# Extending RCU for Realtime and Embedded Workloads

Paul E McKenney
*IBM Linux Technology Center*
paulmck@us.ibm.com

Dipankar Sarma
*IBM India Software Labs*
dipankar@in.ibm.com

Ingo Molnar
*Red Hat*
mingo@elte.hu

Suparna Bhattacharya
*IBM India Software Labs*
bsuparna@in.ibm.com

## Abstract

This past year has seen significant increases in RCU's realtime capabilities, particularly the ability to preempt RCU read-side critical sections. There have even been some cases where use of RCU *improved* realtime latency (and performance and scalability as well), in contrast to earlier implementations, which seemed only to get in the way of realtime response. That said, there is still considerable room for improvement, including (1) lower-overhead `rcu_read_lock()` and `rcu_read_unlock()` primitives, (2) more scalable grace-period detection, (3) better balance of throughput and latency for RCU callback invocation, (4) lower per-structure memory overhead and (5) priority boosting of RCU read-side critical sections. This latter is needed to prevent low-priority tasks from blocking grace periods, resulting in out-of-memory events, due to being preempted for too long while in an RCU read-side critical section.

This paper describes ongoing work to address these five issues, including some interesting failures in addition to a number of unexpected successes. The ultimate goal of providing a single RCU implementation that covers all workloads is tantalizingly close, but is not yet within our grasp. It is safe to say that the very wide variety of workloads supported by Linux™ provides substantial challenges to the design and implementation of synchronization primitives like RCU!

## 1 Introduction

RCU is a synchronization mechanism that provides extremely low-overhead read-side access to shared data structures: in the theoretical best case (which is actually realized in non-realtime server workloads), the read-side RCU primitives generate no code whatsoever. Writers split their updates into "removal" and "reclamation" phases, where the "removal" phase typically removes an element from a globally accessible list, and the "reclamation" phase typically frees the element once it is known that all readers have dropped any references to the previously removed element. Readers do not block and are not blocked by writers, but writers must use some mutual exclusion mechanism to coordinate concurrent updates. RCU does not care what mechanism the writers use.

Readers use the `rcu_read_lock()` and `rcu_`

`read_unlock()` primitives to mark RCU read-side critical sections, and writers use `synchronize_rcu()` (or its continuation form, `call_rcu()`) to wait for a "grace period" to elapse, where all RCU read-side references obtained before the beginning of a given grace period are guaranteed to have been dropped by the end of that grace period. There are a number of other RCU API members, which are discussed at length elsewhere [4, 7], but which are not critical to this paper.

A realtime RCU implementation in an OS kernel must provide the following properties [11]:

1. Deferred destruction. No data element may be destroyed (for example, freed) while an RCU read-side critical section is referencing it.

2. Reliable. The implementation must not be prone to gratuitous failure.

3. Callable from IRQ (interrupt-handler) context.

4. Preemptible RCU read-side critical sections.

5. Small memory footprint. Many realtime systems are configured with modest amounts of memory, so it is highly desirable to limit memory overhead, including the number of outstanding RCU callbacks.

6. Independent of memory blocks. The implementation should not make assumptions about the size and extent of the data elements being protected by RCU, since such assumptions constrain memory allocation design, possibly imposing increased complexity.

7. Synchronization-free read side. RCU read-side critical sections should avoid cache misses and expensive operations, such as atomic instructions, memory barriers, and interrupt disabling.

8. Freely nestable read-side critical sections.

9. Unconditional read-to-write upgrade. RCU permits a read-side critical section to freely acquire the corresponding write-side lock – if two CPUs are both in an RCU read-side critical section, and if they both attempt to acquire the same lock, they must acquire the lock in turn, with no possibility of failure or deadlock, and without needing to exit their RCU read-side critical sections.

10. Compatible API. A realtime RCU implementation should have an API compatible with that of "classic RCU".

| | Callable From IRQ? | Preemptible Read Side? | Small Memory Footprint? | Synchronization-Free Read Side? | Independent of Memory Blocks? | Freely Nestable Read Side? | Unconditional Read-to-Write Upgrade? | Compatible API? |
|---|---|---|---|---|---|---|---|---|
| Classic RCU [18] | | N | N | | | | | |
| Preemptible RCU [12, 17] | | | X | | | | | |
| Jim Houston Patch [3] | | N | | N | | | | |
| Reader-Writer Locking | | | | N | | N | N | n |
| Hazard Pointers [13] | ? | | ? | n | N | | | ? |
| Lock-Based Deferred Free [8, 10] | ? | | | N | | | | |
| Read-Side GP Suppression [16] | | | N | n | | | | |
| Counters w/ Flipping [1, 11] | | | | n | | | | |

Table 1: Realtime RCU State of the Art

Table 1 summarizes the state of the art upon which this paper builds, showing how well each implementation meets the realtime-RCU criteria. In the table, "n" indicates a minor problem, "N" indicates a major problem, "X" indicates an absolute show-stopper problem, and

"?" indicates a possible problem depending on details of the implementation [11]. As can be seen from the table, none of these pre-existing RCU implementations meets all of the criteria. The "Counters w/ Flipping" approach (described in Section 2.1) comes closest, having only minor problems with property 7. This paper looks at ways of improving this approach in order to alleviate these problems, with the long-term goal of enabling a single RCU implementation to serve the full range of Linux workloads. This paper also focusses on property 5 for small-memory embedded machines.

The remainder of this paper is organized by the topics listed in the abstract, with Section 2 focusing on reducing RCU read-side overhead, Section 3 covering improvements to grace-period detection scalability and performance, Section 4 reviewing recent work on balancing callback throughput and latency, Section 5 discussing support of systems with extremely small memories (by 2006 standards, anyway), and Section 6 previewing work to prevent indefinite preemption from resulting in indefinite-duration grace periods. Finally, Section 7 presents summary and conclusions.

# 2 Reduced-Overhead Read Side

The first attempts to produce a realtime-friendly implementation of RCU had serious drawbacks. Sarma and McKenney addressed excessive realtime latencies imposed by long sequences of RCU callbacks [17], but this implementation did nothing to alleviate latencies that could be induced by long RCU read-side critical sections, which ran with preemption disabled. At the time, all known preemptible RCU-like implementations were subject to indefinite-duration grace periods, which could in turn result in system hangs due to memory exhaustion.

In early 2005, McKenney described a lock-based approach [10] that allowed RCU read-side critical sections to run with preemption enabled, thus permitting good process-scheduling latencies in face of long RCU read-side critical sections in Ingo Molnar's -rt patchset [14] (see the LWN article [9] for an overview). However, this implementation allowed realtime latencies to "bleed" from one RCU read-side critical to another, due to its lock-based grace-period-detection mechanism. Subsequent discussions on the Linux-kernel mailing list suggested that a counter-based approach might avoid this problem [11].

The following sections review this counter-based implementation and subsequent improvements.

## 2.1 Simple Counter-Based Implementation

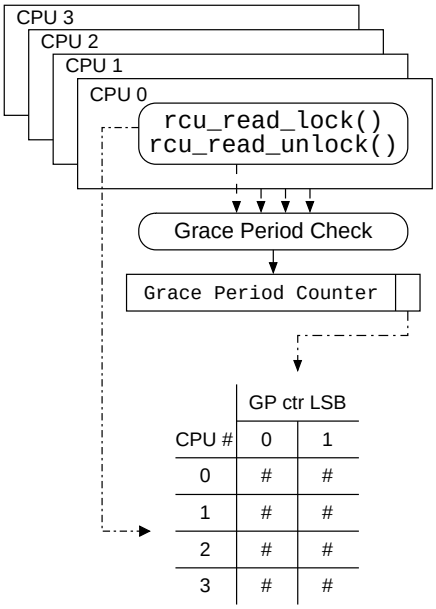


Figure 1: Simple Data Flow

Figure 1 gives a schematic depicting the operation of the simple counter-based algorithm. The basic idea is to maintain per-CPU arrays, with

each array containing a pair of counters [2]. At any given time, one of the pair will be the "current" counter, and the other the "last" counter. Oversimplifying somewhat for clarity, each invocation of `rcu_read_lock()` on a given CPU atomically[1] increments that CPU's "current" counter, and each invocation of `rcu_read_unlock()` atomically decrements whatever counter the corresponding `rcu_read_lock()` incremented. Whenever the end of a grace period is detected, the roles of the "current" and "last" counters are swapped. This means that the "last" counters will now be atomically decremented, but (almost) never incremented, so that they will eventually all reach zero. Once they have all reached zero, we have detected the end of another grace period, and can then swap the roles of the counters once more in order to detect the end of the next grace period.

```
 1 void rcu_read_lock(void)
 2 {
 3   int f;
 4   unsigned long oldirq;
 5   struct task_struct *t = current;
 6
 7   raw_local_irq_save(oldirq);
 8   if (t->rcu_read_lock_nesting++ == 0) {
 9     f = rcu_ctrlblk.completed & 1;
10     smp_read_barrier_depends();
11     t->rcu_flipctr1 =
12       &(__get_cpu_var(rcu_flipctr)[f]);
13     atomic_inc(t->rcu_flipctr1);
14     smp_mb__after_atomic_inc();
15     if (f != (rcu_ctrlblk.completed & 1)) {
16       t->rcu_flipctr2 =
17         &(__get_cpu_var(rcu_flipctr)[!f]);
18       atomic_inc(t->rcu_flipctr2);
19       smp_mb__after_atomic_inc();
20     }
21   }
22   raw_local_irq_restore(oldirq);
23 }
```

Figure 2: Memory-Barrier rcu_read_lock()

The actual sequence of events is a bit more involved. The code for the `rcu_read_lock()` primitive is shown in Figure 2. Lines 7 and 22 suppress and restore interrupts, preventing destructive races between an interrupted `rcu_read_lock()` and a second `rcu_read_lock()` invoked by the interrupt handler. For example, if an interrupt were allowed to occur just after line 8 of the first `rcu_read_lock()`, the interrupt handler's invocation would incorrectly believe that it was completely nested in the interrupted RCU read-side critical section, and thus that it was already protected. This situation could result in premature grace-period completion, and memory corruption. This problem is avoided by suppressing interrupts.

Line 8 increments a per-task-struct RCU read-side nesting-level counter. If the prior value of this counter was non-zero, this is a nested RCU read-side critical section, which is already protected by the outermost critical section. Otherwise, execution proceeds through lines 9-20, which prevent any future grace periods from completing. Lines 9 snapshots the bottom bit of a grace-period counter, and this bit is used to index into one of two elements of a per-CPU counter array, as depicted in Figure 1. Line 10 forces the subsequent array access to be ordered after the computation of its index on Alpha CPUs. Lines 11-12 compute a pointer to the "current" element of the current CPU's counter array, recording this pointer in the task structure. Line 13 then atomically increments this counter, and line 14 adds a memory barrier on architectures for which `atomic_inc()` is not an implicit memory barrier. The memory barrier is required to prevent the contents of the critical section from bleeding out into earlier code.

Since `rcu_read_lock()` does not acquire any explicit locks, it is possible for its execution to race with the detection of the end of an ongoing grace period. This means that the `completed` counter could change at any time during `rcu_read_lock()` execution, which in turn could lead to premature ending of the

[1] Sections 2.2, 2.3, and 2.4 describe various schemes to eliminate atomic instructions and memory barriers.

next grace period, since `rcu_read_lock()` would then erroneously be preventing the end of the second grace period rather than the next grace period. Line 15 detects this race, and, if detected, lines 16-19 atomically increment the CPU's other counter, thus preventing the end of both the next grace period and the one after that. The additional overhead of the second atomic operation and memory barrier is incurred only in the unlikely event of a race between grace-period detection and `rcu_read_lock()` execution.

```
 1 void rcu_read_unlock(void)
 2 {
 3   unsigned long oldirq;
 4   struct task_struct *t = current;
 5
 6   raw_local_irq_save(oldirq);
 7   if (--t->rcu_read_lock_nesting == 0) {
 8     smp_mb__before_atomic_dec();
 9     atomic_dec(t->rcu_flipctr1);
10     t->rcu_flipctr1 = NULL;
11     if (t->rcu_flipctr2 != NULL) {
12       atomic_dec(t->rcu_flipctr2);
13       t->rcu_flipctr2 = NULL;
14     }
15   }
16   raw_local_irq_restore(oldirq);
17 }
```

Figure 3: Memory-Barrier rcu_read_unlock()

The code for `rcu_read_unlock()` is shown in Figure 3. Lines 6 and 16 suppress and restore hardware interrupts to prevent race conditions analogous to those for `rcu_read_lock()`. Line 7 checks to see if we are exiting the outermost RCU read-side critical section; if not, we retain the outer critical section's protection. Otherwise, lines 8-14 update state to allow grace periods to complete.

Lines 8 and 9 execute an atomic decrement, along with a memory barrier, the latter required to prevent the RCU read-side critical section from bleeding out into subsequent code. Line 10 NULLs the pointer in the task structure, primarily for debug purposes. Note that this counter might well belong to some other CPU, for example, if this task was preempted during its RCU read-side critical section. This possibility is one reason that the increments and decrements must be atomic. Line 11 detects the case where `rcu_read_lock()` had to increment both of the CPU's counters, and, if so, lines 12 and 13 atomically decrement it and NULL out the corresponding pointer in the task structure.

```
 1 static void rcu_try_flip(void)
 2 {
 3   int c;
 4   long f;
 5   unsigned long m;
 6
 7   f = rcu_ctrlblk.completed;
 8   if (!spin_trylock_irqsave
 9       (&rcu_ctrlblk.fliplock, m)) {
10     return;
11   }
12   if (f != rcu_ctrlblk.completed) {
13     spin_unlock_irqrestore
14         (&rcu_ctrlblk.fliplock, m);
15     return;
16   }
17   f &= 1;
18   for_each_cpu(c) {
19     if (atomic_read(&per_cpu
20         (rcu_flipctr, c)[!f]) != 0) {
21       spin_unlock_irqrestore
22         (&rcu_ctrlblk.fliplock, m);
23       return;
24     }
25   }
26   smp_mb();
27   rcu_ctrlblk.completed++;
28   spin_unlock_irqrestore
29     (&rcu_ctrlblk.fliplock, m);
30 }
```

Figure 4: Memory-Barrier GP Detection

The code that detects the end of a grace period is shown in Figure 4, and is invoked from the scheduling-clock interrupt. Line 7 records the current value of the `completed` field, which is a count of the number of grace periods detected since boot. Lines 8-9 attempt to acquire the spinlock guarding grace-period detection, and, if unsuccessful, line 10 returns, relying on the task holding the lock to continue doing so.

On the other hand, if we successfully acquire the lock, we proceed to line 12, which checks whether a grace period was detected while we

were acquiring the lock. If so, someone else detected the grace period for us, and we need not repeat the work. Lines 13-15 therefore release the lock and return.

Otherwise, line 17 isolates the low-order bit of the grace-period counter, which is used to identify the "last" counter in each per-CPU array. Lines 18-25 loop through each CPU, with lines 19-20 checking the value of the "last" counter. If any are non-zero, the grace period has not yet ended, in which case lines 21-23 release the lock and return.

If all the "last" counters are zero, the current grace period has ended. Line 26 then executes a memory barrier to ensure that line 27's counting of the newly ended grace period does not bleed back into the earlier counter checks, then lines 28-29 release the lock.

The first patch for a robust implementation of this simple counter-based realtime RCU implementation was posted to LKML in August 2005 [6]. This patch has the shortcomings described in the LKML posting:

1. The `rcu_read_lock()` and `rcu_read_unlock()` primitives contain heavyweight operations, including atomic operations, memory barriers, and disabling of hardware interrupts. These heavyweight operations are undesireable in a primitive whose sole purpose is to provide high-performance read-side operation.

2. The grace-period detection code uses a single global queue, which can result in an SMP locking bottleneck on larger machines.

3. The grace-period detection code is probably too aggressive, particularly on server-class machines with ample memory. A more sophisticated grace-period-detection mechanism would: (1) allow for long grace periods when memory was plentiful, thus permitting the overhead of grace-period detection to be amortized over a larger number of RCU accesses and updates, but (2) seek to minimize grace-period duration when memory was scarce, thus minimizing the amount of memory consumed by outstanding RCU callbacks.

The remainder of this section focusses on the first issue. The second issue is taken up in Section 3. The third issue is beyond the scope of this paper – in the near term, we need to retain "classic RCU" for use in server workloads, but longer term, there is some hope that a single converged RCU mechanism might serve both server and realtime environments. Section 2.5 describes some criteria that need to be met in order to produce such a converged mechanism.

### 2.2 Remove Common-Case Atomic Operations

Although atomic increments and decrements are necessary in the general case in Figures 2 and 3, there are some special cases where there can be at most one CPU manipulating a given counter. The first such case is at line 13 of Figure 2 when the counter is zero. In this case, there cannot possibly be some other CPU attempting to decrement the counter, as it would need to be non-zero for this to happen. In addition, there cannot be some other CPU attempting to increment the counter, since interrupts are disabled, preventing preemption. Therefore, lines 13-14 can be replaced by the following:

```
if (atomic_read(current->rcu_flipctr1) == 0) {
  atomic_set(current->rcu_flipctr1,
    atomic_read(current->rcu_flipctr1) + 1);
  smp_mb();
} else {
```

```
  atomic_inc(current->rcu_flipctr1);
  smp_mb__after_atomic_inc();
}
```

Note that both `atomic_read()` and `atomic_set()` are simple structure-access wrappers; despite the "atomic" in their names, neither of these primitives use atomic instructions or memory barriers. If preemption is rare, the "then"-clause of the above "if" statement should be taken most frequently, avoiding the `atomic_inc()`. A similar change can be made to lines 8-9 of Figure 3, but with an additional check to ensure that the task is running on the same CPU that executed the corresponding `rcu_read_lock()`.

However, this change does not help on x86 machines, since the non-atomic operations must still be accompanied by memory barriers, which, on x86, are implemented using atomic instructions. This situation motivates elimination of these memory barriers, covered in the next section.

### 2.3 Remove Memory Barriers

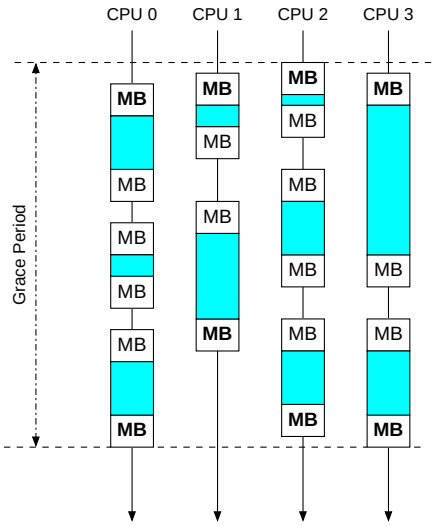


Figure 5: Wasted Memory Barriers

The memory barriers in `rcu_read_lock()` and `rcu_read_unlock()` are needed only to guard against races with grace-period detection, which is normally quite rare compared to RCU read-side critical sections. In Figure 5 each shaded box represents an RCU read-side critical section with associated memory barriers. Only the emboldened MBs represent required memory barriers; the rest consume overhead but provide no added protection. This situation indicates that memory barriers should be associated with grace-period detection rather than RCU read-side critical sections. One simple way to accomplish this is to maintain a per-CPU flag indicating that grace-period detection is in progress. Neither the `rcu_read_lock()` nor the `rcu_read_unlock()` primitive needs memory barriers, although `rcu_read_unlock()` could continue to use them when it is necessary to expedite grace-period detection. Instead, the per-CPU scheduling-clock interrupt handler would execute a memory barrier only (1) when the corresponding per-CPU flag is set and (2) after the corresponding CPU's "last" counter had reached zero. This single memory barrier would protect both the preceding and the following RCU read-side critical sections, as shown in Figure 6.

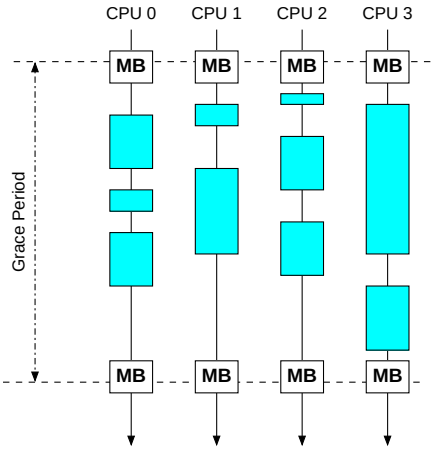


Figure 6: Grace-Period Memory Barriers

The general approach is shown in Figure 7.

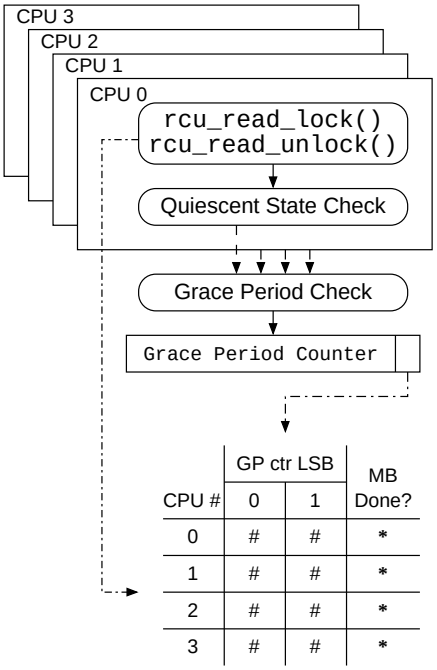


Figure 7: No-Memory-Barrier Data Flow

This approach offers performance benefits, even on x86, but atomic instructions are still required in case of preempted RCU read-side critical sections. It is possible to do better.

### 2.4 Remove All Atomic Instructions and Memory Barriers

Note that `CONFIG_PREEMPT` kernels limit `rcu_read_lock()` nesting depth, since infinite nesting would overflow the `preempt_disable()` counter. This limited nesting depth permits each CPU to increment and decrement its own counters, regardless of what CPU the corresponding `rcu_read_lock()` might have run on. Simply summing the per-CPU "last" counters would give the number of outstanding RCU read-side critical sections holding up the current grace period.

The implementation of `rcu_read_lock()` becomes quite simple, as shown in Figure 8.

```
 1 void rcu_read_lock(void)
 2 {
 3   unsigned long oldirq;
 4   struct task_struct *t = current;
 5
 6   raw_local_irq_save(oldirq);
 7   if (t->rcu_read_lock_nesting++ == 0) {
 8     t->rcu_i = rcu_ctrlblk.completed & 1;
 9     smp_read_barrier_depends();
10     __get_cpu_var(rcu_flipctr)[t->rcu_i]++;
11   }
12   raw_local_irq_restore(oldirq);
13 }
```

Figure 8: Non-MB rcu_read_lock()

Lines 6 and 12 suppress interrupts, as before. Line 7 increments this task's RCU read-side critical-section nesting level, and, if this is the outermost such critical section, executes lines 8-10 to prevent subsequent grace periods from completing. Line 8 records the index of the "current" counter for use by the corresponding `rcu_read_unlock()`, line 9 provides memory barriers for systems that fail to force ordering of data-dependant loads (for example, DEC Alpha [5]), and line 10 increments this CPU's "current" counter, preventing subsequent grace periods from proceeding.

```
 1 void rcu_read_unlock(void)
 2 {
 3   unsigned long oldirq;
 4   struct task_struct *t = current;
 5
 6   raw_local_irq_save(oldirq);
 7   if (--t->rcu_read_lock_nesting == 0) {
 8     __get_cpu_var(rcu_flipctr)[t->rcu_i]--;
 9   }
10   raw_local_irq_restore(oldirq);
11 }
```

Figure 9: Non-MB rcu_read_unlock()

The `rcu_read_unlock()` is also quite simple, as shown in Figure 9. Lines 6 and 10 once again suppress hardware interrupts, line 7 decrements the RCU read-side critical-section nesting level, and, if outermost, line 8 decrements this CPU's counter, but with the same index as that incremented by the corresponding `rcu_read_lock()`.

Unfortunately, this approach invalidates the earlier trick used to get rid of common-case memory barriers, because a given CPU can no longer determine when its "last" counter is zero. We instead use a state machine to detect the end of grace periods, with states executed cyclicly as follows:

1. Idle: not attempting to detect the end of a grace period.
2. Grace period: marks the end of a grace period via a counter flip, and sets per-CPU flip-seen flags, which each CPU will clear after it has seen the flip.
3. Wait-ack: waits for each CPU to clear its flip-seen flag. Once each CPU has cleared its flip-seen flag, there can be no further increments to any of the "last" counters.
4. Wait-zero: waits for the sum of the "last" counters to reach zero. Once zero is reached, each CPU must execute a memory-barrier instruction to force ordering of prior RCU read-side critical sections. Therefore, we then set a per-CPU memory-barrier-done flag, which each CPU will clear after executing its memory barrier.
5. Wait-memory-barrier: waits for all CPUs to execute a memory barrier and clear its memory-barrier-done flag. As before, these memory barriers protect both the earlier and the subsequent RCU read-side critical sections.

Note that this state machine results in a "fuzzy" grace-period boundary extending from state 2 to state 5. This requires an extra staging queue for RCU callbacks, or that the callbacks are advanced only once per two grace periods.

With this state machine, it is not necessary for individual CPUs to determine when their particular counter has reached zero. Instead, once the sum of the counters has reached zero, each CPU is explicitly asked to execute a memory barrier. The data flow is shown in Figure 10.

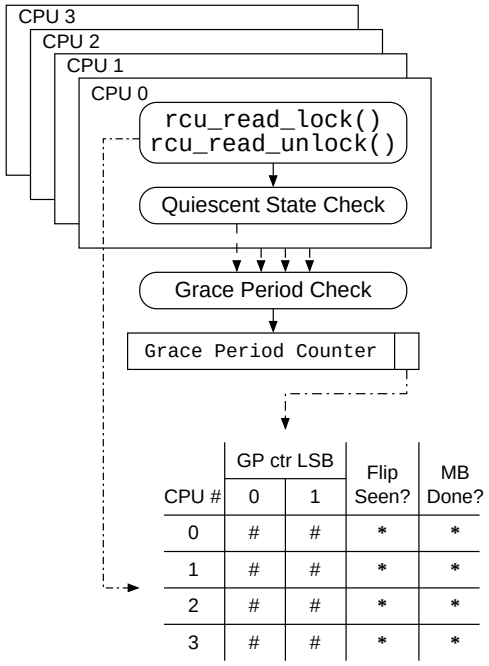


Figure 10: No-Atomic Data Flow

Although this approach results in lightweight read-side primitives, it also increase grace-period detection time by a few scheduling-clock periods compared to the implementations described in the previous sections. It should be possible to overcome this effect, for example, by causing `call_rcu()` to invoke grace-period detection if callbacks are arriving too quickly.

In addition, the read-side primitives still disable interrupts in order to provide the guarantee that all future invocations will be using the correct element of the counter array just after a counter flip. Removing this interrupt disabling is a topic for future investigation.

| Kernel | # Samples | Overhead (ns) | Std |
|---|---|---|---|
| 2.6.15-rt16 (2.1) | 92 | 172.06 | 0.22 |
| 2.6.15-rt16 optatomic (2.2) | 131 | 232.06 | 0.35 |
| 2.6.15-rt16 optmb (2.3) | 84 | 115.09 | 0.08 |
| 2.6.15-rt16 nonatomic (2.4) | 254 | 93.89 | 0.06 |
| 2.6.15 `CONFIG_PREEMPT` | 393 | 10.87 | 0.06 |
| 2.6.15 | 61 | 0.63 | 0.05 |

Table 2: RCU Readside Overhead on 2.8GHz Xeon CPU

### 2.5 Converging Classic and Realtime RCU

Can classic RCU be totally supplanted by realtime RCU? The jury is still out on this question, but here are some criteria that realtime RCU must first meet:

1. It must be rock solid across the full gamut of workloads.
2. Any required tuning for SMP servers must be automated, e.g., computed at boot time based on the amount of physical memory, the number of CPUs, etc.
3. It must be impose negligible additional overhead compared to classic RCU on SMP machines.

The last criterion might be weakened, as it may be acceptable to revert to classic RCU for SMP/NUMA machines with more than (say) 16 CPUs. However, it may be possible to cast realtime RCU into hierarchical form, which could reduce overhead [19]. In any case, the fewer the implementations of RCU, the smaller the testing and maintenance burden imposed by RCU. This situation motivates us to continue working towards the goal of single universal RCU implementation.

Table 2 shows that there may be some hope. The situation has improved about a factor of two, but there is still an order of magnitude to go before we reach the performance of `CONFIG_PREEMPT` RCU.

## 3 Scalable Grace-Period Detection

The current RCU implementation in the -rt tree [14] uses a single global callback queue. In the past, this has simplified the implementation, for example, by removing any CPU-hotplug considerations. However, RCU implementations that avoid memory barriers and atomic instructions *do* need to worry about CPU hotplug, due to their use of per-CPU memory-barrier and flip-seen flags. It therefore makes sense to move to per-CPU queuing for these implementations, since the CPU-hotplug complexity can no longer be avoided.

Other non-trivial changes will be required as well, for example, the heuristic used to determine when to invoke `rcu_check_callbacks()` will likely need to be revisited, most likely by appropriately updating `rcu_pending()`. Larger SMP machines may also need hierarchical bitmaps similar to Manfred Spraul's [19], as well as hierarchical summing of the "last" counters.

## 4 Callback Latency vs. Throughput

Applying RCU to the file structure had the unintended consequence of allowing a simple file open-close loop to generate RCU callbacks at a sufficient rate to exhaust memory. This was fixed by varying the permitted number of RCU callback invocations per softirq instance. Realtime implementations of RCU must gracefully handle this same situation.

One approach is for `call_rcu()` to invoke the grace-period detection code directly when there are large numbers of callbacks. However, the actual invocation of callbacks cannot be done

from `call_rcu()`, as this can result in deadlock. The callbacks must still be invoked from softirq context.

In the -rt tree [14], realtime tasks can preempt softirq handlers. Therefore, a system with runaway realtime processes that consume all available CPU would not execute callbacks at all. In addition, many other critical system services would fail to execute. Lack of critical system services, including RCU callback invocation, would result in system hangs or failures.

What should be done when the system is overloaded with realtime tasks? Realtime tasks must take precedence, but system services cannot be indefinitely delayed. This is a policy decision, with the following possible choices:

1. Degrade realtime response time, thereby keeping the system alive (for example, decrease priority of "hoggy" realtime tasks in order to permit debugging using non-realtime tools).
2. Panic the system and reboot, as might be required in some production realtime workloads.
3. Kill less-critical realtime tasks, thereby keeping system alive. Of course, this option requires some way of determining which tasks to kill.
4. "Fence" the realtime tasks, so that they are not permitted to consume excessive amounts of any given CPU's time [15]. This can be considered to be a variation on the first option.

It is likely that more than one of these will be required, but much experimentation with numerous realtime applications will be required to determine the right options and implementations.

# 5 Reduced Per-Struct Memory Overhead

Any structure passed to `call_rcu()` must contain a two-pointer `struct rcu_head` to track the structure and its callback function. This additional memory overhead is negligible in many environments, but on 32-bit embedded systems with small memory (e.g., 2MB), the additional eight bytes can be problematic. This section looks at the following three options for dealing with this problem:

1. Use `synchronize_rcu()` instead of `call_rcu()`, thus eliminating the need for the `struct rcu_head`.
2. Use the C `union` feature to multiplex the `struct rcu_head` with other fields that are not used by RCU-protected code paths.
3. Shrink the `struct rcu_head` so that it fits into 32 bits, reducing the memory it consumes.

The use of `synchronize_rcu()` has the advantages of reducing the RCU-protected structure by eight bytes rather than by only four, (usually) simplifying the code somewhat, and being already heavily used in the Linux kernel. However, because `synchronize_rcu()` blocks for a full grace period, its use is not appropriate in all situations.

The second option, use of C `union`, also reduces the RCU-protected structure by up to eight bytes rather than by only four, does not affect code complexity, and has seen some use, for example, the `struct dentry` unions the `d_child` list header with the `d_rcu` field. However, not all structures contain data that is unreferenced by all RCU code paths. Such data structures cannot make use of C `union` to reduce the memory overhead of `struct`

rcu_head. Furthermore, use of union can make some data structures more difficult to understand. Nevertheless, where it applies, use of C union is very simple and effective.

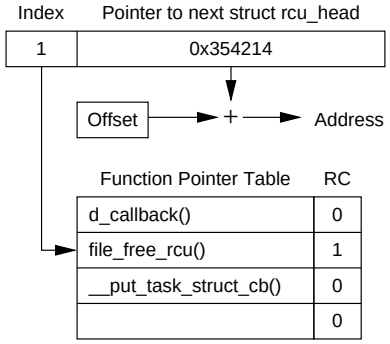


Figure 11: Shrink rcu_head Structure

People using systems with very small memories may wish to experiment with a mapping table to compress the function pointer into a few bits. Linux currently has roughly 40 functions passed to call_rcu(), requiring six bits of index, leaving 26 bits for the pointer to the next struct rcu_head, which could address 64MB of memory, as depicted in Figure 11. Systems with kernel memory mapped at an offset add that offset in order to reduce the number of bits required for the pointer.

If an embedded system were to load and unload modules in order to further reduce memory requirements, and if these modules used call_rcu(), then the index field in Figure 11 would only need to be large enough to handle the number of RCU callback functions that were actually loaded into the kernel at a given time. However, to realize this savings, it would be necessary to reclaim entries in the function pointer table corresponding to callbacks in modules that had been unloaded. One way to do this would be to use reference counts, as illustrated by the column labelled "RC" in Figure 11. Only file_free_rcu() has a non-zero reference count, permitting the slots occupied by the other function pointers to be reused as needed by call_rcu(). If the desired RCU callback already had a slot, call_rcu() would simply increment its reference count. In either case, the reference count would be decremented after invoking the callback function. In UP kernels, the table may be protected by simple disabling of interrupts. At present, it seems unlikely that this function-table approach would be used on SMP systems.

It is possible that the stripped-down Linux kernels used in embedded systems might have fewer uses of call_rcu(), and thus might be able decrease the number of bits of the struct rcu_head and increase the size of the pointer.

Of course, both synchronize_rcu() and C union are more generally useful and also provide greater per-structure memory savings. However, if these approaches are insufficient, it might be worthwhile considering a small-memory configuration parameter that shrinks the size of struct rcu_head for small-memory systems.

# 6 Priority Boosting

Many realtime workloads maintain low CPU utilization in order to avoid excessive latencies due to task queueing in the scheduler. However, any number of software bugs can cause "runaway" tasks to saturate the CPUs. If the CPUs are saturated with realtime tasks, the realtime RCU implementations described in Section 2 are vulnerable to indefinite grace-period durations caused by a low-priority non-realtime task being preempted while executing in an RCU read-side critical section. This can result in OOM conditions, especially on small-memory machines. Of course, as mentioned earlier, starving other system services can also result in system failures.

One way to avoid this problem is to boost the priority of tasks executing in RCU read-side critical sections, in a manner similar to mutex-based priority boosting. However, unlike with locking, it does not make sense to boost and decrease priority in the `rcu_read_lock()` and `rcu_read_unlock()` primitives, because this would require the introduction of locking into these primitives, in turn unacceptably increasing their overhead and destroying their deadlock-immunity properties. In fact, the performance degradation is worse than for locking, since lock-based priority boosting need do nothing except in the (presumably less-common) case where a high-priority task attempts to acquire a lock held by a lower-priority task.

A better approach is to recognize that it does not help to boost the priority of a task that is already running. Boosting its priority will not make it run faster, in fact, the resulting cache-thrashing will likely slow it down. No action need be taken until the task either is preempted or attempts to acquire an already-held lock while still in its RCU read-side critical section, at which point that task's priority can be boosted to the highest non-realtime priority. It may also be necessary to further boost the priority of RCU read-side critical sections when the system exhausts memory.

We have been experimenting with this approach, but additional work is needed to arrive at a simple and stable patch. It is possible that restricting the CPU time that may be consumed by realtime tasks [15] will prove a more fruitful approach, at least in the near term.

## 7 Conclusions

Although there is more work to be done, it appears that a robust and efficient realtime-friendly implementation of RCU is quite feasible. We have shown how atomic instructions and memory barriers can be eliminated from RCU read-side primitives, and how standard techiques, with some innovation, can yield a scalable grace-period detection algorithm.

There has been good progress towards the right balance of RCU callback throughput and scheduling latency on realtime systems, but more work is needed to ensure that this balance is maintained for all workloads.

We described three ways of reducing per-structure `struct rcu_head` overhead, two of which eliminate this overhead completely and are available within the current kernel.org tree, and a third that requires some additional work and saves only 50% of the `struct rcu_head` overhead.

We described a mechanism for boosting the priority of preempted RCU readers in order to expedite grace-period end, however, we do not yet have a stable implementation of this mechanism. In the meantime, limiting the CPU consumption of realtime tasks should help, since this should allow the priority of any preempted RCU reader to age upwards. However, more work is needed to determine whether this approach will suffice in all cases.

The jury is still out as to whether a single RCU implementation can meet the needs of both realtime and SMP-server workloads, but the techniques described in this paper are approaching that goal. That said, whether or not this goal is eventually reached, the implementations described in this paper should improve Linux's ability to provide realtime response on SMP systems.

## Acknowledgements

We owe thanks to Esben Neilsen and Bill Huey for championing counter-based RCU, and to the many developers and users of the -rt tree for their hard work creating and testing this patch-set. We are grateful to Daniel Frye, Vijay Sukthankar, and Reena Malangone for their support of this effort.

## Legal Statement

This work represents the views of the authors and does not necessarily represent the view of Red Hat or of IBM.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

## References

[1] Paul E.McKenney. [RFC,PATCH] RCU and CONFIG_PREEMPT_RT sane patch. Available: `http://lkml.org/lkml/2005/8/1/155` [Viewed March 14, 2006], August 2005.

[2] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, pages 87–100, New Orleans, LA, February 1999.

[3] Jim Houston. [RFC&PATCH] Alternative RCU implementation. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=109387402400673&w=2` [Viewed February 17, 2005], August 2004.

[4] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf` [Viewed October 15, 2004].

[5] Paul E. McKenney. Memory ordering in modern microprocessors, part I. *Linux Journal*, 1(136):52–57, August 2005.

[6] Paul E. McKenney. Re: [fwd: Re: [patch] real-time preemption, -rt-2.6.13-rc4-v0.7.52-01]. Available: `http://lkml.org/lkml/2005/8/8/108` [Viewed March 14, 2006], August 2005.

[7] Paul E. McKenney. Read-copy update (RCU). Available: `http://www.rdrop.com/users/paulmck/RCU` [Viewed May 25, 2005], May 2005.

[8] Paul E. McKenney. Real-time preemption and RCU. Available: `http://lkml.org/lkml/2005/3/17/199` [Viewed September 5, 2005], March 2005.

[9] Paul E. McKenney. A realtime preemption overview. Available: `http://lwn.net/Articles/146861/` [Viewed August 22, 2005], August 2005.

[10] Paul E. McKenney. [RFC] RCU and CONFIG_PREEMPT_RT progress.

Available: http://lkml.org/lkml/2005/5/9/185 [Viewed May 13, 2005], May 2005.

[11] Paul E. McKenney and Dipankar Sarma. Towards hard realtime response from the linux kernel on SMP hardware. In *linux.conf.au 2005*, Canberra, Australia, April 2005. Available: http://www.rdrop.com/users/paulmck/RCU/realtimeRCU.2005.04.23a.pdf [Viewed May 13, 2005].

[12] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002. Available: http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz [Viewed June 23, 2004].

[13] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[14] Ingo Molnar. Index of /mingo/realtime-preempt. Available: http://people.redhat.com/mingo/realtime-preempt/ [Viewed February 15, 2005], February 2005.

[15] Ingo Molnar. Index of /mingo/rt-limit-patches. Available: http://people.redhat.com/mingo/rt-limit-patches/ [Viewed March 31, 2006], January 2006.

[16] Esben Neilsen. Re: Real-time preemption and RCU. Available: http://lkml.org/lkml/2005/3/18/122 [Viewed March 30, 2006], March 2005.

[17] Dipankar Sarma and Paul E. McKenney. Making RCU safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)*, pages 182–191. USENIX Association, June 2004.

[18] John D. Slingwine and Paul E. McKenney. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Technical Report US Patent 5,442,758, US Patent and Trademark Office, Washington, DC, August 1995.

[19] Manfred Spraul. [rfc] 0/5 rcu lock update. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=108546407726602&w=2 [Viewed June 23, 2004], May 2004.