USENIX Association

Proceedings of the FREENIX Track:
2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004
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, microseconds, 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 nonatomically 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, MSA02, McK03]. The performance benefits of RCU in the Linux kernel are also well documented [MAK01, LSS02, MSS04, McK04], and benefits of RCU and of similar synchronization techniques in other environments have been published as well [KL80, ML84, Pag90, MS98, GKAS99, Sei03, SAH03].

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.

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

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

1. 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.
2. 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 mecha-
nism that determines when there is a runnable realtime

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 mecha-
nism that determines when there is a runnable realtime task on a given CPU. Such realtime tasks may be de-
tected by checking a given runqueue’s active bitmap, as was suggested by Nick Piggin, and as shown in Fig-
ure 2. The three solutions are described at length in the

4.1 Per-CPU Kernel RCU-Callback Dae-

Figure 3: krcud Preserves Realtime Scheduling Latency

The per-CPU kernel RCU-callback daemons [Sar04a],
or krcud for short, were inspired by the “rcu” im-
plementation 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
rcud-update.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.

Since krcud is fully preemptible, the situation is as
shown in Figure 3. The first few RCU callbacks are in-
voled from softirq context, which cannot be preempted.
The execution time of these few RCU callbacks thus de-
grade 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)
set_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 or-

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 in-
voled from softirq context, which cannot be preempted.
The execution time of these few RCU callbacks thus de-
grade 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)
set_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 or-

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 in-
voled from softirq context, which cannot be preempted.
The execution time of these few RCU callbacks thus de-
grade 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)
set_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.
static void rcu_do_batch(struct list_head *list) {
    struct list_head *entry;
    struct rcu_head *head;
    unsigned int count = 0;
    int cpu = smp_processor_id();
    unsigned int limit = rcu_bh_callback_limit(cpu);
    while (!list_empty(list)) {
        entry = list->next;
        list_del(entry);
        head = list_entry(entry, struct rcu_head, list);
        head->func(head->arg);
        if (++count > limit && rq_has_rt_task(cpu)) {
            list_splice(list, &RCU_rcudlist(cpu));
            wake_up_process(RCU_krcud(cpu));
            break;
        }
    }
}

static inline unsigned int
rcu_bh_callback_limit(int cpu) {
    if (in_softirq() && RCU_krcud(cpu))
        return bhlimit;
    return (unsigned int)-1;
}

static int __devinit
rcu_cpu_notify(struct notifier_block *self,
               unsigned long action, void *hcpu) {
    long cpu = (long)hcpu;
    switch (action) {
        case CPU_UP_PREPARE:
            rcu_online_cpu(cpu);
            break;
        case CPU_ONLINE:
            if (start_krcud(cpu) != 0)
                return NOTIFY_BAD;
            break;
        /* Space reserved for CPU_OFFLINE :) */
        default:
            break;
    }
    return NOTIFY_OK;
}

static int start_krcud(int cpu) {
    if (bhlimit) {
        if (kernel_thread(krcud, (void *)(long)cpu,
                          CLONE_KERNEL) < 0) {
            printk("krcud for %i failed
", cpu);
            return -1;
        }
    }
    while (!RCU_krcud(cpu))
        yield();
    return 0;
}

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
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 real-time 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 real-time response. Since krcud() runs with preemption enabled, it cannot cause excessive real-time response degradation, and, in addition, can handle any RCU callback load up to the full capacity of the CPU.

Figure 9 shows how a call_rcu() 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() 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

```c
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 #endif
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
```

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 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() 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() 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
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_seq_next(), _ip_route_output_key(), and ip_rt_dump()) is then changed to use these new operations in place of rcu_readlock() and rcu_readunlock().

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 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 RCUplugticks, which counts down the number of jiffies to block RCU callback invocation. RCUplugticks 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 RCUplugticks 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
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   while (!list_empty(list)) {
8     entry = list->next;
9     list_del(entry);
10    head = list_entry(entry, struct rcu_head, list);
11    head->func(head->arg);
12    if (++count >= rcumaxbatch) {
13      RCU_plugticks(cpu) = rcuplugticks;
14      if (!RCU_plugticks(cpu))
15         tasklet_hi_schedule(&RCU_tasklet(cpu));
16      break;
17    }
18   }
19 }

Figure 14: Limiting RCU Callback Batch Size

Figure 13: Throttling Preserves Realtime Scheduling Latency

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


[MS98] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Latency (microseconds)</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base 2.6.0</td>
<td>811.0</td>
<td>85.26</td>
</tr>
<tr>
<td>krcud</td>
<td>406.4</td>
<td>17.87</td>
</tr>
<tr>
<td>Direct Invocation</td>
<td>393.0</td>
<td>37.18</td>
</tr>
<tr>
<td>Throttle</td>
<td>414.8</td>
<td>37.83</td>
</tr>
</tbody>
</table>

Table 1: Realtime Scheduling Latencies


