What is RCU? Part 2: Usage

Introduction

December 24, 2007
This article was contributed by Paul McKenney

Read-copy update (RCU) is a synchronization mechanism that was added to the Linux kernel in October of 2002. RCU is most frequently described as a replacement for reader-writer locking, but it has also been used in a number of other ways. RCU is notable in that RCU readers do not directly synchronize with RCU updaters, which makes RCU read paths extremely fast, and also permits RCU readers to accomplish useful work even when running concurrently with RCU updaters.

This leads to the question "what exactly is RCU?", a question that this document addresses from the viewpoint of someone using it. Because RCU is most frequently used to replace some existing mechanism, we look at it primarily in terms of its relationship to such mechanisms, as follows:

These sections are followed by conclusions and answers to the Quick Quizzes.

RCU is a Reader-Writer Lock Replacement

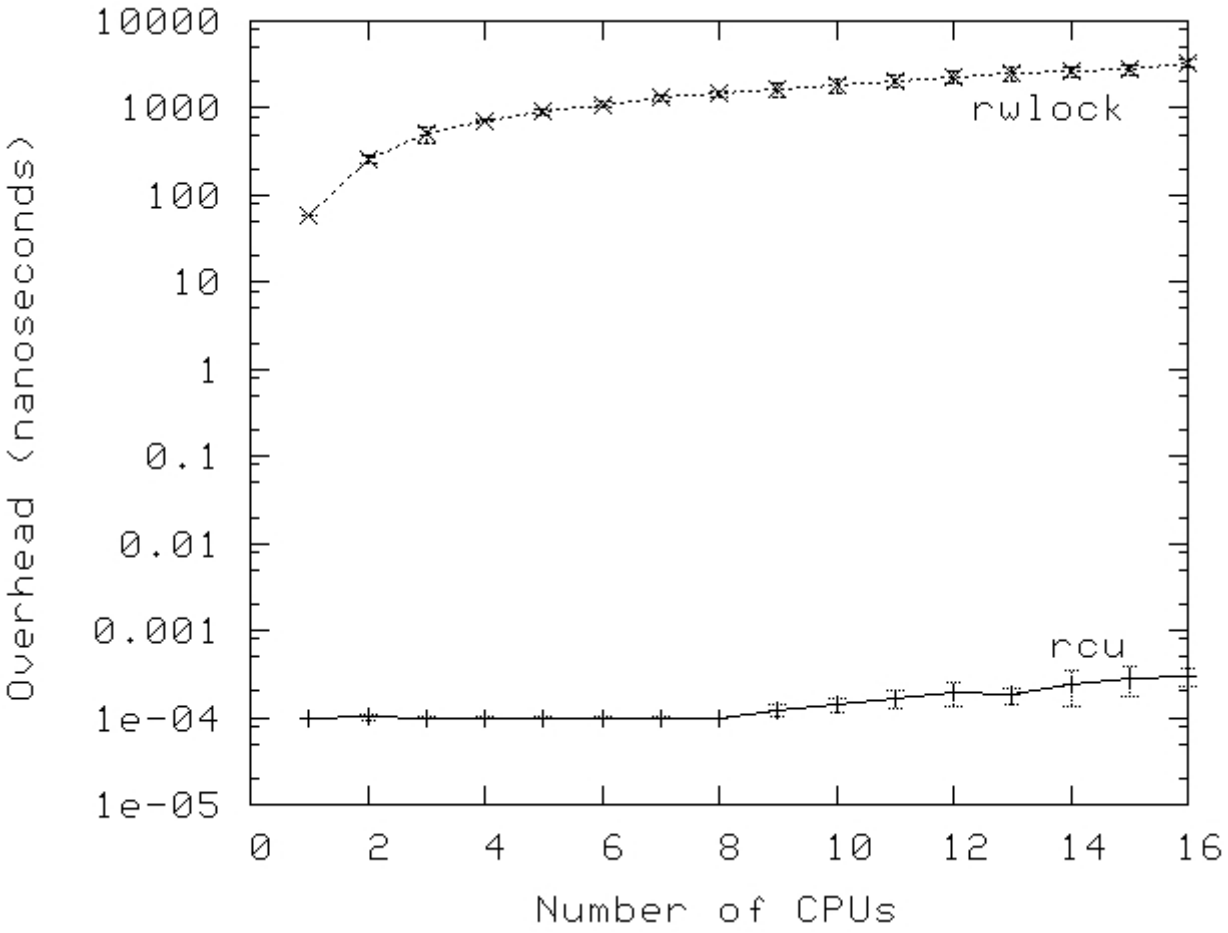
Perhaps the most common use of RCU within the Linux kernel is as a replacement for reader-writer locking in read-intensive situations. Nevertheless, this use of RCU was not immediately apparent to me at the outset, in fact, I chose to implement something similar to `brlock`before implementing a general-purpose RCU implementation back in the early 1990s. Each and every one of the uses I envisioned for the proto-brlock primitive was instead implemented using RCU. In fact, it was more than three years before the proto-`brlock` primitive saw its first use. Boy, did I feel foolish!

The key similarity between RCU and reader-writer locking is that both have read-side critical sections that can execute in parallel. In fact, in some cases, it is possible to mechanically substitute RCU API members for the corresponding reader-writer lock API members. But first, why bother?

Advantages of RCU include performance, deadlock immunity, and realtime latency. There are, of course, limitations to RCU, including the fact that readers and updaters run concurrently, that low-priority RCU readers can block high-priority threads waiting for a grace period to elapse, and that grace-period latencies can extend for many milliseconds. These advantages and limitations are discussed in the following sections.

Performance

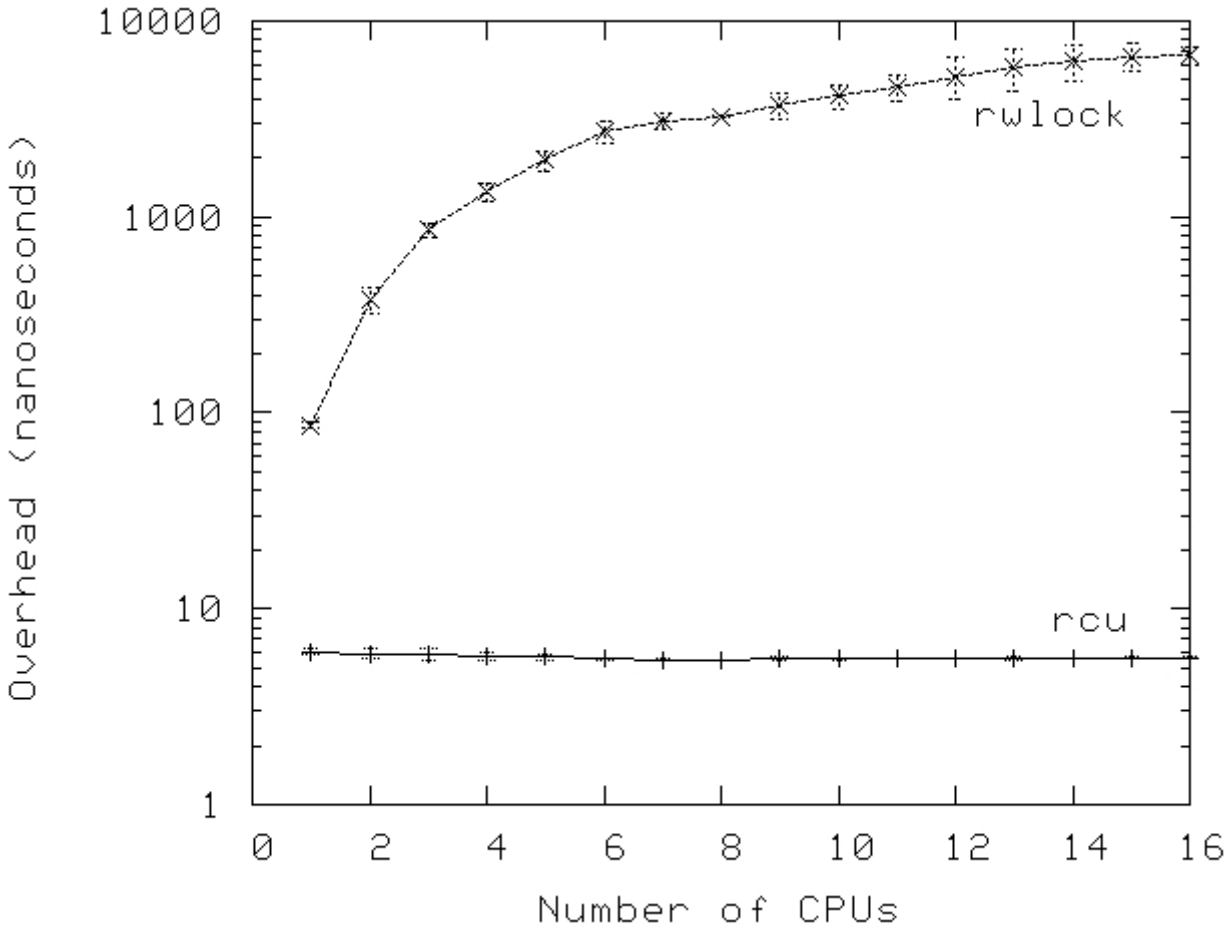
The read-side performance advantages of RCU over reader-writer lock are shown on the following graph for a 16-CPU 3GHz Intel x86 system.



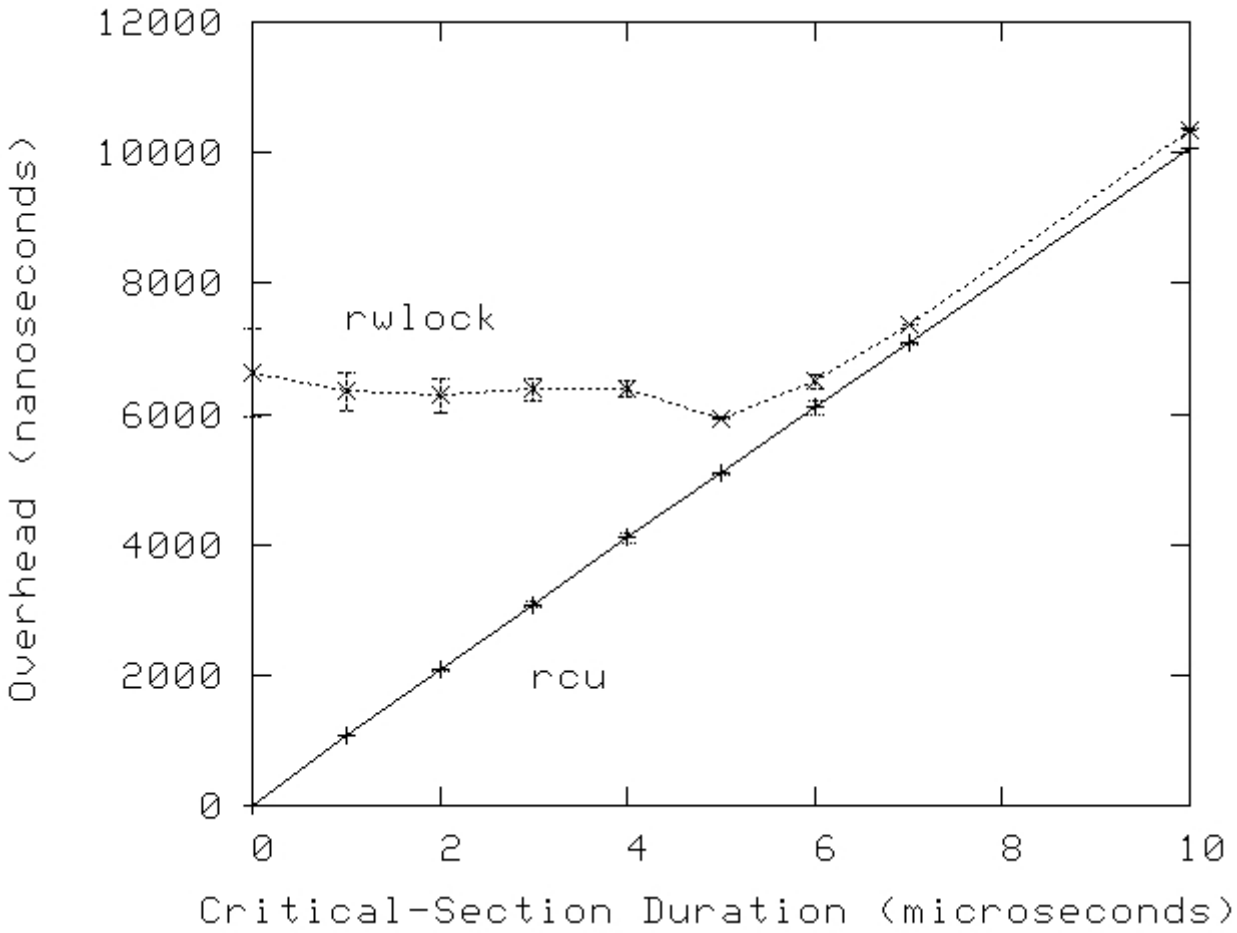
Note that reader-writer locking is orders of magnitude slower than RCU on a single CPU, and is almost two *additional* orders of magnitude slower on 16 CPUs. In contrast, RCU scales quite well. In both cases, the error bars span a single standard deviation in either direction.

Quick Quiz 1: WTF??? How the heck do you expect me to believe that RCU has a 100-femtosecond overhead when the clock period at 3GHz is more than 300 *picoseconds*?

A more moderate view may be obtained from a `CONFIG_PREEMPT` kernel, though RCU still beats reader-writer locking by between one and three orders of magnitude. Note the high variability of reader-writer locking at larger numbers of CPUs. The error bars span a single standard deviation in either direction.



Of course, the low performance of reader-writer locking will become less significant as the overhead of the critical section increases, as shown in the following graph for a 16-CPU system.



However, this observation must be tempered by the fact that a number of system calls (and thus any RCU read-side critical sections that they contain) can complete within a few microseconds. In addition, as is discussed in the next section, RCU read-side primitives are almost entirely deadlock-immune.

Quick Quiz 2: Why does both the variability and overhead of rwlock decrease as the critical-section overhead increases?

Deadlock Immunity

Although RCU offers significant performance advantages for read-mostly workloads, one of the primary reasons for creating RCU in the first place was in fact its immunity to read-side deadlocks. This immunity stems from the fact that RCU read-side primitives do not block, spin, or even do backwards branches, so that their execution time is deterministic. It is therefore impossible for them to participate in a deadlock cycle.

An interesting consequence of RCU's read-side deadlock immunity is that it is possible to unconditionally upgrade an RCU reader to an RCU updater. Attempting to do such an upgrade with reader-writer locking results in deadlock. A sample code fragment that does an RCU read-to-update upgrade follows:

Quick Quiz 3: Is there an exception to this deadlock immunity, and if so, what sequence of events could lead to deadlock?

```
 1 rcu_read_lock();
 2 list_for_each_entry_rcu(p, head, list_field) {
 3   do_something_with(p);
 4   if (need_update(p)) {
 5     spin_lock(&my_lock);
 6     do_update(p);
 7     spin_unlock(&my_lock);
 8   }
 9 }
10 rcu_read_unlock();
```

Note that `do_update()` is executed under the protection of the lock *and* under RCU read-side protection.

Another interesting consequence of RCU's deadlock immunity is its immunity to a large class of priority inversion problems. For example, low-priority RCU readers cannot prevent a high-priority RCU updater from acquiring the update-side lock. Similarly, a low-priority RCU updater cannot prevent high-priority RCU readers from entering an RCU read-side critical section.

Realtime Latency

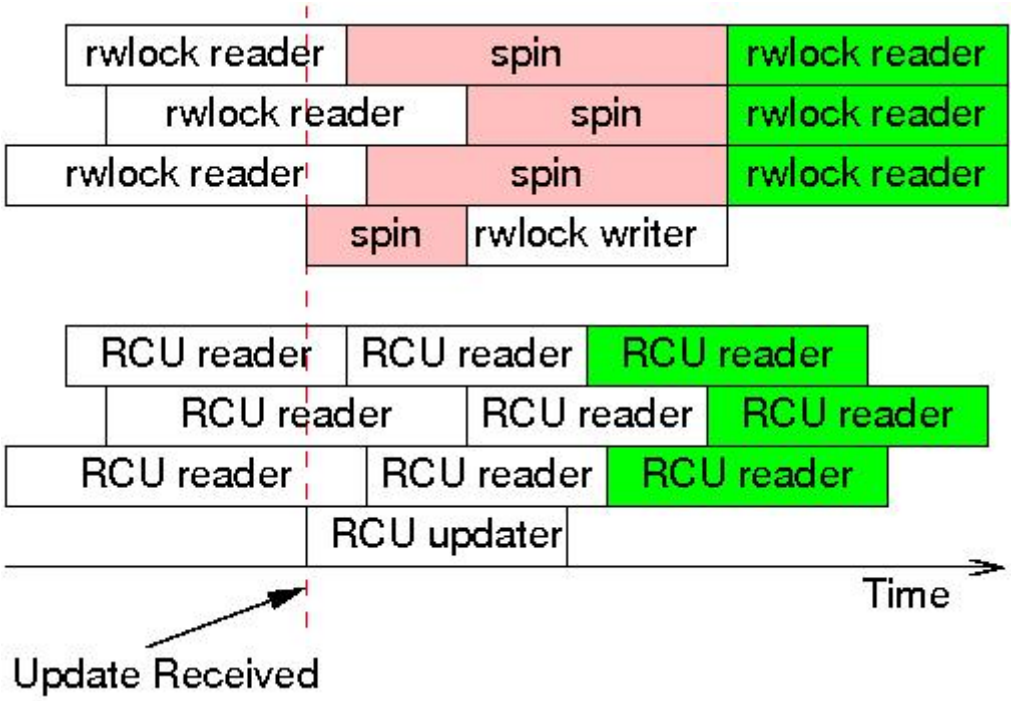
Because RCU read-side primitives neither spin nor block, they offer excellent realtime latencies. In addition, as noted earlier, this means that they are immune to priority inversion involving the RCU read-side primitives and locks.

However, RCU is susceptible to more subtle priority-inversion scenarios, for example, a high-priority process blocked waiting for an RCU grace period to elapse can be blocked by low-priority RCU readers in -rt kernels. This can be solved by using RCU priority boosting.

RCU Readers and Updaters Run Concurrently

Because RCU readers never spin nor block, and because updaters are not subject to any sort of rollback or abort semantics, RCU readers and updaters must necessarily run concurrently. This means that RCU readers might access stale data, and might even see inconsistencies, either of which can render conversion from reader-writer locking to RCU non-trivial.

However, in a surprisingly large number of situations, inconsistencies and stale data are not problems. The classic example is the networking routing table. Because routing updates can take considerable time to reach a given system (seconds or even minutes), the system will have been sending packets the wrong way for quite some time when the update arrives. It is usually not a problem to continue sending updates the wrong way for a few additional milliseconds. Furthermore, because RCU updaters can make changes without waiting for RCU readers to finish, the RCU readers might well see the change more quickly than would batch-fair reader-writer-locking readers, as shown in the following figure.



Once the update is received, the rwlock writer cannot proceed until the last reader completes, and subsequent readers cannot proceed until the writer completes. However, these subsequent readers are guaranteed to see the new value, as indicated by the green background. In contrast, RCU readers and updaters do not block each other, which permits the RCU readers to see the updated values sooner. Of course, because their execution overlaps that of the RCU updater, *all* of the RCU readers might well see updated values, including the three readers that started before the update. Nevertheless only the RCU readers with green backgrounds are *guaranteed* to see the updated values, again, as indicated by the green background.

Reader-writer locking and RCU simply provide different guarantees. With reader-writer locking, any reader that begins after the writer starts executing is guaranteed to see new values, and readers that attempt to start while the writer is spinning might or might not see new values, depending on the reader/writer preference of the rwlock implementation in question. In contrast, with RCU, any reader that begins after the updater completes is guaranteed to see new values, and readers that end after the updater begins might or might not see new values, depending on timing.

The key point here is that, although reader-writer locking does indeed guarantee consistency within the confines of the computer system, there are situations where this consistency comes at the price of increased *inconsistency* with the outside world. In other words, reader-writer locking obtains internal consistency at the price of silently stale data with respect to the outside world.

Nevertheless, there are situations where inconsistency and stale data within the confines of the system cannot be tolerated. Fortunately, there are a number of approaches that avoid inconsistency and stale data, as discussed in the FREENIX paper on applying RCU to System V IPC [PDF] and in my dissertation [PDF]. However, an in-depth discussion of these approaches is beyond the scope of this article.

Low-Priority RCU Readers Can Block High-Priority Reclaimers

In Realtime RCU, SRCU, or QRCU, each of which is described in the final installment of this series, a preempted reader will prevent a grace period from completing, even if a high-priority task is blocked waiting for that grace period to complete. Realtime RCU can avoid this problem by substituting `call_rcu()` for `synchronize_rcu()` or by using RCU priority boosting, which is still in experimental status as of late 2007. It might become necessary to augment SRCU and QRCU with priority boosting, but not before a clear real-world need is demonstrated.

RCU Grace Periods Extend for Many Milliseconds

With the exception of QRCU, RCU grace periods extend for multiple milliseconds. Although there are a number of techniques to render such long delays harmless, including use of the asynchronous interfaces where available (`call_rcu()` and `call_rcu_bh()`), this situation is a major reason for the rule of thumb that RCU be used in read-mostly situations.

Comparison of Reader-Writer Locking and RCU Code

In the best case, the conversion from reader-writer locking to RCU is quite simple, as shown in the following example taken from Wikipedia.

```
 1 struct el {
 2   struct list_head list;
 3   long key;
 4   spinlock_t mutex;
 5   int data;
 6   /* Other data fields */
 7 };
 8 rwlock_t listmutex;
 9 struct el head;

 1 int search(long key, int *result)
 2 {
 3   struct list_head *lp;
 4   struct el *p;
 5
 6   read_lock(&listmutex);
 7   list_for_each_entry(p, head, lp) {
 8     if (p->key == key) {
 9       *result = p->data;
10       read_unlock(&listmutex);
11       return 1;
12     }
13   }
14   read_unlock(&listmutex);
15   return 0;
16 }

 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   write_lock(&listmutex);
 6   list_for_each_entry(p, head, lp) {
 7     if (p->key == key) {
 8       list_del(&p->list);
 9       write_unlock(&listmutex);
10       kfree(p);
11       return 1;
12     }
13   }
14   write_unlock(&listmutex);
```

```
 1 struct el {
 2   struct list_head list;
 3   long key;
 4   spinlock_t mutex;
 5   int data;
 6   /* Other data fields */
 7 };
 8 spinlock_t listmutex;
 9 struct el head;

 1 int search(long key, int *result)
 2 {
 3   struct list_head *lp;
 4   struct el *p;
 5
 6   rcu_read_lock();
 7   list_for_each_entry_rcu(p, head, lp) {
 8     if (p->key == key) {
 9       *result = p->data;
10       rcu_read_unlock();
11       return 1;
12     }
13   }
14   rcu_read_unlock();
15   return 0;
16 }

 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   spin_lock(&listmutex);
 6   list_for_each_entry(p, head, lp) {
 7     if (p->key == key) {
 8       list_del_rcu(&p->list);
 9       spin_unlock(&listmutex);
10       synchronize_rcu();
11       kfree(p);
12       return 1;
13     }
14   }
15   spin_unlock(&listmutex);
```

```
15   return 0;                          16   return 0;
16 }                                    17 }
```

More-elaborate cases of replacing reader-writer locking with RCU are beyond the scope of this document.

RCU is a Restricted Reference-Counting Mechanism

Because grace periods are not allowed to complete while there is an RCU read-side critical section in progress, the RCU read-side primitives may be used as a restricted reference-counting mechanism. For example, consider the following code fragment:

```
1 rcu_read_lock();  /* acquire reference. */
2 p = rcu_dereference(head);
3 /* do something with p. */
4 rcu_read_unlock();  /* release reference. */
```

The `rcu_read_lock()` primitive can be thought of as acquiring a reference to `p`, because a grace period starting after the `rcu_dereference()` assigns to `p` cannot possibly end until after we reach the matching `rcu_read_unlock()`. This reference-counting scheme is restricted in that we are not allowed to block in RCU read-side critical sections, nor are we permitted to hand off an RCU read-side critical section from one task to another.

Regardless of these restrictions, the following code can safely delete `p`:

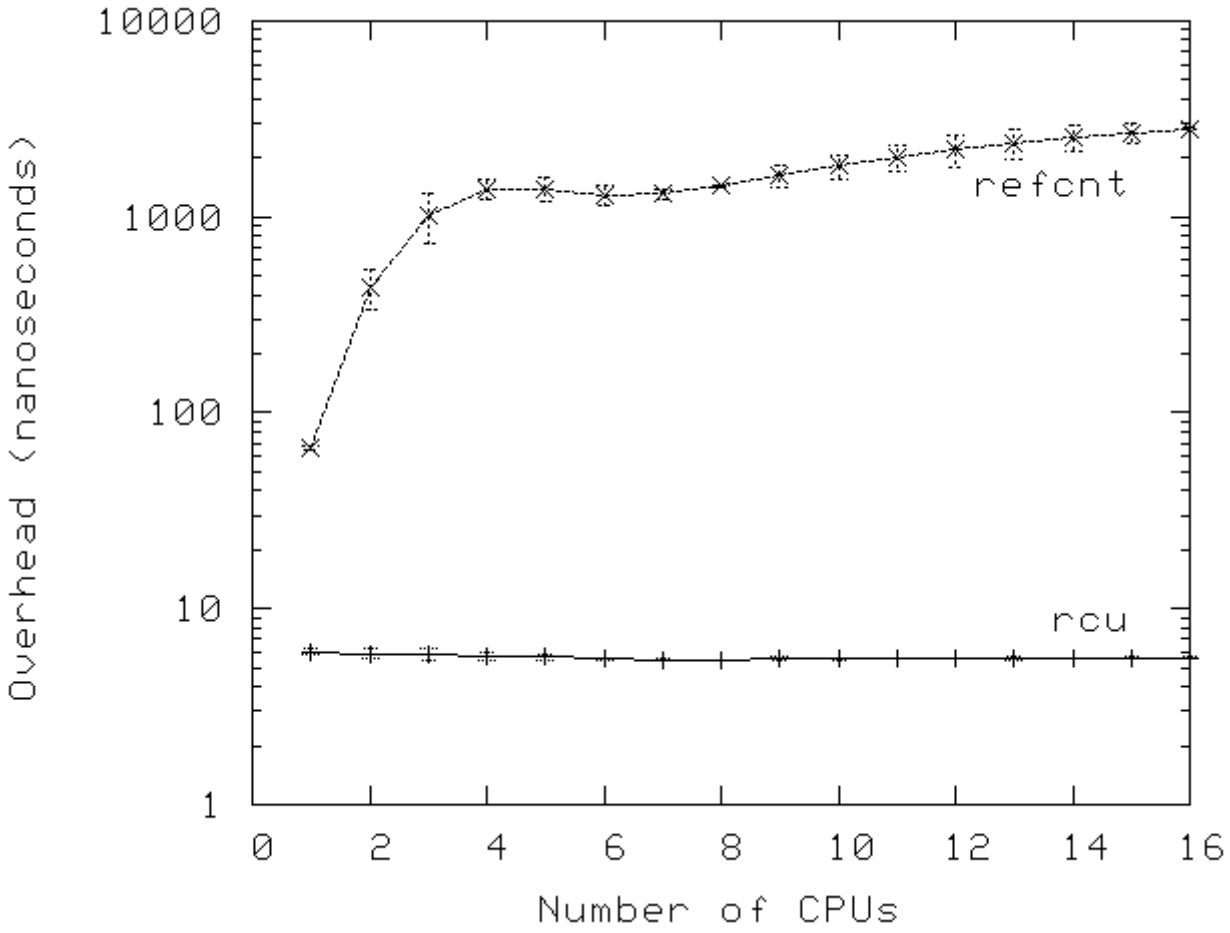
```
1 spin_lock(&mylock);
2 p = head;
3 head = NULL;
4 spin_unlock(&mylock);
5 synchronize_rcu();  /* Wait for all references to be released. */
6 kfree(p);
```

The assignment to `head` prevents any future references to `p` from being acquired, and the `synchronize_rcu()` waits for any references that had previously been acquired to be released.

Of course, RCU can also be combined with traditional reference counting, as has been discussed on LKML and as summarized in anOverview of Linux-Kernel Reference Counting [PDF].

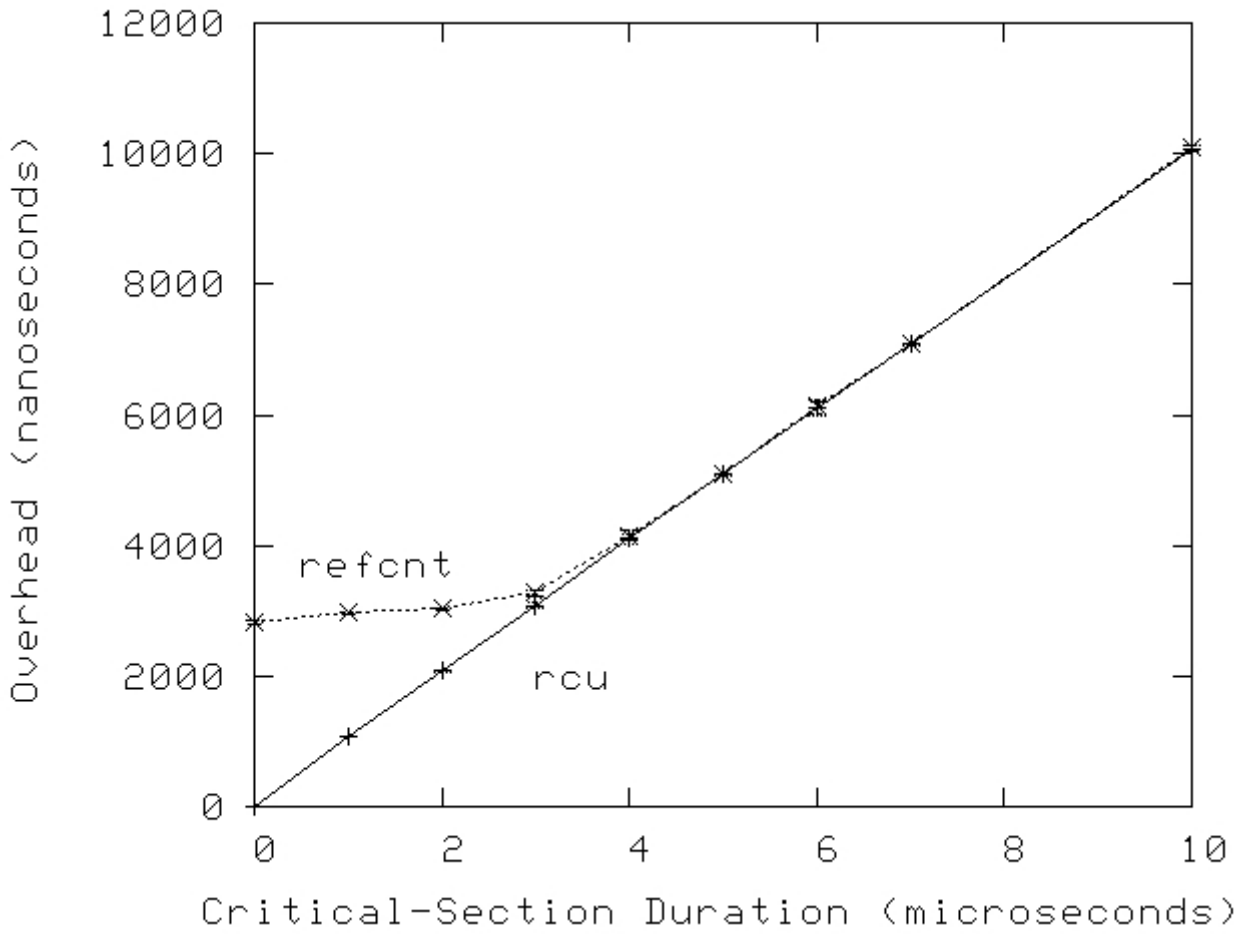
But why bother? Again, part of the answer is performance, as shown in the following graph, again showing data taken on a 16-CPU 3GHz Intel x86 system.

Quick Quiz 4: But wait! This is exactly the same code that might be used when thinking of RCU as a replacement for reader-writer locking! What gives?



And, as with reader-writer locking, the performance advantages of RCU are most pronounced for short-duration critical sections, as shown in the following graph for a 16-CPU system. In addition, as with reader-writer locking, many system calls (and thus any RCU read-side critical sections that they contain) complete in a few microseconds.

Quick Quiz 5: Why the dip in refcnt overhead near 6 CPUs?



However, the restrictions that go with RCU can be quite onerous. For example, in many cases, the prohibition against sleeping while in an RCU read-side critical section would defeat the entire purpose. The next section looks at ways of addressing this problem, while also reducing the complexity of traditional reference counting, in some cases.

RCU is a Bulk Reference-Counting Mechanism

As noted in the preceding section, traditional reference counters are usually associated with a specific data structure, or perhaps a specific group of data structures. However, maintaining a single global reference counter for a large variety of data structures typically results in bouncing the cache line containing the reference count. Such cache-line bouncing can severely degrade performance.

In contrast, RCU's light-weight read-side primitives permit extremely frequent read-side usage with negligible performance degradation, permitting RCU to be used as a "bulk reference-counting" mechanism with little or no performance penalty. Situations where a reference must be held by a single task across a section of code that blocks may be accommodated with Sleepable RCU (SRCU). This fails to cover the not-uncommon situation where a reference is "passed" from one task to another, for example, when a reference is acquired when starting an I/O and released in the corresponding completion interrupt handler. (In principle, this could be handled by the SRCU implementation, but in practice, it is not yet clear whether this is a good tradeoff.)

Of course, SRCU brings a restriction of its own, namely that the return value from `srcu_read_lock()` be passed into the corresponding `srcu_read_unlock()`. The jury is still out as to how much of a problem is presented by this restriction, and as to how it can best be handled.

RCU is a Poor Man's Garbage Collector

A not-uncommon exclamation made by people first learning about RCU is "RCU is sort of like a garbage collector!". This exclamation has a large grain of truth, but it can also be misleading.

Perhaps the best way to think of the relationship between RCU and automatic garbage collectors (GCs) is that RCU resembles a GC in that the *timing* of collection is automatically determined, but that RCU differs from a GC in that: (1) the programmer must manually indicate when a given data structure is eligible to be collected, and (2) the programmer must manually mark the RCU read-side critical sections where references might legitimately be held.

Despite these differences, the resemblance does go quite deep, and has appeared in at least one theoretical analysis of RCU. Furthermore, the first RCU-like mechanism I am aware of used a garbage collector to handle the grace periods. Nevertheless, a better way of thinking of RCU is described in the following section.

RCU is a Way of Providing Existence Guarantees

Gamsa et al. [PDF] discuss existence guarantees and describe how a mechanism resembling RCU can be used to provide these existence guarantees (see section 5 on page 7). The effect is that if any RCU-protected data element is accessed within an RCU read-side critical section, that data element is guaranteed to remain in existence for the duration of that RCU read-side critical section.

Alert readers will recognize this as only a slight variation on the original "RCU is a way of waiting for things to finish" theme, which is addressed in the following section.

RCU is a Way of Waiting for Things to Finish

As noted in the first article in this series, an important component of RCU is a way of waiting for RCU readers to finish. One of RCU's great strengths is that it allows you to wait for each of thousands of different things to finish without having to explicitly track each and every one of them, and without having to worry about the performance degradation, scalability limitations, complex deadlock scenarios, and memory-leak hazards that are inherent in schemes that use explicit tracking.

In this section, we will show how `synchronize_sched()`'s read-side counterparts (which include anything that disables preemption, along with hardware operations and primitives that disable irq) permit you to implement interactions with non-maskable interrupt (NMI) handlers that would be quite difficult if using locking. I called this approach "Pure RCU" in my dissertation [PDF], and it is used in a number of places in the Linux kernel.

The basic form of such "Pure RCU" designs is as follows:

1. Make a change, for example, to the way that the OS reacts to an NMI.
2. Wait for all pre-existing read-side critical sections to completely finish (for example, by using the `synchronize_sched()` primitive). The key observation here is that subsequent RCU read-side critical sections are guaranteed to see whatever change was made.
3. Clean up, for example, return status indicating that the change was successfully made.

The remainder of this section presents example code adapted from the Linux kernel. In this example, the `timer_stop` function uses`synchronize_sched()` to ensure that all in-flight NMI notifications have completed before freeing the associated resources. A simplified version of this code follows:

```
1 struct profile_buffer {
2   long size;
3   atomic_t entry[0];
4 };
```

```
 5 static struct profile_buffer *buf = NULL;
 6
 7 void nmi_profile(unsigned long pcvalue)
 8 {
 9   atomic_t *p = rcu_dereference(buf);
10
11   if (p == NULL)
12     return;
13   if (pcvalue >= p->size)
14     return;
15   atomic_inc(&p->entry[pcvalue]);
16 }
17
18 void nmi_stop(void)
19 {
20   atomic_t *p = buf;
21
22   if (p == NULL)
23     return;
24   rcu_assign_pointer(buf, NULL);
25   synchronize_sched();
26   kfree(p);
27 }
```

Lines 1-4 define a profile_buffer structure, containing a size and an indefinite array of entries. Line 5 defines a pointer to a profile buffer, which is presumably initialized elsewhere to point to a dynamically allocated region of memory.

Lines 7-16 define the `nmi_profile()` function, which is called from within an NMI handler. As such, it cannot be preempted, nor can it be interrupted by a normal irq handler, however, it is still subject to delays due to cache misses, ECC errors, and cycle stealing by other hardware threads within the same core. Line 9 gets a local pointer to the profile buffer using the `rcu_dereference()` primitive to ensure memory ordering on DEC Alpha, and lines 11 and 12 exit from this function if there is no profile buffer currently allocated, while lines 13 and 14 exit from this function if the `pcvalue` argument is out of range. Otherwise, line 15 increments the profile-buffer entry indexed by the `pcvalue` argument. Note that storing the size with the buffer guarantees that the range check matches the buffer, even if a large buffer is suddenly replaced by a smaller one.

Lines 18-27 define the `nmi_stop()` function, where the caller is responsible for mutual exclusion (for example, holding the correct lock). Line 20 fetches a pointer to the profile buffer, and lines 22 and 23 exit the function if there is no buffer. Otherwise, line 24 NULLs out the profile-buffer pointer (using the `rcu_assign_pointer()` primitive to maintain memory ordering on weakly ordered machines), and line 25 waits for an RCU Sched grace period to elapse, in particular, waiting for all non-preemptible regions of code, including NMI handlers, to complete. Once execution continues at line 26, we are guaranteed that any instance of `nmi_profile()` that obtained a pointer to the old buffer has returned. It is therefore safe to free the buffer, in this case using the`kfree()` primitive.

In short, RCU makes it easy to dynamically switch among profile buffers (you just *try* doing this efficiently with atomic operations, or at all with locking!). However, RCU is normally used at a higher level of abstraction, as was shown in the previous sections.

Quick Quiz 6: Suppose that the `nmi_profile()` function was preemptible. What would need to change to make this example work correctly?

Conclusions

At its core, RCU is nothing more nor less than an API that provides:

1. a publish-subscribe mechanism for adding new data,

2. a way of waiting for pre-existing RCU readers to finish, and
3. a discipline of maintaining multiple versions to permit change without harming or unduly delaying concurrent RCU readers.

That said, it is possible to build higher-level constructs on top of RCU, including the reader-writer-locking, reference-counting, and existence-guarantee constructs listed in the earlier sections. Furthermore, I have no doubt that the Linux community will continue to find interesting new uses for RCU, as well as for any of a number of other synchronization primitives.

Acknowledgements

We are all indebted to Andy Whitcroft, Jon Walpole, and Gautham Shenoy, whose review of an early draft of this document greatly improved it. I owe thanks to the members of the Relativistic Programming project and to members of PNW TEC for many valuable discussions. I am grateful to Dan Frye for his support of this effort.

This work represents the view of the author and does not necessarily represent the view of IBM.

Answers to Quick Quizzes

Quick Quiz 1: WTF??? How the heck do you expect me to believe that RCU has a 100-femtosecond overhead when the clock period at 3GHz is more than 300 *picoseconds*?

Answer: First, consider that the inner loop used to take this measurement is as follows:

```
1 for (i = 0; i < CSCOUNT_SCALE; i++) {
2   rcu_read_lock();
3   rcu_read_unlock();
4 }
```

Next, consider the effective definitions of `rcu_read_lock()` and `rcu_read_unlock()`:

```
1 #define rcu_read_lock()   do { } while (0)
2 #define rcu_read_unlock() do { } while (0)
```

Consider also that the compiler does simple optimizations, allowing it to replace the loop with:

```
i = CSCOUNT_SCALE;
```

So the "measurement" of 100 femtoseconds is simply the fixed overhead of the timing measurements divided by the number of passes through the inner loop containing the calls to `rcu_read_lock()` and `rcu_read_unlock()`. And therefore, this measurement really is in error, in fact, in error by an arbitrary number of orders of magnitude. As you can see by the definition of `rcu_read_lock()` and `rcu_read_unlock()` above, the actual overhead is precisely zero.

It certainly is not every day that a timing measurement of 100 femtoseconds turns out to be an overestimate!

Back to Quick Quiz 1.

Quick Quiz 2: Why does both the variability and overhead of rwlock decrease as the critical-section overhead increases?

Answer: Because the contention on the underlying `rwlock_t` decreases as the critical-section overhead increases. However, the rwlock overhead will not quite drop to that on a single CPU

because of cache-thrashing overhead.

Back to Quick Quiz 2.

Quick Quiz 3: Is there an exception to this deadlock immunity, and if so, what sequence of events could lead to deadlock?

Answer: One way to cause a deadlock cycle involving RCU read-side primitives is via the following (illegal) sequence of statements:

```
idx = srcu_read_lock(&srcucb);
synchronize_srcu(&srcucb);
srcu_read_unlock(&srcucb, idx);
```

The `synchronize_rcu()` cannot return until all pre-existing SRCU read-side critical sections complete, but is enclosed in an SRCU read-side critical section that cannot complete until the `synchronize_srcu()`returns. The result is a classic self-deadlock--you get the same effect when attempting to write-acquire a reader-writer lock while read-holding it.

Note that this self-deadlock scenario does not apply to RCU Classic, because the context switch performed by the `synchronize_rcu()` would act as a quiescent state for this CPU, allowing a grace period to complete. However, this is if anything even worse, because data used by the RCU read-side critical section might be freed as a result of the grace period completing.

In short, do not invoke synchronous RCU update-side primitives from within an RCU read-side critical section.

Back to Quick Quiz 3.

Quick Quiz 4: But wait! This is exactly the same code that might be used when thinking of RCU as a replacement for reader-writer locking! What gives?

Answer: This is an effect of the Law of Toy Examples: beyond a certain point, the code fragments look the same. The only difference is in how we think about the code. However, this difference can be extremely important. For but one example of the importance, consider that if we think of RCU as a restricted reference counting scheme, we would never be fooled into thinking that the updates would exclude the RCU read-side critical sections.

It nevertheless is often useful to think of RCU as a replacement for reader-writer locking, for example, when you are replacing reader-writer locking with RCU.

Back to Quick Quiz 4.

Quick Quiz 5: Why the dip in refcnt overhead near 6 CPUs?

Answer: Most likely NUMA effects. However, there is substantial variance in the values measured for the refcnt line, as can be seen by the error bars. In fact, standard deviations range in excess of 10% of measured values in some cases. The dip in overhead therefore might well be a statistical aberration.

Back to Quick Quiz 5.

Quick Quiz 6: Suppose that the `nmi_profile()` function was preemptible. What would need to change to make this example work correctly?

Answer: One approach would be to use `rcu_read_lock()` and `rcu_read_unlock()` in `nmi_profile()`, and to replace the`synchronize_sched()` with `synchronize_rcu()`, perhaps as follows:

```
 1 struct profile_buffer {
 2   long size;
 3   atomic_t entry[0];
 4 };
 5 static struct profile_buffer *buf = NULL;
 6
 7 void nmi_profile(unsigned long pcvalue)
 8 {
 9   atomic_t *p;
10
11   rcu_read_lock();
12   p = rcu_dereference(buf);
13   if (p == NULL) {
14     rcu_read_unlock();
15     return;
16   }
17   if (pcvalue >= p->size) {
18     rcu_read_unlock();
19     return;
20   }
21   atomic_inc(&p->entry[pcvalue]);
22   rcu_read_unlock();
23 }
24
25 void nmi_stop(void)
26 {
27   atomic_t *p = buf;
28
29   if (p == NULL)
30     return;
31   rcu_assign_pointer(buf, NULL);
32   synchronize_rcu();
33   kfree(p);
34 }
```

Back to Quick Quiz 6.