b, 1);`	`WRITE_ONCE(c, 1);`	`WRITE_ONCE(d, 1);`	`WRITE_ONCE(e, 1);`
`rcu_read_unlock();`	`rcu_read_unlock();`		`rcu_read_unlock();`

Thread 4	Thread 5	Thread 6	Thread 7
`r1 = READ_ONCE(e);`	`r1 = READ_ONCE(f);`	`rcu_read_lock();`	`r1 = READ_ONCE(h);`
`synchronize_rcu();`	`synchronize_rcu();`	`r1 = READ_ONCE(g);`	`synchronize_rcu();`
`WRITE_ONCE(f, 1);`	`WRITE_ONCE(g, 1);`	`WRITE_ONCE(h, 1);`	`WRITE_ONCE(a, 1);`
		`rcu_read_unlock();`	

```
BUG_ON(0:r1 == 1 && 1:r1 == 1 && 2:r1 == 1 && 3:r1 == 1 &&
       4:r1 == 1 && 5:r1 == 1 && 6:r1 == 1 && 7:r1 == 1);
```

(Cycle prohibited)

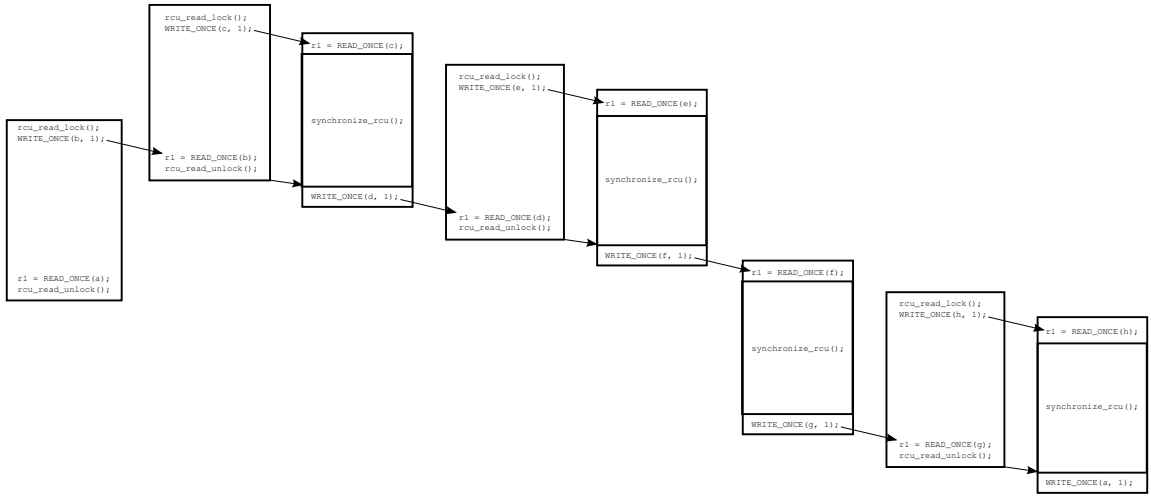

Figure 29: Two Readers, Two Composite Grace Periods