USENIX Association

Proceedings of the FREENIX Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

© 2004 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Making RCU Safe for Deep Sub-Millisecond Response Realtime Applications

Dipankar Sarma
Linux Technology Center
IBM Software Lab, India
dipankar@in.ibm.com

Paul E. McKenney
Linux Technology Center and Storage Software Architecture
IBM Corporation
Beaverton, OR 97006
Paul.McKenney@us.ibm.com
http://www.rdrop.com/users/paulmck

Abstract

Linux™ has long been used for soft realtime applications. More recent work is preparing Linux for more aggressive realtime use, with scheduling latencies in the small number of hundreds of microseconds (that is right, *micro*seconds, *not* milliseconds). The current Linux 2.6 RCU implementation both helps and hurts. It helps by removing locks, thus reducing latency in general, but hurts by causing large numbers of RCU callbacks to be invoked all at once at the end of the grace period. This batching of callback invocation improves throughput, but unacceptably degrades realtime response for the more discerning realtime applications.

This paper describes modifications to RCU that greatly reduce its effect on scheduling latency, without significantly degrading performance for non-realtime Linux servers. Although these modifications appear to prevent RCU from interfering with realtime scheduling, other Linux kernel components are still problematic. We are therefore working on tools to help identify the remaining problematic components and to definitively determine whether RCU is still an issue. In any case, to the best of our knowledge, this is the first time that anything resembling RCU has been modified to accommodate the needs of realtime applications.

1 Introduction

Tests of realtime response on the Linux 2.6 kernel found unacceptable scheduling latency, in part due to the batching of callbacks used in the RCU implementation. This batching is essential to good performance on non-realtime servers, since the larger the batch, the more callbacks the overhead of detecting an RCU grace period may be amortized over. However, because these callbacks run in a tasklet that runs at softirq level, callback processing cannot be preempted. Since heavy loads can result in well over a thousand RCU callbacks per grace period, RCU's contribution to scheduling latency can approach 500 microseconds, which far exceeds the amount that can be tolerated by some classes of realtime applications. Furthermore, extreme denial-of-service workloads have been observed to generate more than 30,000 RCU callbacks in a single grace period, which would result in a correspondingly greater degradation of scheduling latency. This situation motivated some modifications to RCU, with the goal of eliminating RCU's contribution to the excessive scheduling latency.

This paper presents some background on RCU in Section 2, describes the problem that was causing excessive scheduling latency in Section 3, discusses three proposed solutions in Section 4, and evaluates the three solutions in Section 5.

2 RCU Background

RCU is a reader-writer synchronization mechanism that takes asymmetric distribution of synchronization overhead to its logical extreme: read-side critical sections incur zero synchronization overhead, containing no locks, no atomic instructions, and, on most architectures, no memory-barrier instructions. RCU therefore achieves near-ideal performance for read-only workloads on most architectures. Write-side critical sections must therefore incur substantial synchronization overhead, deferring destruction and maintaining multiple versions of data structures in order to accommodate the read-side critical sections. In addition, writers must use some synchronization mechanism, such as locking, to provide for orderly updates. Readers must somehow inform writers when they finish so that writers can determine when it is safe to complete destructive operations.

In the Linux 2.6 kernel, RCU signals writers by non-atomically incrementing a local counter in the context-switch code. If this is a given CPU's first such increment for the current grace period, then the CPU clears its bit from a global bitmask. If it is the last CPU to clear its bit, then the end of the grace period has been reached, and RCU callbacks may safely be invoked.

The actual implementation is more heavily optimized than is described here. More details are available else-

where [ACMS03, MSA+02, McK03]. The performance benefits of RCU in the Linux kernel are also well documented [MAK+01, LSS02, MSS04, McK04], and benefits of RCU and of similar synchronization techniques in other environments have been published as well [KL80, ML84, Pug90, MS98, GKAS99, Sei03, SAH+03].

3 RCU Scheduling Latency Problem

The `amlat` test program runs a realtime task that schedules itself to run at a specific time. The `amlat` test program then measures how much the actual time is delayed from that specified.

In one test of a small configuration under heavy load, 1,660 callbacks were queued to be executed at the end of a single grace period, resulting in a scheduling latency of 711 microseconds on a single-CPU 2.4GHz x86 system. This far exceeds the goal of 250 microseconds.

The heavy load consisted of filesystem and networking operations, which resulted in large numbers of RCU callbacks being scheduled from the dcache and IP route cache subsystems.

Note that RCU callbacks are executed in the context of a tasklet, which runs either in interrupt context or in the context of the "ksoftirqd" kernel-daemon process. However, `do_softirq()`, which actually invokes the `rcu_process_callbacks()` function, uses a combination of `local_irq_save()` and `local_bh_disable()`, which has the effect of disabling preemption across the invocation of all RCU callbacks, even when running in ksoftirqd context.

Large numbers of RCU callbacks can therefore degrade realtime scheduling latency, as shown in Figure 1. In this figure, two CPUs go through a grace period while scheduling RCU callbacks. Each CPU's set of RCU callbacks is executed from `rcu_do_batch()` in softirq context after the end of the grace period, which directly increases the realtime scheduling latency, as shown in the lower right portion of the figure. This situation raises the question of what might be done to mitigate this latency increase, thereby preventing degradation of realtime response.

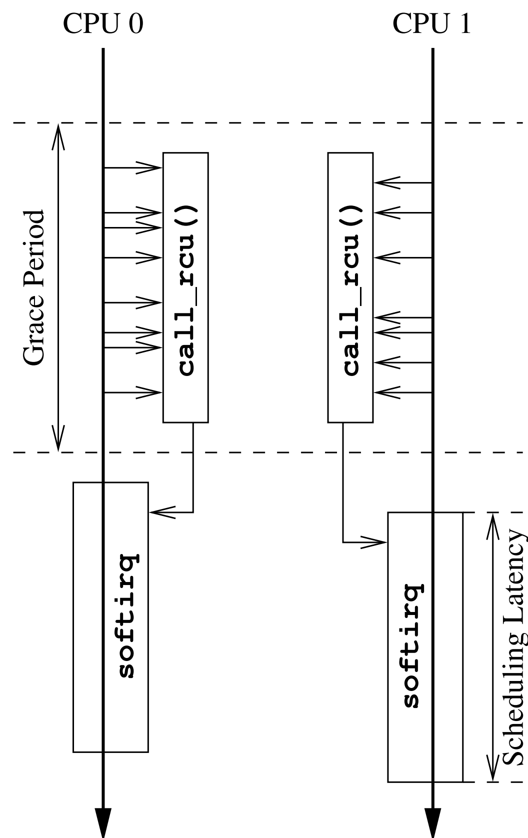

Figure 1: RCU Degrading Realtime Scheduling Latency

4 RCU Scheduling Latency Solutions

One could also imagine solving this problem by going back to traditional locking primitives, but this would impose unacceptable performance degradation and scaling limitations on Linux servers. We therefore resolved to solve the scheduling-latency problem in such a way that RCU could be used in realtime environments.

Thus far, we are investigating three solutions to this problem:

1. Providing per-CPU kernel daemons to process RCU callbacks when there are too many to process

```
1 int
2 rq_has_rt_task(int cpu)
3 {
4   struct runqueue *rq = cpu_rq(cpu);
5   return (sched_find_first_bit(rq->active->bitmap) <
6           MAX_RT_PRIO);
7 }
```

Figure 2: Functions Encapsulating Per-CPU Realtime-Task Count

at softirq level.

2. Directly invoking the RCU callback in those cases where it is safe to do so, rather than queuing the callback to be executed at the end of the next grace period.
3. Throttling RCU callback invocation so that only a limited number are invoked at a given time.

The first and last of these solutions require a mechanism that determines when there is a runnable realtime task on a given CPU. Such realtime tasks may be detected by checking a given runqueue's `active` bitmap, as was suggested by Nick Piggin, and as shown in Figure 2. The three solutions are described at length in the next sections.

4.1 Per-CPU Kernel RCU-Callback Daemons

The per-CPU kernel RCU-callback daemons [Sar04a], or *krcud* for short, were inspired by the "rcu" implementation of the RCU infrastructure in the Linux 2.6 kernel [MSA+02]. The idea is to modify `rcu_do_batch()` to limit the number of callbacks processed at a time to the value in module parameter `rcupdate.bhlimit`, which defaults to 256, but only under the following conditions:

1. the kernel has been built with the `CONFIG_LOW_LATENCY` kernel parameter,
2. there is a runnable realtime task on this CPU, and
3. `rcu_do_batch()` is running from softirq context.

If either of the first two conditions do not hold, then there is no reason to limit latency on this CPU. If the last condition does not hold, then preemption will limit execution time as needed, so no explicit limit checking is required.

When limiting does occur in `rcu_do_batch()`, any excess callbacks are queued for processing by the CPU's *krcud* on that CPU's `rcudlist` CPU-local variable. These callbacks are added to the *head* of this list in order to avoid any possibility of callback starvation. Note that callbacks can be processed out of order when limiting is in effect, since `rcu_do_batch()` can be invoked from the softirq context at the end of a grace period, even when *krcud* is running. We do not know any situation where such reordering is harmful, but strict ordering can

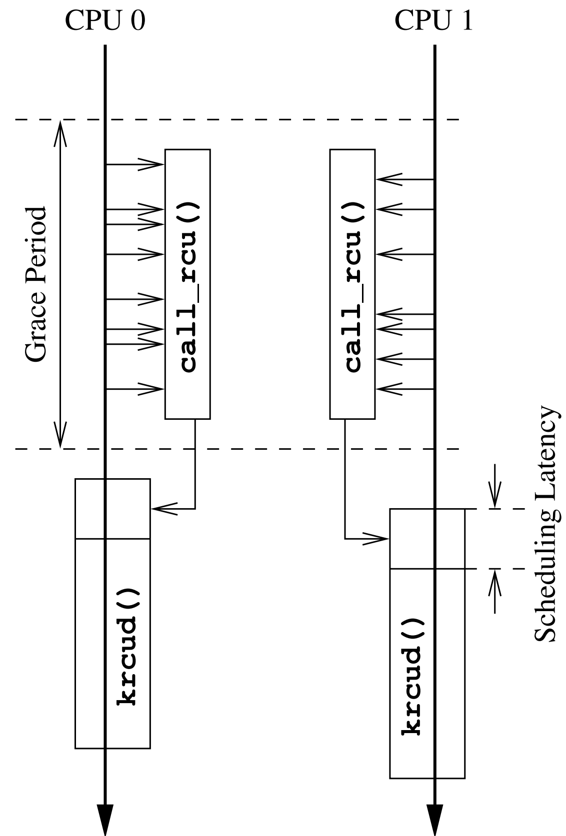

Figure 3: krcud Preserves Realtime Scheduling Latency

be easily enforced should such a situation arise.

Since *krcud* is fully preemptible, the situation is as shown in Figure 3. The first few RCU callbacks are invoked from softirq context, which cannot be preempted. The execution time of these few RCU callbacks thus degrade realtime scheduling latency, but only slightly, as any additional RCU callbacks are invoked from *krcud* context, which is fully preemptible.

The `rcu_do_batch()` function, which invokes RCU callback, but limits the callback batch size when run from softirq context, is shown in Figure 4. Line 6 captures the current CPU. Note that (for once) `get_cpu()` is not needed:

1. If invoked from `krcud()`, execution is forced to remain on a single CPU via a `set_cpus_allowed()` call.
2. If invoked from `rcu_process_callbacks()`, preemption is disabled due to running in softirq context.

Line 7 invokes `rcu_bh_callback_limit()` in order to determine the maximum number of callbacks that may be executed, which is `bhlimit` if we are running in softirq context and there is a runnable realtime

```
 1 static void rcu_do_batch(struct list_head *list)
 2 {
 3   struct list_head *entry;
 4   struct rcu_head *head;
 5   unsigned int count = 0;
 6   int cpu = smp_processor_id();
 7   unsigned int limit = rcu_bh_callback_limit(cpu);
 8
 9   while (!list_empty(list)) {
10     entry = list->next;
11     list_del(entry);
12     head = list_entry(entry, struct rcu_head, list);
13     head->func(head->arg);
14     if (++count > limit && rq_has_rt_task(cpu)) {
15       list_splice(list, &RCU_rcudlist(cpu));
16       wake_up_process(RCU_krcud(cpu));
17       break;
18     }
19   }
20 }
```

Figure 4: Limiting RCU Callback Batches

```
 1 static inline unsigned int
 2 rcu_bh_callback_limit(int cpu)
 3 {
 4   if (in_softirq() && RCU_krcud(cpu))
 5     return bhlimit;
 6   return (unsigned int)-1;
 7 }
```

Figure 5: Determining Maximum RCU Callback Batch Size

task on this CPU, or a very large integer otherwise, as can be seen in Figure 5. Lines 9-19 look over the callbacks, invoking each in turn until either the list is empty or the maximum allowable number has been exceeded. Lines 10 and 11 remove the first element from the list, and line 12 obtains a pointer to the `struct rcu_head`. Line 13 then invokes the RCU callback. Lines 14-18 check for exceeding the limit, but only if there is a runnable realtime task on this CPU. If there is, line 15 prepends the remainder of the list to this CPU's list of callbacks that are waiting for `krcud()`, line 16 wakes up this CPU's `krcud()`, and line 17 exits the "while" loop.

A given CPU's krcud task is created when that CPU is first brought online by `rcu_cpu_notify`, as shown in Figure 6. CPUs are brought up in two stages, the `CPU_UP_PREPARE` stage and the `CPU_ONLINE` stage. The `CPU_UP_PREPARE` stage is handled by lines 7-9, which invoke `rcu_online_cpu()`, which in turn initializes the RCU per-CPU data and initializes the per-CPU tasklet that processes callbacks when limiting is not in effect. At boot time, `rcu_online_cpu()` is instead called from `rcu_init()`. The `CPU_ONLINE` stage is handled by lines 10-13, which invoke `start_krcud()`, which starts the krcud tasks if appropriate. At boot time, `start_krcud()` is instead called from `rcu_late_init()`, which is registered for execution via `__initcall()`. Failure to start the

```
 1 static int __devinit
 2 rcu_cpu_notify(struct notifier_block *self,
 3                unsigned long action, void *hcpu)
 4 {
 5   long cpu = (long)hcpu;
 6   switch (action) {
 7   case CPU_UP_PREPARE:
 8     rcu_online_cpu(cpu);
 9     break;
10   case CPU_ONLINE:
11     if (start_krcud(cpu) != 0)
12       return NOTIFY_BAD;
13     break;
14   /* Space reserved for CPU_OFFLINE :) */
15   default:
16     break;
17   }
18   return NOTIFY_OK;
19 }
```

Figure 6: Creating krcud Tasks: `rcu_cpu_notify`

```
 1 static int start_krcud(int cpu)
 2 {
 3   if (bhlimit) {
 4     if (kernel_thread(krcud, (void *)(long)cpu,
 5                       CLONE_KERNEL) < 0) {
 6       printk("krcud for %i failed\n", cpu);
 7       return -1;
 8     }
 9
10     while (!RCU_krcud(cpu))
11       yield();
12   }
13   return 0;
14 }
```

Figure 7: Creating krcud Tasks

krcud task results in failure to start the CPU.

The `start_krcud()` function starts a krcud task for a specified CPU, and is shown in Figure 7. If the module parameter `bhlimit` is non-zero, the kernel thread is created by lines 4-8. Lines 10-12 then wait until the newly created krcud has initialized itself and is ready to accept callbacks. This function returns 0 on success and -1 on failure.

The `krcud()` function processes callbacks whose execution has been deferred, and is shown in Figure 8. Unlike the tasklets used by the 2.6 RCU infrastructure, `krcud()` invokes the RCU callbacks preemptibly, so that RCU callback execution from `krcud()` cannot degrade realtime scheduling latency. Note that each `krcud()` runs only on its own CPU, so that RCU callbacks are guaranteed never to be switched from one CPU to another while executing.

Line 3 of `krcud()` casts the argument, and line 4 converts this task to a daemon, setting the name, discarding any user-mode address space, blocking signals, closing open files, and setting the init task to be the newly created task's parent. Line 5 sets the krcud task's priority to the highest non-realtime priority. Line 6 marks the krcud task as required for swap operations, and line 8 restricts the task to run only on the specified CPU. Line 10

```
 1 static int krcud(void * __bind_cpu)
 2 {
 3   int cpu = (int) (long) __bind_cpu;
 4   daemonize("krcud/%d", cpu);
 5   set_user_nice(current, -19);
 6   current->flags |= PF_IOTHREAD;
 7   /* Migrate to the right CPU */
 8   set_cpus_allowed(current, cpumask_of_cpu(cpu));
 9   BUG_ON(smp_processor_id() != cpu);
10   __set_current_state(TASK_INTERRUPTIBLE);
11   mb();
12   RCU_krcud(cpu) = current;
13   for (;;) {
14     LIST_HEAD(list);
15     if (list_empty(&RCU_rcudlist(cpu)))
16       schedule();
17     __set_current_state(TASK_RUNNING);
18     local_bh_disable();
19     while (!list_empty(&RCU_rcudlist(cpu))) {
20       list_splice(&RCU_rcudlist(cpu), &list);
21       INIT_LIST_HEAD(&RCU_rcudlist(cpu));
22       local_bh_enable();
23       rcu_do_batch(&list);
24       cond_resched();
25       local_bh_disable();
26     }
27     local_bh_enable();
28     __set_current_state(TASK_INTERRUPTIBLE);
29   }
30 }
```

Figure 8: krcud Function

marks the task as alive, line 11 executes a memory barrier to prevent misordering, and line 12 sets the CPU's `krcud` per-CPU variable to reference this krcud task. Lines 13-29 loop processing any RCU callbacks placed on the `rcudlist`. Lines 15-16 wait for RCU callbacks to appear on this list, and line 17 sets the task state to running. Line 18 masks interrupts (which are restored by line 27), and lines 19-26 loop processing the callbacks on this CPU's `rcudlist`. Lines 20-21 move the contents of this CPU's `rcudlist` onto the local `list` variable, at which point it is safe for line 22 to re-enable interrupts. Line 23 invokes `rcu_do_batch()` to invoke the callbacks, and, since we are calling it from krcud context, it will unconditionally invoke all of them, relying on preemption to prevent undue delay of realtime tasks. Line 24 yields the CPU, but only if there is some other more deserving task, as would be the case after timeslice expiration. Line 25 then disables interrupts, setting up for the next pass through the "while" loop. As noted earlier, line 27 re-enables interrupts. Line 28 sets up to block on the next pass through the "for" loop.

This approach limits the number of callbacks that may be executed by `rcu_do_batch()` from softirq context. The duration of a grace period protects against too-frequent invocations of `rcu_do_batch()`, which could otherwise result in an aggregate degradation of realtime response. Since `krcud()` runs with preemption enabled, it cannot cause excessive realtime response degradation, and, in addition, can handle any RCU callback load up to the full capacity of the CPU.

```
 1 #if !defined(CONFIG_SMP) && defined(CONFIG_LOW_LATENCY)
 2 static inline void
 3 call_rcu_rt(struct rcu_head *head,
 4             void (*func)(void *arg), void *arg)
 5 {
 6   func(arg);
 7 }
 8 #else
 9 static inline void
10 call_rcu_rt(struct rcu_head *head,
11             void (*func)(void *arg), void *arg)
12 {
13   call_rcu(head, func, arg);
14 }
15 #endif
```

Figure 9: Uniprocessor Call-Through RCU

Further refinements under consideration include:

1. Use elapsed time rather than numbers of callbacks to enforce the limiting in `rcu_do_batch()`.
2. Dynamically varying the number of callbacks to be executed based on workload or other measurement.

4.2 Direct Invocation of RCU Callbacks

Traditionally, most realtime and embedded systems have had but a single CPU. Single-CPU systems can in some cases short-circuit some of the RCU processing in some cases.

For example, if an element has just been removed from an RCU-protected data structure, and if there are no references to this element anywhere in the call stack, the element may safely be freed, since there is no other CPU that can be holding any additional references. However, it is not always possible to determine whether the call stack is free of references. For example, interrupt handlers can interrupt any function that runs without masking interrupts. Furthermore, many functions are invoked via function pointers or APIs that might be used anywhere in the kernel.

Therefore, direct invocation of RCU callbacks cannot be applied in all cases. Each use of RCU must be inspected to determine whether or not that particular use qualifies for direct invocation. However, it turns out that the important cases of dcache and of the IP route cache do qualify. When running on a uniprocessor, these two subsystems can simply immediately execute the RCU callback, so that there is no "pileup" of RCU callbacks at the end of the grace period.

Figure 9 shows how a `call_rcu_rt()` primitive may be defined, which immediately invokes the RCU callback in a realtime uniprocessor kernel, but invokes `call_rcu()` otherwise [Sar03]. The new `call_rcu_rt()` API prevents existing `call_rcu()` users from breaking, while allowing specific subsystems to use RCU in a more realtime-friendly manner.

Given this primitive, the trivial change to `d_free()` shown in Figure 10 renders the dcache subsystem

```
1 static void d_free(struct dentry *dentry)
2 {
3   if (dentry->d_op && dentry->d_op->d_release)
4     dentry->d_op->d_release(dentry);
5    call_rcu_rt(&dentry->d_rcu, d_callback, dentry);
6 }
```

Figure 10: dcache Call-Through RCU

```
1 #define rcu_read_lock_bh()      local_bh_disable()
2 #define rcu_read_unlock_bh()    local_bh_enable()
```

Figure 11: Disabling softirq Processing

realtime-friendly. The single `call_rcu()` in dcache has simply been replaced by `call_rcu_rt()`.

The changes required to the IP route cache are more complex, due to the fact that the route cache may be updated from interrupt context, but is accessed from process context. For an example of the problem that this poses, suppose that `__ip_route_output_key()` is interrupted while accessing the IP route cache in process context, and that the interrupt handler invokes softirq upon return. A softirq action might then delete the entry that `__ip_route_output_key()` is currently referencing. If the interrupt handler were to invoke `call_rcu_rt()`, then `__ip_route_output_key()` would fail upon return from interrupt.

This problem can be solved by having `__ip_route_output_key()` disable softirq (and bottom-half processing) during the traversal, similar to the manner in which preemption is already disabled. New `rcu_read_lock_bh()` and `rcu_read_unlock_bh()` primitives do just this, as shown in Figure 11. The IP route cache code (in functions `rt_cache_get_first()`, `rt_cache_get_next()`, `rt_cache_get_next()`, `rt_cache_seq_next()`, `__ip_route_output_key()`, and `ip_rt_dump()`) is then changed to use these new operations in place of `rcu_read_lock()` and `rcu_read_unlock()`.

Finally, as with dcache, the `rt_free()` and `rt_drop()` functions are changed to use `call_rcu_rt()` instead of `call_rcu()`, as shown in Figure 12.

These changes are quite straightforward, but of course this `call_rcu_rt()` approach works only on single-CPU systems. The increasing popularity of multi-threaded CPUs makes this restriction less tenable on x86 CPUs, though it would still hold on some embedded CPUs. In addition, existing and planned uses of `call_rcu()` must be carefully vetted in order to ensure that direct invocation of the RCU callback is safe. At this writing, dcache and IP route cache are the two biggest realtime offenders, and they both are amenable to use of `call_rcu_rt()`, but it is easy to imagine less

```
1 static __inline__ void rt_free(struct rtable *rt)
2 {
3   call_rcu(&rt->u.dst.rcu_head,
4            (void (*)(void *))dst_free, &rt->u.dst);
5 }
6
7 static __inline__ void rt_drop(struct rtable *rt)
8 {
9   ip_rt_put(rt);
10  call_rcu(&rt->u.dst.rcu_head,
11           (void (*)(void *))dst_free, &rt->u.dst);
12 }
```

Figure 12: Freeing and Dropping IP Route Table Entries

fortunate circumstances.

As a result, a realtime-friendly `call_rcu()` implementation would be preferable.

4.3 Throttling of RCU Callbacks

Another solution to the realtime-degradation problem is to throttle softirq, so that only a limited number of RCU callbacks may execute during a given invocation of `do_softirq()` [Sar04b]. This approach was independently suggested by Andrea Arcangeli, and is illustrated in Figure 13, where the callbacks are executed in short bursts, limiting the realtime scheduling-latency degradation.

This solution is implemented using two additional per-CPU variables, `RCU_donelist`, which is a list of RCU callbacks awaiting invocation, and `RCU_plugticks`, which counts down the number of jiffies to block RCU callback invocation. `RCU_plugticks` is decremented each scheduling clock tick on each CPU in `scheduler_tick()`. There are also two module parameters, `rcumaxbatch`, which is the maximum number of callbacks that may be executed in a single softirq invocation, and `rcuplugticks`, which is the number of jiffies to wait after exceeding the `rcumaxbatch` limit before resuming RCU callback invocation. Note that `rcuplugticks` may be set to zero, in which RCU callbacks can be run continuously, which allows easy experimentation.

This callback limiting is enforced in `rcu_do_batch()`, which is shown in Figure 14. The differences from the stock 2.6 kernel implementation are quite small. Lines 5 and 6 add `count` and `cpu` variables that count the number of RCU callbacks invoked and track the current CPU, respectively. Line 13 checks for too many RCU callback invocations and line 14 sets the per-CPU `RCU_plugticks` variable in order to prevent RCU callback invocation on this CPU for the next `rcuplugticks` jiffies. Line 15 checks to see if there is to be no such delay, and, if so, line 16 reschedules the tasklet.

The `rcu_process_callbacks()` function has

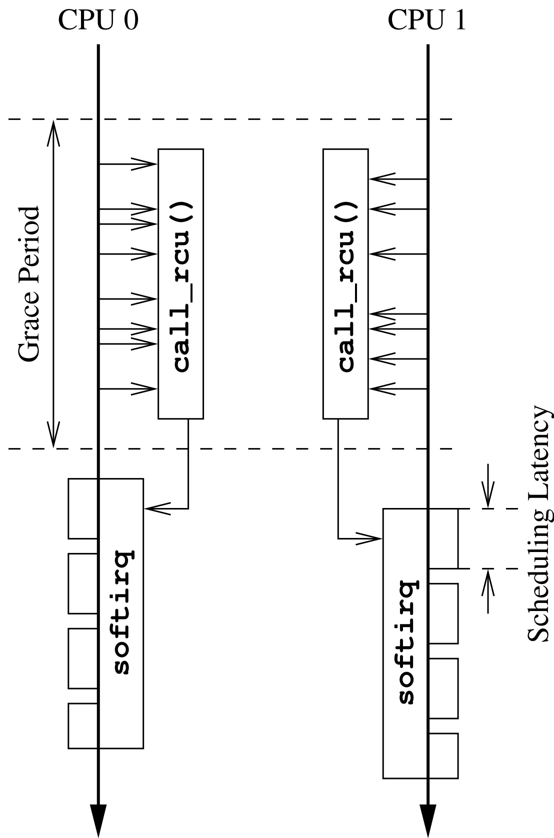

Figure 13: Throttling Preserves Realtime Scheduling Latency

```
 1 static void rcu_do_batch(struct list_head *list)
 2 {
 3   struct list_head *entry;
 4   struct rcu_head *head;
 5   int count = 0;
 6   int cpu = smp_processor_id();
 7
 8   while (!list_empty(list)) {
 9     entry = list->next;
10     list_del(entry);
11     head = list_entry(entry, struct rcu_head, list);
12     head->func(head->arg);
13     if (++count >= rcumaxbatch) {
14       RCU_plugticks(cpu) = rcuplugticks;
15       if (!RCU_plugticks(cpu))
16         tasklet_hi_schedule(&RCU_tasklet(cpu));
17       break;
18     }
19   }
20 }
```

Figure 14: Limiting RCU Callback Batch Size

```
@@ -153,18 +164,16 @@
   spin_unlock(&rcu_ctrlblk.mutex);
 }

-
 /*
  * This does the RCU processing work from
  * tasklet context.
  */
 static void
 rcu_process_callbacks(unsigned long unused)
 {
   int cpu = smp_processor_id();
-  LIST_HEAD(list);

   if (!list_empty(&RCU_curlist(cpu)) &&
       rcu_batch_after(rcu_ctrlblk.curbatch,
                       RCU_batch(cpu))) {
-    list_splice(&RCU_curlist(cpu), &list);
+    list_splice_tail(&RCU_curlist(cpu),
+                     &RCU_donelist(cpu));
     INIT_LIST_HEAD(&RCU_curlist(cpu));
   }

@@ -185,8 +194,8 @@
     local_irq_enable();
   }
   rcu_check_quiescent_state();
-  if (!list_empty(&list))
-    rcu_do_batch(&list);
+  if (!list_empty(&RCU_donelist(cpu)) &&
+      !RCU_plugticks(cpu))
+    rcu_do_batch(&RCU_donelist(cpu));
 }

 void rcu_check_callbacks(int cpu, int user)
```

Figure 15: Callback-Processing Changes

small modifications to place RCU callbacks that are ready to be invoked onto the per-CPU `RCU_donelist` list rather than on a local list, and to check for `RCU_plugticks`. The diffs are shown in Figure 15.

This small set of changes relies on the fact that `do_softirq()` exits after `MAX_SOFTIRQ_RESTART` number of iterations. When `do_softirq()` is invoked from `ksoftirqd()`, returning to `ksoftirqd()` re-enables preemption. On the other hand, when `do_softirq()` is invoked from interrupt context, returning to interrupt context in turn results in exiting interrupt context. Either alternative prevents `rcu_do_batch()` from excessively degrading realtime response.

5 Evaluation

These three approaches were tested on a uniprocessor 2.4GHz P4 system with 256MB of RAM running `dbench 32` in a loop. The kernel was built with `CONFIG_PREEMPT=y`, and the configuration excluded realtime-problematic modules such as VGA. Realtime scheduling latency was measured using Andrew Morton's `amlat` utility. The results are shown in Table 1. All three approaches greatly decrease realtime scheduling latency. Although direct invocation performs somewhat better than do the other two approaches, the differ-

ence is not statistically significant. Therefore, the simpler throttling approach seems preferable at present.

Although these numbers do not meet the 250-microsecond goal, they do indicate that RCU has been made safe for realtime environments. Changes to other parts of Linux will be needed in order to fully meet this goal. Such changes are likely to expose more significant performance differences between the three low-latency RCU approaches, so these tests should be re-run at that time.

Note that although the current testing techniques are not sufficient to validate the Linux 2.6 kernel for use by hard-realtime applications on which lives depend, they do demonstrate usefulness to soft realtime applications, even those requiring deep sub-millisecond realtime response.

6 Future Work

Future work includes applying realtime modifications to RCU in order to better withstand denial-of-service attacks, including taking full-network-adaptor-speed attacks while still providing good response to console input and user commands. It is likely that successfully withstanding such attacks will require additional work on the softirq layer in order to ensure that user processes are allowed to run even when the attack is sufficient to consume the entire system with softirq processing.

Of course, Linux will require more work if it is to meet more stringent realtime scheduling latencies, to say nothing of hard realtime requirements. Since some realtime applications require 10-microsecond scheduling latencies, it will be interesting to see if Linux can meet these applications' needs without sacrificing its usefulness to other workloads or its simplicity.

7 Acknowledgments

We owe thanks to Robert Love and Andrew Morton, who brought this problem to our attention. We are indebted to Andrew Morton for the `amlat` application that measures realtime scheduling latency, and to Jon Walpole and to Orran Krieger for many valuable discussions regarding RCU. We are grateful to Tom Hanrahan, Vijay Sukthankar, Daniel Frye, Jai Menon, and Juergen Deicke for their support of this effort.

8 Availability

RCU is freely available as part of the Linux 2.6 kernel from `ftp://kernel.org/pub/linux/kernel/v2.6`. The patches described in this paper are freely available from any archive of the Linux Kernel Mailing List.

Legal Statement

This work represents the view of the author and does not necessarily represent the view of IBM.
Linux is a registered trademark of Linus Torvalds.
Other company, product, and service names may be trademarks or service marks of others.

References

[ACMS03] 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)*, June 2003.

[GKAS99] 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*, New Orleans, LA, February 1999.

[KL80] H. T. Kung and Q. Lehman. Concurrent maintenance of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, September 1980.

[LSS02] Hanna Linder, Dipankar Sarma, and Maneesh Soni. Scalability of the directory entry cache. In *Ottawa Linux Symposium*, pages 289–300, June 2002.

[MAK+01] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001.

[McK03] Paul E. McKenney. Using RCU in the Linux 2.5 kernel. *Linux Journal*, 1(114):18–26, October 2003.

[McK04] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels (in preparation)*. PhD thesis, Oregon Graduate Institute of Science and Technology, 2004.

[ML84] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems*, 9(3):439–455, September 1984.

[MS98] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history

Configuration	Latency (microseconds)	Standard Deviation
Base 2.6.0	811.0	85.26
krcud	406.4	17.87
Direct Invocation	393.0	37.18
Throttle	414.8	37.83

Table 1: Realtime Scheduling Latencies

to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[MSA+02] 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.

[MSS04] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling dcache with RCU. *Linux Journal*, 1(118), January 2004.

[Pug90] William Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, College Park, Maryland, June 1990.

[SAH+03] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Robert W. Wisniewski, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.

[Sar03] Dipankar Sarma. Rcu low latency patches. Message ID: 20031222180114.GA2248@in.ibm.com, December 2003.

[Sar04a] Dipankar Sarma. [patch] rcu for low latency (experimental). `http://marc.theaimsgroup.com/?l=linux-kernel\&m=108003746402892\&w=2`, March 2004.

[Sar04b] Dipankar Sarma. Re: [patch] rcu for low latency (experimental). `http://marc.theaimsgroup.com/?l=linux-kernel\&m=108016474829546\&w=2`, March 2004.

[Sei03] Joseph W. Seigh II. Read copy update. email correspondence, March 2003.