Towards Hard Realtime Response from the Linux Kernel on SMP Hardware

Paul E. McKenney
IBM Beaverton
paulmck@us.ibm.com

Dipankar Sarma
IBM India Software Laboratory
dipankar@in.ibm.com

Abstract

Not that many years ago, many people doubted that a single operating system could support both desktop and server workloads. Linux is well on its way to proving these people wrong.

But new times bring new challenges, and so there are now many people who doubt that a single operating system can support both both general-purpose and realtime workloads. This question is still open, although Linux has been used for a surprising number of realtime applications for quite some time. This paper looks at a few mechanisms that have been proposed to push Linux further into the realtime arena, with an eye towards maintaining Linux's high-end performance and scalability.

In other words, can Linux offer realtime support even when running on SMP hardware?

1 Introduction

Traditionally, hard realtime response has been designed into operating systems offering it. Retrofitting hard realtime response into an existing general-purpose OS may not be impossible, but is quite difficult: all non-preemptive code paths in the OS must have deterministic execution time. Since this requires a full rewrite of the OS, the traditional approach has been to simply create a new OS specially designed to offer hard-realtime response. There have also been the first stirrings of desire for multiprocessor systems with hard realtime response.

Although the Linux™ 2.6 kernel offers much improved realtime response, performance, and scalability when compared to the 2.4 kernel, it does not offer hard realtime response, nor has it been able to provide scheduling latencies below about 100 microseconds. However, this is changing with the advent of two new approaches.

The first approach, lead up by Ingo Molnar, is to aggressively reduce the amount of time that the kernel is non-preemptible. This effort has been quite successful, with average scheduling latencies reported to be as low as one microsecond on single-CPU systems. One of the challenges met in this effort was that of creating a specialized RCU implementation that met these scheduling latencies. However, as we will see, although this approach preserves most RCU semantics, it does so in a way that does not scale well on multiprocessor systems.

The second approach is an extension of the interrupt-shielding techniques used by Dipankar Sarma, Dinakar Guniguntala, and Jim Houston. Their approach directs interrupts away from a particular CPU of an SMP system, thereby reducing the scheduling latency on that particular CPU. The advent of low-cost hardware multithreaded and multi-core CPUs makes this approach quite attractive for some applications, as it provides excellent realtime latency for CPU-bound user-mode applications.

This paper looks at the problem of producing a kernel that both scales and supports microsecond scheduling latencies. Section 2 reports on attempts, unsuccessful thus far, to make an RCU implementation that can run in the realtime-preempt environment but with excellent performance and scalability on SMP hardware. Section 3 gives a brief overview of an alternative approach that potentially offers both hard realtime and good scalability. Finally Section 4 presents concluding remarks.

2 Realtime-Preempt Approach

Ingo Molnar has implemented a fully preemptible approach to extreme realtime [8]. In this approach, almost all spinlocks are mapped into blocking locks, and fine-grained scheduling-latency measurement tools are used to identify and recode kernel codepaths with excessive scheduling latency. RCU read-side critical sections are also mapped into blocking locks, requiring additional APIs, as summarized in Table 1.

RCU requires special attention, because: (1) RCU callbacks execute at softirq level, and can impose unacceptable scheduling delay [10], and (2) RCU read-side critical sections must suppress preemption in preemptible kernels. Ingo Molnar transformed RCU to a locking protocol, so that the RCU read-side critical sections acquire a sleeplock, the same sleeplock used by updaters. When the lock is a reader-writer lock, readers can recursively acquire it (see Figure 1), as needed to accommodate nested RCU read-side critical sections, but only a single task may read-hold such a lock. However, writers cannot recursively acquire this lock, thus losing RCU's ability to unconditionally upgrade from an RCU read-side critical section to a write-side critical section. On the brighter side, this implementation permits `call_rcu()` to immediately invoke RCU callbacks, since the lock prevents any readers from executing concurrently with a writer. This immediate invocation prevents memory-consumption problems that could otherwise occur in workloads that generate large numbers of RCU callbacks.

This approach works well, but it does lead to the question of whether there is a way to accommodate the realtime requirements while preserving more of RCU's semantics, performance, and scalability. The next section summarizes RCU's properties from a realtime perspective.

2.1 Summary of Desirable RCU Properties

The desired properties of RCU are as follows:

- Deferred destruction. No data element may be destroyed (for example, freed) while an RCU read-side critical section is referencing it. This property absolutely must be implemented, as failing to do so will break code that uses RCU.

- Reliable. The implementation must not be prone to gratuitous failure; it must be able to run 24x7 for months at a time, and preferably for years.

- Callable from IRQ. Since RCU is used from softirq state, a realtime implementation must either eliminate softirq and interrupt states, eliminate use of RCU from these states, or make RCU callable from these states.

- Preemptible read side. RCU read-side critical sections can be quite large, degrading realtime scheduling response. Preemptible RCU read-side critical sections avoid such degradation.

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

- 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 and can impose increased complexity.

Name	Description
`rcu_read_lock_spin(lock)`	Begin RCU read-side critical section corresponding to a spinlock-guarded write-side critical section. Nesting is only permitted for different locks.
`rcu_read_unlock_spin(lock)`	End RCU read-side critical section corresponding to a spinlock-guarded write-side critical section.
`rcu_read_lock_read(lock)`	Begin RCU read-side critical section corresponding to a reader-writer-spinlock-guarded write-side critical section. Nesting is permitted.
`rcu_read_unlock_read(lock)`	End RCU read-side critical section corresponding to a reader-writer-spinlock-guarded write-side critical section.
`rcu_read_lock_bh_read(lock)`	Begin RCU read-side critical section corresponding to a reader-writer-spinlock-guarded write-side critical section, blocking bottom-half execution. Nesting is permitted.
`rcu_read_unlock_bh_read(lock)`	End RCU read-side critical section corresponding to a reader-writer-spinlock-guarded write-side critical section.
`rcu_read_lock_down_read(rwsem)`	Begin RCU read-side critical section corresponding to a reader-writer-semaphore-guarded write-side critical section. Nesting is permitted.
`rcu_read_unlock_up_read(rwsem)`	End RCU read-side critical section corresponding to a reader-writer-semaphore-guarded write-side critical section.
`rcu_read_lock_nort()`	Ignore RCU read-side critical section in realtime-preempt kernels.
`rcu_read_unlock_nort()`	Ignore RCU read-side critical section in realtime-preempt kernels.

Table 1: Realtime-Preempt RCU API

- Synchronization-free read side. RCU read-side critical sections should avoid expensive synchronization instructions or cache misses. It is most important to avoid global locks and atomic instructions that modify global memory, as these operations can inflict severe performance and scalability bottlenecks. Avoiding memory barriers is desirable but not as critical.

- Freely nestable read-side critical sections. Restrictions on nestability can result in code duplication, so should be avoided if possible.

- Unconditional read-to-write upgrade. RCU permits a read-side critical section to acquire the corresponding write-side lock. This capability can be convenient in some cases. However, since it is rarely used, it cannot be considered mandatory.

- Compatible API. A realtime RCU implementation should have an API compatible with that in the kernel.

The realtime-preemptible RCU implementation does defer destruction, but does not provide a synchronization-free read side. It also restricts read-side nesting, for example, the code fragments shown in Figure 2 are legal in classic RCU, but result in deadlock in the realtime-preemptible implementation. It prohibits read-to-write upgrade, resulting in some code duplication, for example, in the System V IPC code. It also changes the API, but arguably in a good way, documenting the locks corresponding to

```
static void __sched
down_read_mutex(struct rw_semaphore *rwsem, unsigned long eip)
{
  /*
   * Read locks within the write lock succeed.
   */
  if (rwsem->lock.owner == current) {
    rwsem->read_depth++;
    return;
  }
  SAVE_BKL(__down_mutex(&rwsem->lock, eip));
}
```

Figure 1: Recursive Read Acquisition

the RCU read-side critical sections.

```
void foo()
{
rcu_read_lock_read(lockA);
...
rcu_read_lock_read(lockB);
}

void bar()
{
rcu_read_lock_read(lockB);
...
rcu_read_lock_read(lockA);
}
```

Figure 2: ABA Deadlock Scenario

It would be quite useful to have a realtime-friendly RCU implementation with synchronization-free read-side critical sections, as this would give larger multiprocessor Linux systems these realtime capabilities. It is well known that use of global locks for the RCU read-side critical sections results in high contention on moderate-sized SMP systems [1, 9]. The remainder of this section examines various alternatives in an attempt to come up with an approach that both scales *and* offers realtime response. Those wishing to skip straight to the scorecard should refer to Table 2 in Section 2.16.

2.2 Classic RCU

This section reviews slight variations on the classic Linux 2.6 RCU implementation, first described elsewhere [10]:

1. Run all RCU callbacks from preemptible kernel daemons.
2. Invoke RCU callbacks directly from `call_rcu()` on uniprocessor systems, in situations where this can be done safely.
3. Process only a few RCU callbacks at a time at the softirq level.

Although these approaches greatly reduce or eliminate the realtime scheduling-latency degradation due to invoking large numbers of RCU callbacks, they do nothing about similar degradation caused by overly long RCU read-side critical sections. Given that there have been RCU patches that walk the entire task list in a single RCU read-side critical section, this degradation can be severe indeed.

In addition, the amount of memory consumed by RCU callbacks waiting for a grace period to expire, though negligible on large servers, can be a serious problem for embedded systems.

The classic RCU implementation therefore seems inappropriate for realtime-preemptible kernels.

2.3 Preemptible RCU Read-Side Critical Sections

Perhaps the simplest solution is to make only voluntary context switch be a quiescent state. This has been shown to work [2, 6], and does permit preemptible RCU read-side critical sections, but is prone to indefinite grace periods if a given low-priority task remains preempted indefinitely. This approach is therefore absolutely unacceptable for production use in the Linux kernel, particularly in restricted-memory environments.

2.4 Jim Houston Patch

Jim Houston submitted a patch to RCU that enables all interrupts to be directed away from a given CPU [4]. The changes are needed because the mainline RCU implementation critically depends on the scheduling-clock interrupt to detect grace periods and advance RCU callbacks.

This patch uses a per-CPU flag that indicates whether the corresponding CPU is in an RCU read-side critical section. This flag is set by `rcu_read_lock()`, and is accompanied by a counter that allows the usual nesting of RCU read-side critical sections. Other CPUs can read this flag, and if they detect that a given CPU is holding up a grace period, they will use the `cmpxchg()` primitive that is available on some CPU types to set a `DO_RCU_COMPLETION` flag.

When the interrupt-free CPU encounters the `rcu_read_unlock()` that completely exits the RCU read-side critical section, it atomically exchanges the flag with zero using the `xchg()` primitive. If the old value of this flag indicates that a quiescent state is required, it invokes `rcu_quiescent()` for this purpose, advancing RCU callbacks and invoking any that have persisted through a full grace period.

This approach provides almost all of RCU's desirable features. Unfortunately, it does not permit RCU read-side critical sections to be preempted, nor does it provide synchronization-free RCU read-side critical sections. In particular, `rcu_read_unlock()` does atomic instructions on a flags variable that is also atomically manipulated by other CPUs, which is quite expensive on large SMP systems.

2.5 Reader-Writer Locking

Since reader-writer locking is in some ways similar to RCU, it is a tempting alternative. And in fact reader-writer locks are used in the realtime preemptible RCU implementation.

However, reader-writer locks inflict substantial overhead on readers, prevent read-side critical sections from being freely nested, and can in no way support unconditional upgrade from reader to writer critical sections. This latter issue can be subtle, and the key to understanding it is to consider cases where several readers on different CPUs simultaneously attempt an upgrade. Only one of them can succeed; the rest must fail, perhaps silently. The only other alternative is deadlock.

This approach also requires a change to the classic RCU API, but this is arguably an improvement, since read-side critical sections are flagged with the corresponding write-side lock. Experience indicates that there will be situations where there are either many corresponding write-side locks or none at all, but these situations can be addressed as they arise.

2.6 Hazard Pointers

The idea behind hazard pointers is to maintain explicit per-CPU (or per-task) pointers to the data elements that are being referenced by the RCU read-side critical section [3, 7]. This approach introduces some complexity because the hazard pointers must be removed some time after the critical section drops its reference to the corresponding data element.

It is fairly easy to transform the RCU APIs to use hazard pointers:

- The `rcu_read_lock()` primitive must increment a per-task counter. Interrupt handlers would presumably use their CPU's idle task.

- The `rcu_dereference()` primitive assigns a hazard pointer to guard the pointer passed to it. The task maintains multiple lists of such hazard pointers, indexed by the per-task counter that is incremented by `rcu_read_lock()`. Memory barriers are required so that the update side sees the hazard pointer *before* the reference is actually acquired.

- The `rcu_read_unlock()` primitive must execute a memory barrier, free up the current group of hazard pointers, then decrement the per-task counter. Because of the grouping, freeing up the hazard pointers can be a simple linked-list operation.

- The `call_rcu()` primitive must check to see if the pointer is covered by any task's hazard pointers, and must defer the callback if so. Otherwise, the callback may be invoked immediately. Note that it is necessary to detect a hazard pointer referencing *any* part of the data structure. This is important because the hazard pointer might have been placed on a `struct list_head` located in the middle of the data structure, and `call_rcu()` will be given a reference to some `struct rcu_head` that is located elsewhere in this same data structure.

  In the general case, which might involve structures containing variable-length arrays of other structures, an assist from the memory allocator will be needed. Alternatively, if data structures are appropriately aligned, the hazard pointers can reference a given fixed block of memory rather than a given data structure.

One problem is the large amount of storage required in the worst case: in the presence of preemption, the worst-case number of hazard pointers is unbounded. To see this, imagine a large number of tasks, all traversing a long RCU-protected list, and each being preempted just before reaching the end of the list. The number of hazard pointers required will be the number of elements times the number of tasks, which is ridiculously large.

But this is not the theoretical worst case. Suppose that a list is traversed, and at each element another function is called, which itself traverses the list. The number of hazard pointers required in this case, which is *still* not the theoretical worst case, will be the square of the number of list elements times the number of tasks. One can construct increasingly ornate (and, thankfully, increasingly unlikely) situations that consume ever-greater numbers of hazard pointers.

It is possible to use reference counters to limit the memory consumption, but this requires that a task's hazard pointer be efficiently located given the referenced data structure and the task. This is possible, but becomes considerably more complex. And the hazard pointers must grow larger to accommodate the additional state required.

Rather than following this example to the bitter end, we turn now to dynamic allocation of hazard pointers.

2.7 Hazard Pointers with Memory-Allocation Failure Indication

We could change the `rcu_dereference()` API to report memory-allocation failures to the caller. However, this would introduce significant complexity, since currently `rcu_dereference()` cannot fail. The resulting unwind code would not be pretty.

That said, the memory usage could be limited to any desired value, simply by using the appropriate failure checks.

2.8 Hazard Pointers with Memory-Allocation Failure Kernel Death

Although the theoretical number of hazard pointers is quite large, in many cases the actual number used will be quite manageable. After all, a given critical section can use only so many hazard pointers while still maintaining realtime response, right?

This approach would provide a reasonable number of hazard pointers, based on the actual number used by the desired workload. If an attempt to allocate a hazard pointer fails, just die!

There are probably situations where this could work, but this possibility of random failure will not be acceptable to all users.

2.9 Hazard Pointers with Blocking Memory Allocation

Another variation on this theme is to use blocking memory allocation. This approach normally would not be feasible, however, most spinlocks have been converted into blocking locks. Nonetheless, blocking memory allocations are not permitted from interrupt handlers or from the softirq environment, nor are they permitted while holding any of the remaining raw spinlocks. Therefore, blocking memory allocation might be used in some cases, but is not a general solution.

2.10 Reference Counters

The basic idea is seductively simple, namely, have `rcu_dereference()` increment a reference counter associated with the data element that is protected by RCU. The problem appears when it comes time to decrement this reference counter. Tracking the reference counters to be decremented consumes as much memory as do the hazard pointers themselves.

Nothing to see here, time to move along...

2.11 Explicit rcu_donereference() Primitive

One way to address the problems with hazard pointers and reference counters is to provide an explicit `rcu_donereference()` primitive. The purpose of this primitive is to signal that a given pointer, which was subject of a prior `rcu_dereference()` invocation, will no longer be dereferenced, at least pending a later `rcu_dereference()`. This primitive will need to invoke a memory barrier so that the update-side code does not see the hazard pointer disappear until the corresponding reference has been dropped.

This `rcu_donereference()` can then invalidate hazard pointers or decrement reference counts. Given properly placed `rcu_donereference()` calls, cleaning up hazard pointers and reference counts is trivial. Furthermore, many fewer hazard pointers are required. A linked list may be traversed with but a single hazard pointer, pointing to each element in turn. In contrast, the earlier approaches required one hazard pointer per element in the list.

Like all the other hazard-pointer based approaches, there must be a way to identify the size and extent of each RCU-protected block of memory, since a given read-side critical section might be traversing one of a number of lists contained in that element.

Unfortunately, nontrivial changes are required to add the needed `rcu_donereference()` calls. One might hope to bury such calls into the `list_for_each.*rcu()` primitives, as shown by the (broken!) code in Figure 3. Of course, for this to work, the `rcu_dereference()` primitive needs to politely ignore a NULL pointer. But even with this concession, this code is unsafe. To see this, consider the code fragment from `net/bridge/br_forward.c` shown in Figure 4.

This is a perfectly sane piece of code, which leaves the pointer prev assigned to the last member of the list for which `should_deliver()` succeeds, or NULL if there is no such member. Unfortunately, the preceding implementation of `list_for_each_entry_rcu()` would have released the hazard pointers (or reference counts) that were protecting the memory pointed to by `prev` on the next pass through the loop.

This can be manually fixed as shown in Figure 5. Here, the explicit `rcu_dereference()` compensates for the `rcu_donereference()` contained in the `list_for_each_entry_rcu()`.

And these are not the only changes required. It is also necessary to track down all the places where the reference to `prev` is dropped and to add an `rcu_donereference()` there as well. This is not too difficult in this particular case, since only one more `rcu_donereference()` is required, and it is only a few lines after the code fragment shown in Figure 5. However, it is not hard to imagine cases where the `prev` pointer might travel far and wide throughout the kernel. Here are some rules for where the `rcu_donereference()` primitives should be placed:

1. When the pointer goes out of scope.

2. When the pointer is overwritten.

3. When an `rcu_read_unlock()` is encountered

```
#define list_for_each_entry_rcu(pos, head, member)                 \
  for (pos = list_entry((head)->next, typeof(*pos), member); \
       rcu_donereference(pos),                               \
       prefetch(pos->member.next), &pos->member != (head);   \
       pos = rcu_dereference(list_entry(pos->member.next,    \
                             typeof(*pos), member)))
```

Figure 3: Broken Hazard-Pointer List Scan

```
prev = NULL;
list_for_each_entry_rcu(p, &br->port_list, list) {
  if (should_deliver(p, skb)) {
    if (prev != NULL) {
      struct sk_buff *skb2;

      if ((skb2 = skb_clone(skb, GFP_ATOMIC)) == NULL) {
        br->statistics.tx_dropped++;
        kfree_skb(skb);
        return;
      }

      __packet_hook(prev, skb2);
    }
    prev = p;
  }
}
```

Figure 4: Use of Broken Hazard-Pointer List Scan

(taking nesting into account, of course).

4. When the pointer is assigned to a globally accessible RCU-protected data structure.

Of course, if the value is assigned to a second local pointer, an additional `rcu_dereference()` / `rcu_donereference()` pair may be required.

Rule #4 is problematic. It is good practice to have a single set of functions to handle a given data structure, regardless of whether that data structure is globally accessible or local to this CPU. But with rule #4, these functions must execute an `rcu_donereference()` if the structure is global, but not if it is local. There is a name for this situation, namely "code bloat".

2.12 Lock-Based Deferred Free

RCU provides a very limited notion of mutual exclusion, namely that a reader referencing an item in an RCU read-side critical section will exclude an updater attempting to free that item. This exclusion works by deferring the updater, either by blocking a `synchronize_kernel()` inovocation until all current readers exit their RCU read-side critical sections, or by deferring invocation of a `call_rcu()` callback,again until all current readers exit their RCU read-side critical sections. RCU provides a very lightweight deferral mechanism, but this lightness is achieved by deferring much longer than is actually required, which in turn increases the number of outstanding callbacks, which can be problematic on

```
prev = NULL;
list_for_each_entry_rcu(p, &br->port_list, list) {
  if (should_deliver(p, skb)) {
    if (prev != NULL) {
      struct sk_buff *skb2;

      if ((skb2 = skb_clone(skb, GFP_ATOMIC)) == NULL) {
        br->statistics.tx_dropped++;
        kfree_skb(skb);
        return;
      }

      __packet_hook(prev, skb2);
    }
    rcu_donereference(prev);
    prev = p;
    rcu_dereference(prev);
  }
}
```

Figure 5: Manually Fixed Hazard-Pointer List Scan

small-memory configurations.

This suggests use of a heavier weight, but more immediate, exclusion strategy, and "heavy weight mutual exclusion" immediately brings to mind locking. But we need not make reads exclude updates, since any bug-free RCU code already handles concurrent reads and updates. We need only exclude execution of reads from execution of callbacks that were registered after the start of the corresponding read. It is perfectly legal to allow callbacks to run in parallel with reads that started *after* the corresponding callback was registered. Again, any bug-free RCU code already handles concurrent reads and callbacks.

The trivial implementation in the next section demonstrates these principles.

2.12.1 Trivial Implementation

This section describes a simple implementation with extremely limited scalability, whose code is displayed in Figure 6. In addition, this overly simple implementation adds some unwanted constraints to `call_rcu()` usage, since it cannot be called from:

1. an RCU read-side critical section, since this would result in deadlock, and
2. an interrupt handler, since this would again result in deadlock if an RCU read-side critical section were interrupted.

Nonetheless, its 29 lines of code, counting whitespace, serve to illustrate the principles underlying lock-based deferred free. Straightforward elaborations of this example remove these disadvantages.

Line 1 of the figure defines the reader-writer lock that is used to defer frees. Lines 3-7 show the implementation of `rcu_read_lock()`, which simply read-acquires the lock. Lines 9-13 show `rcu_read_unlock()`, which, predictably, read-releases the lock. Lines 15-20 display `synchronize_kernel()`, which write-acquires the lock then immediately releases it, which guarantees that any RCU read-side critical sections that were in progress before the call to `synchronize_kernel()` have completed. Lines 22-29 display `call_rcu()`, which simply invokes `synchronize_kernel()` and invokes the callback, which guarantees that any RCU

```
rwlock_t rcu_deferral_lock = RW_LOCK_UNLOCKED;

void
rcu_read_lock(void)
{
  read_lock(&rcu_deferral_lock);
}

void
rcu_read_unlock(void)
{
  read_unlock(&rcu_deferral_lock);
}

void
synchronize_kernel(void)
{
  write_lock(&rcu_deferral_lock);
  write_unlock(&rcu_deferral_lock);
}

void
call_rcu(struct rcu_head *head,
   void (*func)(struct rcu_head *rcu))
{
  synchronize_kernel();
  func(head);
}
```

Figure 6: Trivial Lock-Based Deferred Free

read-side critical sections that were in progress before the call to `call_rcu()` have completed before the callback is invoked.

The `rcu_dereference()` and `rcu_assign_pointer()` primitives are unchanged from their current Linux 2.6 kernel implementation, which provides memory barriers for those CPU architectures that require them.

Those who might hope that this 29-line implementation could completely replace the "classic" RCU implementation in the Linux kernel should note the following:

1. The lock-based deferred-free implementation imposes significant synchronization overhead on readers.

2. The use of reader-writer locks results in severe memory contention, even for read-only access, on SMP systems.

3. A similar "toy" implementation of "classic" RCU is only 26 lines, as shown in Figure 7.

```
#define rcu_read_lock()
#define rcu_read_unlock()

void synchronize_kernel(void)
{
  cpumask_t oldmask;
  cpumask_t curmask;
  int cpu;

  if (sched_getaffinity(0, &oldmask) < 0) {
    oldmask = cpu_possible_mask;
  }
  for_each_cpu(cpu) {
    sched_setaffinity(0, cpumask_of_cpu(cpu));
    schedule();
  }
  sched_setaffinity(0, oldmask);
}

void
call_rcu(struct rcu_head *head,
   void (*func)(struct rcu_head *rcu))
{
  synchronize_kernel();
  func(head);
}
```

Figure 7: Toy Implementation of Classic RCU

The following sections present shortcomings of the trivial implementation of the lock-based deferred-free approach, then.

2.12.2 Shortcomings of Trivial Implementation

The trivial implementation in the previous section has a number of shortcomings, including:

1. deviations from RCU semantics, since `call_rcu()` cannot be invoked from an RCU read-side critical section or from interrupt handlers;

2. memory contention on the `rcu_deferral_lock`;

3. write-side lock contention on `rcu_deferral_lock`;

4. memory-constrained environments require some way to actively reap callbacks; and

5. CPU hotplug is not taken into account.

These shortcomings are addressed in the following sections, starting with the deviations from semantics.

2.12.3 Exact RCU Semantics

The difficulty with the trivial implementation was that `call_rcu()` write-acquired the `rcu_deferral_lock`, which prohibits use of `call_rcu()` anywhere that might appear in an RCU read-side critical section, including functions called from within such critical sections, whether invoked directly or via an interrupt handler. One way to avoid this prohibition is for `call_rcu()` to place a callback on a queue, so that some other piece of code write-acquires the lock from a controlled environment. This environment might be a kernel daemon or work queue. It cannot be any sort of interrupt or softirq handler unless interrupts are disabled during RCU read-side critical sections, which would have the undesired effect of rendering these critical sections non-preemptible.

Data structures to support queuing of callbacks are shown in Figure 8. The `rcu_data` structure is replicated per-CPU. The `waitlist` and `waittail` fields hold a linked list of RCU callbacks that are waiting for a grace period to expire. The `batch` field contains the batch number that was in effect at the time that the callbacks on `waitlist` were registered. The `donelist` and `donetail` fields hold a linked list of RCU callbacks that have passed through a grace period, and are therefore waiting to be invoked.

The `rcu_ctrlblk` structure is global, and contains a reader-writer lock that operates similarly to `rcu_deferral_lock`. The `batch` field counts the number of batches of RCU callbacks that have been processed.

The RCU read-side critical sections work in the same manner as they did in the trivial implementation, as shown in Figure 9.

The update side, shown in Figure 10, is where most of the changes occur. The `synchronize_kernel()` primitive (lines 1-7) is very similar, adding only the counter increment. However, the implementation of `call_rcu()` changes completely, as shown in lines 9-24. Lines 16-17 initialize the `rcu_head` fields. Line 18 disables interrupts to avoid races between invocations from process and irq contexts. Line 19 invokes `rcu_do_my_batch()` process any of this CPU's callbacks whose grace period has already expired.

```
struct rcu_data {
  long    batch;
  struct rcu_head  *waitlist;
  struct rcu_head  **waittail;
  struct rcu_head  *donelist;
  struct rcu_head  **donetail;
};
struct rcu_ctrlblk {
  rwlock    lock;
  long    batch;
}
DECLARE_PER_CPU(struct rcu_data, rcu_data);
struct rcu_ctrlblk rcu_ctrlblk = {
  .lock = RW_LOCK_UNLOCKED,
  .batch = 0,
};
```

Figure 8: LBDF, Exact Semantics (Data)

```
void
rcu_read_lock(void)
{
  read_lock(&rcu_ctrlblk.lock);
}

void
rcu_read_unlock(void)
{
  read_unlock(&rcu_ctrlblk.lock);
}
```

Figure 9: LBDF, Exact Semantics (Read Side)

Lines 20-22 enqueue this RCU callback, and line 23 restores interrupts.

Figure 11 shows how callbacks are processed. This code might be called from a kernel daemon, from a work queue, or from an out-of-memory handler (which should first invoke `synchronize_kernel()`, of course!). Line 7 disables interrupts (redundant when called from `call_rcu()`). Line 9 is a memory barrier to prevent the CPU from reordering the test of the batch numbers before any prior removal of an element from an RCU-protected list. If the test on line 10 indicates that a grace period has expired, then lines 11-14 move any callbacks in `waitlist` to the end of `donelist`, and line 15 snapshots the new batch number. Line 18 checks to see if there are any callbacks waiting to be invoked, and, if so, lines 19-20 removes them from `donelist`, line 20 restores interrupts (thus permitting callback invocation to be preempted, at least when not called from `call_rcu()`, and the while loop from lines 22-26 processes the call-

```
 1 void
 2 synchronize_kernel(void)
 3 {
 4   write_lock_bh(&rcu_ctrlblk.lock);
 5   rcu_ctrlblk.batch++;
 6   write_unlock_bh(&rcu_ctrlblk.lock);
 7 }
 8
 9 void
10 call_rcu(struct rcu_head *head,
11    void (*func)(struct rcu_head *rcu))
12 {
13   unsigned long flags;
14   struct rcu_data *rdp;
15
16   head->func = func;
17   head->next = NULL;
18   local_irq_save(flags);
19   rcu_do_my_batch();
20   rdp = &__get_cpu_var(rcu_data);
21   *rdp->waittail = head;
22   rdp->nexttail = &head->next;
23   local_irq_restore(flags);
24 }
```

Figure 10: LBDF, Exact Semantics (Update Side)

```
 1 void rcu_do_my_batch(void)
 2 {
 3   unsigned long flags;
 4   struct rcu_data *rdp;
 5   struct rcu_head *next, *list;
 6
 7   local_irq_save(flags);
 8   rdp = &__get_cpu_var(rcu_data);
 9   smp_mb();
10   if (rdp->batch != rcu_ctrlblk.batch) {
11     *rdp->donetail = rdp->waitlist;
12     rdp->donetail = rdp->waittail;
13     rdp->waitlist = NULL;
14     rdp->waittail = &rdp->waitlist;
15     rdp->batch = rcu_ctrlblk.batch;
16   }
17   list = rdp->donelist;
18   if (list != NULL) {
19     rdp->donelist = NULL;
20     rdp->donetail = &rdp->waitlist;
21     local_irq_restore(flags);
22     while (list) {
23       next = list->next;
24       list->func(list);
25       list = next;
26     }
27   } else {
28       local_irq_restore(flags);
29   }
30 }
```

Figure 11: LBDF, Exact Semantics (Callback Processing)

backs. If there are no callbacks waiting to be invoked, line 28 restores interrupts.

This approach is far from perfect:

- Since the CONFIG_PREEMPT_RT kernel only permits one thread at at time to read-acquire a lock, this approach is subject to read-side contention, which can result in excessive scheduling latency.
- Scheduling latency can "bleed" over from one reader to another in the case where synchronize_kernel() is waiting for one reader to finish, and another reader is waiting for synchronize_kernel().
- There are a number of implementation details that cause various difficulties in the CONFIG_PREEMPT_RT environment. These have since been resolved [5].

2.13 Read-Side Grace-Period Suppression

The initial difficulties with lock-based deferred free in the CONFIG_PREEMPT_RT environment motivated Ingo Molnar to make a small change to Classic RCU, based on a suggestion by Esben Neilsen. This change suppresses a given CPU's quiescent states for the duration of any RCU read-side critical sections that are initiated on that CPU. This greatly reduces read-side overhead, but can result in indefinite-duration grace periods. Nonetheless, it suffices for experimentation, and is currently part of the CONFIG_PREEMPT_RT environment.

2.14 Read-Side Counters With Flipping

The K42 and Tornado implementations of an RCU-like mechanism called "generations" use two sets of counters to track RCU read-side critical sections. A similar approach was used in some early Linux patches implementing RCU. The idea is that one counter of the pair is incremented any time a task enters a RCU read-side critical section. When that task exits its critical section, it decrements that same counter. The roles of the counters in a pair can be flipped, which allows a grace period to end a soon as the other counter drops to zero, despite the possibility of subsequent entries into RCU read-side critical sections.

Design of this approach is ongoing.

2.15 Discussion

The lock-based deferred-free approach has all of the desireable properties called out in Section 2.1, with the exception of a synchronization-free read side. However, this read-side synchronization is light weight, since only per-CPU locks are acquired, thus avoiding the much of the cache-miss overhead that normally comes with lock-based synchronization.

2.16 Summary of Realtime-Preempt Approaches

Each of the previously discussed methods of permiting RCU to scale in a realtime-preempt kernel has its drawbacks, as shown in Table 2. Each column corresponds to one of the items in the list of desirable RCU properties in Section 2.1, with the exception of "deferred destruction", which is met by all of the proposed approaches. Blank cells are goodness, cells with "n" are mild badness, with "N" are major badness, and with "X" are grounds for instant disqualification. Note that the API change required for the "Reader-Writer Locking" implementation is arguably a good thing, since it tags RCU read-side critical sections with the corresponding write-side lock, which can be a valuable documentation aid.

Although there is no perfect solution, at least not yet, the lock-based deferred free approach and the read-side counter-flipping approach should work well for uniprocessor machines, and should scale to modest SMP systems.

Nonetheless, it is also worthwhile to look into other alternatives.

	Reliable?	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			N	N					
Preemptible RCU				X					
Jim Houston Patch			N		N				
Reader-Writer Locking					N		N	N	n
Unconditional Hazard Pointers				X	n	N			
Hazard Pointers: Failure				n	n	N			N
Hazard Pointers: Panic	N			n	n	N			
Hazard Pointers: Blocking		N		n	n	N			
Reference Counters				N	n	N			
`rcu_donereference()`					n	N			N
Lock-Based Deferred Free					N				
Read-Side GP Suppression				N	n				
Read-Side Counters w/ Flipping					n				

Table 2: Comparison of Realtime-Preempt RCU Implementations

3 Migration Approach

One such alternative is the migration approach, where any task that is about to execute any operation that does not have deterministic execution time is migrated to some other CPU. Given the recent advent of low-cost multithreaded and multicore CPUs, "some other CPU" is becoming more commonly available. Now, tasks that are executing in user mode may be preempted at any time, and such tasks may therefore be allowed to run at will. Interrupt handlers might well acquire locks, and therefore interrupts should be directed aways from CPUs that must provide realtime service.

However, any time a process on such a realtime CPU executes a system call, it will likely acquire a lock in the kernel, thus incurring excessive latencies if a non-realtime CPU holds that lock. This might not be a problem for the process executing the system call, after all, a synchronous write to a disk drive involves considerable unavoidable latency. The problem is that *other* realtime processes running on the same CPU would also see degraded realtime response due to locks acquired while setting up the write.

This problem can be addressed by surprisingly small changes:

1. Migrate realtime tasks that execute a system call to a non-realtime CPU, and then back at the completion of the system call.

2. Add checks to the scheduler's load-balancing code so that realtime CPUs do not attempt to acquire other CPUs' runqueue locks and vice versa, except under carefully controlled conditions.

With these changes, the code-path-length work that currently must cover almost the entire kernel can focus on the system-call trap path (but not the system calls themselves) and portions of the scheduler, a much smaller body of code. This approach does have some shortcomings: (1) realtime processes lose their realtime characteristics within system calls, (2) any page faults destroy the page-faulting process's realtime characteristics, (3) unnecessary overhead is incurred migrating any deterministic system calls, and (4) the interactions between realtime system-call migration and other types of migration have not been thoroughly dealt with. In the short term, these can be addressed by carefully coding the realtime application, for example, by pinning the application's pages into memory.

Nonetheless, this approach promises to provide many of the benefits of a realtime OS with minimal modifications to the Linux kernel. In addition, this approach permits the Linux kernel to be incrementally moved to a deterministic realtime design on a system-call-by-system-call and trap-by-trap basis, as needed. It is quite possible that further improvements to this approach will permit Linux to offer hard-realtime response.

Figure 12 shows how the system-call path can be set up to migrate any hard-realtime tasks. Additional code is required to designate the hard-realtime CPUs and tasks, but any of a number of approaches would work here. The two rtoffload functions are quite simple, as shown in Figure 13. Of course, the current implementation of `sched_migrate_task()` acquires the runqueue lock, and therefore is not yet fully hard realtime.

Future work in this area includes engineering deterministic mechanisms that allow `sched_migrate_task()` to transfer tasks from one CPU to another, so that a realtime CPU does not suffer scheduler latency degradation due to spinning on a lock held by a non-realtime CPU. It is possible to apply this approach to single-CPU systems by providing a thin Xen-like layer that makes it appear to Linux that there are really two CPUs, but it is not clear that this is worthwhile given the existence of the realtime-preempt approach.

4 Conclusions

This paper studies a number of approaches to a scalable, performant, and aggressively realtime RCU implementation, concluding that none of them fits the bill. That said, this paper does *not* present a proof that no such implementation is possible, and furthermore, any such implementation would be quite valuable, as it would get much wider workload coverage

```
@@ -303,6 +303,12 @@ static void do_syscall_trace(void)

 void do_syscall_trace_enter(struct pt_regs *regs)
 {
+
+  /* The offload check must precede any non-realtime-safe code. */
+
+  if (test_thread_flag(TIF_SYSCALL_RTOFFLOAD))
+    do_syscall_rtoffload();
+
   if (unlikely(current->audit_context))
     audit_syscall_entry(current, regs->gpr[0],
             regs->gpr[3], regs->gpr[4],
@@ -321,4 +327,9 @@ void do_syscall_trace_leave(void)
   if (test_thread_flag(TIF_SYSCALL_TRACE)
       && (current->ptrace & PT_PTRACED))
     do_syscall_trace();
+
+  /* The offload check must follow any non-realtime-safe code. */
+
+  if (test_thread_flag(TIF_SYSCALL_RTOFFLOAD))
+    do_syscall_rtoffload_return();
 }
```

Figure 12: System-Call Migration

from a single configuration setting. Unfortunately, the probability of such an implementation existing seems to the authors to be quite low. The paper therefore takes a brief look at a migration-based approach to attaining scalability, performance, and realtime response from a Linux kernel. Much work is needed to implement and evaluate this migration-based approach, as well as to continue the search for a scalable realtime-preempt-safe RCU implementation.

Although this paper covers only one of the many obstacles that must be overcome to extend Linux's realtime capabilities, it is clear that continued extension of these capabilities is inevitable. It is no longer a question of whether Linux can take on challenging realtime workloads, but rather a question of what, how, and when.

Acknowledgements

We take a deep bow to Ingo Molnar for his implementation of an extremely preemptible Linux kernel.

Legal Statement

References

[1] Arcangeli, A., Cao, M., McKenney, P. E., and Sarma, D. Using read-copy update tech-

```
static void do_syscall_rtoffload(void)
{
  sched_migrate_task(current, realtime_offload_cpu);
}

static void do_syscall_rtoffload_return(void)
{
  sched_migrate_task(current, realtime_cpu);
}
```

Figure 13: Offload Functions

niques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)* (June 2003), USENIX Association, pp. 297–310.

[2] Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M. 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), pp. 87–100.

[3] Herlihy, M., Luchangco, V., and Moir, M. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing* (October 2002), pp. 339–353.

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

[5] McKenney, P. E. [RFC] RCU and CONFIG_PREEMPT_RT progress. Available: `http://lkml.org/lkml/2005/5/9/185` [Viewed May 13, 2005], May 2005.

[6] McKenney, P. E., Sarma, D., Arcangeli, A., Kleen, A., Krieger, O., and Russell, R. Read-copy update. In *Ottawa Linux Symposium* (June 2002), pp. 338–367. Available: `http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz` [Viewed June 23, 2004].

[7] Michael, M. M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing* (August 2002), pp. 21–30.

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

[9] Morris, J. [PATCH 2/3] SELinux scalability - convert AVC to RCU. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=110054979416004&w=2` [Viewed December 10, 2004], November 2004.

[10] Sarma, D., and McKenney, P. E. Making rcu safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)* (June 2004), USENIX Association, pp. 182–191.