Priority Boosting Preemptible RCU

Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@linux.vnet.ibm.com
paul.mckenney@linaro.org

ABSTRACT

Read-copy update (RCU) is a light-weight synchronization mechanism that has been used in production for well over a decade, most recently, as part of the Linux kernel. The key concept behind RCU is the ability of RCU update-side primitives (`synchronize_rcu()` and `call_rcu()`) to wait on pre-existing RCU read-side critical sections, which are delimited by `rcu_read_lock()` and `rcu_read_unlock()`. The time required for all pre-existing RCU read-side critical sections to complete is termed an RCU grace period. A common usage pattern is to remove an element from a data structure, wait for an RCU grace period to elapse, then free that element.

Older implementations of RCU operated by suppressing preemption across the RCU read-side critical sections, but more recent implementations designed for real-time use permit such preemption. This can lead to a priority-inversion problem, where a low-priority non-real-time task is preempted within an RCU read-side critical section by several medium-priority real-time tasks (at least one per CPU). This situation prevents any subsequent RCU grace periods from completing, which prevents the corresponding memory from being freed, which can exhaust memory, which can block a high-priority real-time task that is attempting to allocate memory.

This situation is similar to lock-based priority inversion, and, as with lock-based priority inversion, and can be solved by temporarily boosting the priority of the low-priority task that was blocked in an RCU read-side critical section.

1. INTRODUCTION

RCU is a synchronization mechanism that allows execution to be deferred until all potentially conflicting operations have completed, which greatly simplifies the design and implementation of concurrent algorithms [5]. Although RCU antecedents date back to 1980 [4], RCU attained widespread use only after its acceptance into the Linux kernel in 2002, where it has since become quite heavily used, as shown in Figure 1. RCU's popularity stems from its solution to the "existence problem" [2]. The potentially conflicting operations, termed *RCU read-side critical sections*, are bracketed with `rcu_read_lock()` and `rcu_read_unlock()` primitives. Production-quality implementations of these primitives scale linearly, are wait-free, are immune to both deadlock and livelock, and incur extremely low overheads. In fact, in server-class (`CONFIG_PREEMPT=n`) Linux-kernel builds, these two primitives incur exactly zero overhead [3].[1]

Any statement that is not within an RCU read-side critical section is a *quiescent state*, and a quiescent state that persists for a significant time period is called an *extended quiescent state*. Any time period during which each thread has occupied at least one quiescent state is a *grace period*. The `synchronize_rcu()` primitive waits for a grace period to elapse. Updates that cannot block may use an asynchronous primitive named `call_rcu()`, which causes a specified function to be invoked with a specified argument at the end of a subsequent grace period. In some (but not all) production-quality implementations, `call_rcu()` simply appends a callback to a per-thread list, and is therefore wait-free [1, 11]. Nevertheless, RCU updates do incur some overhead, so that RCU is best-suited for read-mostly situations.

Taken together, these four primitives implement RCU's grace-period guarantee: a given grace period is guaranteed to extend past the end of any pre-existing RCU read-side critical section [12]. It is important to note that RCU provides this guarantee regardless of the memory model of the underlying computer system.

On weakly ordered systems (a category including all commodity microprocessors), RCU also provides a publish-subscribe guarantee via the `rcu_assign_pointer()` publication and `rcu_dereference()` subscription primitives. These primitives disable any compiler and CPU optimizations that might otherwise result in an RCU reader seeing a pre-initialized view of a newly published data structure. Both of these primitives have $O(1)$ computational complexity with small constant, and incur zero overhead on sequentially consistent computer systems, where "system" includes the compiler.

Although older implementations of RCU relied on disabling preemption across all RCU read-side critical sections, more recent implementations have permitted these critical sections to be preempted in order to improve real-time scheduling latency [3, 8, 10]. As noted earlier, such preemption opens the door to a priority-inversion situation where a low-priority task is preempted in an RCU read-side critical section by medium-priority real-time tasks, preventing any subsequent RCU grace periods from ever completing. If grace periods never complete, the corresponding memory is never freed, eventually running the system out of memory. The resulting hang can be expected to block even the highest-priority real-time tasks.

Section 2 provides an overview of the design, Section 3 de-

[1] Sequent's DYNIX/ptx operating system also provided zero-overhead RCU read-side primitives [11].

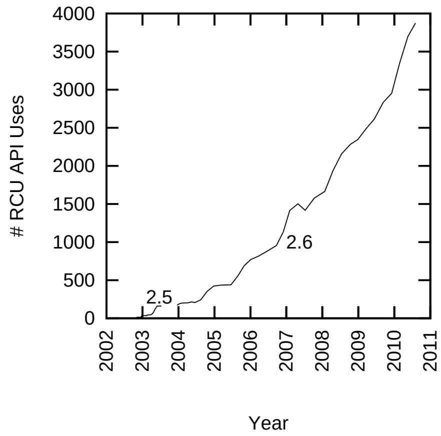

Figure 1: RCU API Usage in the Linux Kernel

scribes code changes, Section 4 describes changes to rcutorture testing, Section 5 plots out a tentative implementation plan, and Section 6 presents concluding remarks.

2. DESIGN OVERVIEW

This section gives an overview of the design, including controls over the boosting process (Section 2.1), additions to data structures (Section 2.2), locking design (Section 2.3), and limitations of RCU priority boosting (Section 2.4).

2.1 Control of Boosting

The operation of RCU priority boosting is controlled by the following:

1. The new `RCU_BOOST` kernel configuration parameter, which depends on `RT_MUTEXES` (default off, at least initially). This dependency is an implementation constraint rather than a policy decision, as in absence of `RT_MUTEXES` the priority-boosting scheduler infrastructure is not compiled into the kernel. Therefore, in absence of `RT_MUTEXES`, `RCU_BOOST` simply cannot do its job. There is also a dependency on `PREEMPT_RCU` by design, given that there is no reason to boost priority for a non-preemptible RCU implementation. The default value of `RCU_BOOST` will be "n" in order to avoid an automatic Linus rejection. If experience indicates that enabling `RCU_BOOST` by default is wise, that change will be made in a later release of the kernel.

2. The priority that the `RCU_SOFTIRQ` task runs at, but only in the -rt patchset, at least until `PREEMPT_SOFTIRQS` reaches mainline. In kernels lacking `PREEMPT_SOFTIRQS`, the priority instead defaults to the least-important real-time priority.

`RCU_SOFTIRQ` Priority	`RCU_BOOST_PRIO` Kernel Parameter	`rcu_boost_prio` Module Parameter	Priority
A	-1	-1	Priority A from `RCU_SOFTIRQ` priority
A	-1	C	Priority C from `rcu_boost_prio`
A	B	-1	Priority B from `RCU_BOOST_PRIO`
A	B	C	Priority C from `rcu_boost_prio`
X	-1	-1	RT Priority 1 by default
X	-1	C	Priority C from `rcu_boost_prio`
X	B	-1	Priority B from `RCU_BOOST_PRIO`
X	B	C	Priority C from `rcu_boost_prio`

Table 1: Relationship of Priority Defaults

3. The new `RCU_BOOST_PRIO` kernel configuration parameter, which depends on `RCU_BOOST`. This specifies the default priority to which blocked RCU readers are to be boosted. A value of zero specifies no boosting. If the value is -1, then the default is taken from the `RCU_SOFTIRQ` priority above. `RCU_BOOST_PRIO` depends on `RCU_BOOST`.

4. The new `rcu_boost_prio` module parameter in `kernel/rcupdate.c`, which also controls the priority to which blocked RCU readers are to be boosted. A value of zero specifies no boosting. If the value is -1, then the default is taken from the `RCU_BOOST_PRIO` kernel parameter above. If the value is neither zero nor a valid real-time scheduler priority, then it is treated as if it was -1, though a warning will be printed in this case. This parameter may also be controlled at runtime via sysfs.

5. The new `RCU_BOOST_DELAY` kernel parameter, which specifies the number of jiffies to wait after a given grace period begins before doing RCU priority boosting for blocked tasks that are stalling that grace period. A value of 0 says never to do RCU priority boosting (but this may be overridden at boot time and at run time). `RCU_BOOST_DELAY` depends on `RCU_BOOST`.

6. A new `rcu_boost_delay` module parameter in `kernel/rcupdate.c`, which also controls the RCU boost delay. A value of -1 says to take the default from the `RCU_BOOST_DELAY` kernel parameter, though any other negative value will have the same effect (but possibly accompanied by a warning). This parameter may also be controlled at runtime via sysfs.

The relationship between the `RCU_SOFTIRQ` priority, the `RCU_BOOST_PRIO` kernel parameter, and the `rcu_boost_prio` module parameter is involved, so their relationship is shown by Table 1. A letter A, B, or C in the first column indicates some real-time priority (which currently ranges from 1 to

99 in the Linux kernel), while the letter X indicates a kernel without threaded softirqs.[2] The "Priority" column indicates what controls the resulting task priority.

2.2 New Data Structures

Although RCU priority boosting does not introduce any new data structures to RCU, it does add fields and values to a number of the existing data structures under `RCU_BOOST` ifdef as follows:

1. Add a `RCU_READ_UNLOCK_BOOSTED` value to those values that the `->rcu_read_unlock_special` `task_struct` field.

2. Add an `rcu_boost_prio` global variable to `kernel/rcupdate.c`, which is initialized to the `RCU_BOOST_PRIO` kernel configuration parameter. This variable is exported as a module parameter and via `sysfs`, as noted in Section 2.1.

3. Add an `rcu_boost_prio_old` global variable to `kernel/rcupdate.c`, which is also initialized to the the `RCU_BOOST_PRIO` kernel configuration parameter. This is used to detect changes to the `rcu_boost_prio` global variable.

4. Add an `rcu_boost_delay` global variable to `kernel/rcupdate.c`, which is initialized to the `RCU_BOOST_DELAY` kernel configuration parameter. This variable is exported as a module parameter and via `sysfs`, as noted in Section 2.1.

5. A new `boost_boost_kthread` global variable that contains a pointer to the `task_struct` structure for a second-order booster kernel thread. This kernel thread scans the `->rcu_prio` fields, reboosting the corresponding first-order booster kernel threads as needed.

6. Add an `rcu_needboost` global variable, which is initialized to zero. This counter signals the second-order booster kernel thread to wake up.

7. Add an `rcu_boostwq` global variable, which is initialized using the `DECLARE_WAIT_QUEUE_HEAD()` macro. This wait queue is where the second-order booster kernel thread blocks when there is no boosting to be done.

8. Change the `TINY_PREEMPT_RCU` type of `rcu_preempt_ctrlblk` to be `rcu_node` to promote common code, and move the definition from `kernel/rcutiny_plugin.h` to `kernel/rcutiny.h`. A (trivial) definition of `rcu_for_each_leaf_node()` will also need to be added to `kernel/rcutiny.h`. Note that `kernel/rcupdate.c` will need to include one of `kernel/rcutiny.h` or `kernel/rcutree.h` under appropriate ifdef.

The `TINY_PREEMPT_RCU` implementation additionally needs the following additional fields in existing data structures, again under `RCU_BOOST` ifdef:

1. Add a `->boost_tasks` pointer to the `rcu_preempt_ctrlblk` structure, which is initialized to `NULL`. This field points to the first tasks structure in the `->blkd_tasks` lists that has needs to be boosted but has not yet been boosted, or `NULL` if there are no tasks in need of RCU priority boosting.

2. Add an `->rcu_prio` field to the `rcu_preempt_ctrlblk` structure, which is initialized to `MAX_PRIO-1`. This field records the priority to which tasks should be boosted. Note that it is possible for this field to have a different value than the `rcu_boost_prio` global variable. This situation indicates that the desired boost priority changed recently, and that all tasks that have already been boosted need to be boosted again (or deboosted in the case where the `rcu_boost_prio` global variable has changed to a lower priority.

3. Add a `->boosttime` field to the `rcu_preempt_ctrlblk` structure, which is initialized to `jiffies+rcu_boost_delay`. This field records the jiffies value at which boosting should begin.

4. Add a `->boosted_this_gp` field to the `rcu_preempt_ctrlblk`, which is initialized to zero both at initialization and at the beginning of each grace period. This field tracks whether or not RCU priority boosting has been initiated during the current grace period.

5. Add a `->boost_kthread` field to the `rcu_preempt_ctrlblk` structure, which is initalized to `NULL`. This field contains a pointer to the booster kernel thread's task structure, which is needed to allow the booster kernel thread's priority to be boosted.

6. Add a `->needboost` field to the `rcu_preempt_ctrlblk` structure, which is initialized to zero. This flag signals the booster kernel thread to wake up.

7. Add a `->boostwq` field to the `rcu_preempt_ctrlblk` structure, which is initialized using the `DECLARE_WAIT_QUEUE_HEAD()` macro. This wait queue is where the booster kernel thread blocks when there is no boosting to be done.

8. Add a `->boost_rt_mutex` field to the `rcu_preempt_ctrlblk` structure, which is an `rt_mutex` structure used to carry out the boosting. This field is initialized via the `DEFINE_RT_MUTEX()` macro.

The `TREE_PREEMPT_RCU` implementation additionally needs the same additional fields that `TINY_PREEMPT_RCU` does, but instead in each `rcu_node` structure (though `->boosttime` and `->boosted_this_gp` might instead go into the `rcu_state` structure, given the `rcu_time_to_boost()`, `rcu_boosting_initiated()`, and `rcu_has_boosted_this_gp()` access functions described below). In addition, the `->needboost` field takes on an additional value to tell the corresponding booster kernel thread to stop.

The `rcu_time_to_boost()` access function returns non-zero if the current grace period has extended long enough that boosting is required. The `rcu_boosting_initiated()` access function records the fact that boosting has been initiated. The `rcu_has_boosted_this_gp()` access function returns non-zero if boosting has already been initiated during the current grace period.

[2] As of this writing, mainline Linux does not permit threaded softirqs, aside from the non-realtime `run_ksoftirqd()` task that is used to handle overflow from the softirq environment. However, `PREEMPT_RT` kernels that include the -rt patchset provide a `PREEMPT_SOFTIRQS` kernel parameter that causes softirqs to be executed in the context of a real-time thread whose priority may be controlled at run time.

Figure 2: Limitations of Priority Boosting

In PREEMPT_SOFTIRQS kernels, additional data will be needed if the RCU_SOFTIRQ tasks are also to be boosted. However, the initial implementation will assume that the initial priority chosen for the RCU_SOFTIRQ tasks is sufficient, and will therefore refrain from boosting them.

Once the SRCU implementation is folded into TREE_PREEMPT_RCU implementation, an addition field will be needed to link together the corresponding rcu_node structures for TREE_PREEMPT_RCU and for the SRCU instances that are subject to priority boosting. Priority boosting may only be used by SRCU instances for which SRCU read-side critical sections only acquire mutexes, but do no other general-blocking operation. In contrast, SRCU instances for which SRCU read-side critical sections do things like block waiting for network input cannot use priority boosting. After all, how are you going to decide what to boost to make the network packet arrive more quickly? As can be seen in Figure 2, the limitations of priority boosting also affect other scenarios.

2.3 Locking Design

The locking design of RCU priority boosting must respect RCU's current locking design, which uses irq-disabled spinlocks, but must also accommodate the scheduler's priority-boosting locking design, which requires that boosting be undertaken with preemption enabled. The reason that preemption must be enabled while boosting is that disabling preemption would result in excessive scheduling latencies. The conflict between these two locking designs is resolved by the following simplifying constraints:

1. Boost only blocked tasks, so that the lock guarding the ->blkd_tasks list may be used to coordinate boosting and so that a mutex to be easily used to boost priority.

2. A given booster thread boosts only one blocked thread at a time, easing implementation of rt_mutex-based.

3. Provide a separate kernel thread to do the boosting. This thread first modifies the relevant RCU-specific state with interrupts disabled and under protection of the appropriate RCU spinlock, then invokes the scheduler function that adjusts priorities.

The booster kernel thread does bookkeeping under the irq-disabled RCU spinlocks, which consists of updating the ->boost_tasks pointer, creating an rt_mutex on the stack, invoking rt_mutex_init_proxy_locked() on this mutex on behalf of the task to be boosted, releasing those locks, and enabling interrupts (thus re-enabling preemption). Only then does the boster kernel thread invoke the scheduler to adjust the priority of the task in question as well as any processes connected to it via priority-inheritance chains, and by attempting to acquire the afore-mentioned on-stack rt_mutex. This approach leverages the pre-existing priority-inheritance mechanism, thereby boosting not only the task in question, but any other tasks in its priority-inheritance chain. The priority-inheritance mechanism already handles race conditions involving concurrent changes in priority, for example, via the sched_setscheduler() system call.

2.4 Limitations of RCU Priority Boosting

In addition to the limitations called out in Figure 2, RCU priority boosting is subject to the following limitations:

1. Real-time processes running at priorities higher than the current RCU boost priority can still block RCU grace periods.

2. In PREEMPT_RT kernels, real-time processes running at priorities higher than the RCU_SOFTIRQ thread can still block grace periods. (And networking. And disk I/O. And...)

3. RCU priority boosting can degrade scheduling latencies for real-time processes running at priorities lower than the current RCU boost priority.

4. Theoreticians will probably choke on the concept of RCU priority boosting. On the other hand, a good many theoreticians have already choked on RCU, even without priority boosting, so why worry?

3. OVERVIEW OF CODE CHANGES

This section gives an overview of the code changes required, using pseudo-code rather than actual C. Section 3.1 describes the priority booster kernel thread, Section 3.2 describes the second-order priority booster kernel thread, Section 3.3 describes de-boosting, Section 3.4 describes changes to the core RCU grace-period code, Section 3.5 describes how the blkd_task lists are merged in TREE_PREEMPT_RCU when the last CPU for a given rcu_node structure is offlined, Section 3.7 describes statistics, and Section 3.8 describes the changes required to the scheduler code.

3.1 Priority Booster Kernel Thread

The priority booster kernel threads cannot be created until the scheduler is up and running, and therefore cannot be created in rcu_init().[3] Instead, these can be started via kthread_run() from rcu_scheduler_starting(), which currently enables debug checks that are disabled during early boot, and disables single-CPU optimizations that operate only during early boot. This means that the ifdefs covering the definitions of rcu_scheduler_active and rcu_scheduler_starting() in TINY_PREEMPT_RCU must now include RCU_BOOST. The return value from kthread_run() is a pointer to the task_struct structure, which must be recorded in the appropriate ->boost_kthread field.

Additional startup/shutdown work is required for TREE_PREEMPT_RCU:

[3] I know, because I have tried!

1. Each booster thread must be affinitied to the set of CPUs associated with the corresponding rcu_node structure.

2. When the last CPU associated with the corresponding rcu_node structure goes offline, the corresponding booster thread must be stopped via kthread_stop() as follows:

   (a) Set ->needboost to two to indicate a need to stop.

   (b) Wake up the booster kernel thread.

   (c) Invoke kthread_stop().

   Note that this process must be carried out *after* all blocked tasks have been migrated to the root rcu_node structure. One way to accomplish this is to place the new code near the end of the rcu_preempt_offline_tasks() function.

3. When the first CPU associated with the corresponding rcu_node structure comes online, the corresponding booster thread must be created via kthread_start(). The rcu_preempt_init_percpu_data() function is a good home for this functionality, but only for the CPU-online case. An explicit check for rcu_scheduler_active is required to avoid invoking kthread_start() before the scheduler is ready. The code added to rcu_preempt_init_percpu_data() should be placed before the call to rcu_init_percpu_data() so that a simple test of ->qsmaskinit being equal to zero will determine whether this is indeed the first CPU coming online.

Once created, the booster kernel thread operates as follows:

1. Blocks on the combination of ->needboost and ->boostwq using wait_event() so as to wake up when ->needboost is set to one.

2. If the value of ->needboost is two, invoke kthread_stop() to terminate execution.

3. Disable interrupts, and, if in TREE_PREEMPT_RCU, acquire the rcu_node structure's ->lock.

4. Enter an RCU read-side critical section via rcu_read_lock().

5. Check the ->boost_tasks pointer. If it is NULL, set ->needboost to zero, release the ->lock (if in TREE_PREEMPT_RCU), re-enable interrupts, and restart from the beginning. Otherwise, continue.

6. Set local variable p to the value of ->boost_tasks, but translated back to the task_struct structure.

7. Advance ->boost_tasks to the next element of the ->blkd_tasks list. If there is no next element, instead set ->boost_tasks to NULL and set ->needboost to zero.

8. Invoke rt_mutex_init_proxy_locked() on ->boost_rt_mutex and p.

9. Set the RCU_READ_UNLOCK_BOOSTED bit in the p->rcu_read_unlock_special bitmask.

10. If in TREE_PREEMPT_RCU, release ->lock and in either case re-enable interrupts. This has the side-effect of re-enabling preemption.

11. Exit the RCU read-side critical section via rcu_read_unlock().

12. Invoke rt_mutex_lock() on ->boost_rt_mutex.

13. Restart from the beginning.

In the TREE_PREEMPT_RCU case in PREEMPT_SOFTIRQS kernels, the booster thread must also control the priority of each of the RCU_SOFTIRQ tasks associated with CPUs corresponding to that booster thread's rcu_node structure. However, the booster thread need not de-boost the PREEMPT_SOFTIRQS tasks below their original priority.

The code for the booster kernel thread lives in kernel/rcupdate.c so that it can be shared between TINY_PREEMPT_RCU and TREE_PREEMPT_RCU.

3.2 Second-Order Priority Booster Kernel Thread

The purpose of the second-order priority booster kernel thread is to ensure that changes in the desired RCU-boost priority are dealt with in a timely fashion. To see the need for this, suppose that some CPU-bound real-time processes are preventing the thread that is currently boosted from running. One reaction to this might be to increase the RCU-boost priority via the rcu_boost_prio sysfs entry. However, the priority booster kernel thread is blocked, and thus cannot react to this change. The scheduling-clock interrupt can boost the priority of this thread, but that won't help unless the threads that it is blocked on are also boosted. But there might be a long priority-inheritance chain of rt_mutex_lock() calls from one thread to the next, and all the tasks in that chain must be boosted. Although it is legal to boost a single task from the scheduling clock interrupt handler, it is necessary to have preemption enabled when boosting a priority-inheritance chain. This priority-inheritance chain must be boosted by another thread, and this is the job of the second-order priority booster kernel thread.

The second-order priority booster kernel thread operates as follows:

1. Blocks on a combination of the rcu_boostwq global wait queue and the rcu_needboost global variable, so that it will awaken when rcu_needboost is non-zero.

2. Set the rcu_needboost global variable to zero and do smp_mb() to ensure that the checks happen after the zeroing.

3. For each leaf-level rcu_node structure, do the following:

   (a) If the ->rcu_prio field is equal to the rcu_boost_prio global variable, restart from the first step.

   (b) Disable interrupts, and in TREE_PREEMPT_RCU, acquire the rcu_node's ->lock field.

   (c) Set the value of the ->rcu_prio field to that of the rcu_needboost global variable.

(d) If the ->boost_tasks field is non-NULL, set its value to that of the ->gp_tasks. This forces re-boosting of any already-boosted tasks.

(e) If in TREE_PREEMPT_RCU, release the rcu_node's ->lock field, and in either case re-enable interrupts.

(f) Invoke sched_setscheduler_nocheck() on the task referenced by the ->boost_kthread field, setting its scheduling policy to SCHED_FIFO and its priority to ->rcu_prio (via the sched_param structure).

4. Restart this procedure from the beginning.

Of course, TINY_PREEMPT_RCU has only one rcu_node structure, and the compiler can be trusted to optimize away the loop, especially given the definition of rcu_for_each_leaf_node().

3.3 De-Boosting

The de-boosting process is much simpler than the boosting process described in Section 3.1 because:

1. The task is operating on itself, and therefore need not enter an RCU read-side critical section.

2. The task is running, and therefore must have an empty priority-inheritance list. It is therefore unnecessary for rt_mutex_unlock() to invoke rt_mutex_adjust_pi(), which in turn makes it unnecessary to ensure that preemption is enabled.

The de-boosting takes place in rcu_read_unlock_special() with interrupts enabled and, in the case of TREE_PREEMPT_RCU, with the rcu_node structure's ->lock *not* held.[4] The following very simple procedure therefore suffices:

1. Release the booster kernel thread's rt_mutex using rt_mutex_unlock() on ->boost_rt_mutex.

3.4 Core RCU Grace-Period Code

The scheduling-clock interrupt invokes rcu_preempt_check_callbacks() every jiffy on each CPU that has at least one RCU callback in flight.[5] This function can therefore check to see if rcu_boost_prio differs from both rcu_boost_prio_old and -1, and if so, carry out the following procedure:

1. Set the value of rcu_boost_prio_old to -1 using the xchg() primitive. If the value returned is -1, someone else is doing this work, so skip the following steps.

2. Invoke rcu_boosting_initiated() to record the boost attempt.

3. Invoke sched_setscheduler_nocheck() on the task referenced by the boost_boost_kthread field, setting its scheduling policy to SCHED_FIFO and its priority to rcu_boost_prio (via the sched_param structure).

4. Execute an smp_mb() to ensure that the above reads are executed before the change to rcu_needboost.

5. Set the rcu_needboost global variable to 1.

6. Invoke wake_up() on the global rcu_boostwq wait queue.

7. Set the value of rcu_boost_prio_old to that of rcu_boost_prio using the xchg() primitive. Complain if the result is not -1.

Note that it is important that the global rcu_boost_prio be read exactly once. The ACCESS_ONCE() macro can be used to enforce this, copying the value of rcu_boost_prio to a local variable.

The rcu_preempt_check_callbacks() must also initiate RCU priority boosting if the current grace period extends for too long (it also resets the time, atomically, of course). It uses a rcu_time_to_boost() function supplied by each of TINY_PREEMPT_RCU and TREE_PREEMPT_RCU for this purpose. If this function indicates that it is time to do boosting,

However, grace-period-timeout boosting must interact nicely with change-in-priority boosting. One way to do this is to follow the same procedure used above for reacting to a sysfs-induced change in priority. This means that the condition for entering the above procedure is as follows:

1. The rcu_time_to_boost() function returns non-zero, or

2. the rcu_has_boosted_this_gp() function returns non-zero, and:

   (a) rcu_boost_prio is not -1, and

   (b) rcu_boost_prio is not equal to rcu_boost_prio_old.

Of course, this condition is not even checked unless there is an RCU grace period in progress.

3.5 Merging Task Lists onto the Root RCU-Node Structure

In the TREE_PREEMPT_RCU implemementation, when the last CPU corresponding to a given leaf rcu_node structure goes offline, that structure's list of blocked tasks must be merged onto that of the root rcu_node structure. Of course, if all of the ->gp_tasks, ->exp_tasks, and ->boost_tasks fields are NULL for both the leaf and root structures, these lists can be merged in any order, however, if any of these pointers are non-NULL, care is required.

3.5.1 Canonical Approach

One approach to merging the lists would be to keep track of the relative order in the list of the tasks referenced by ->gp_tasks, ->exp_tasks, and ->boost_tasks, and then merge the lists piecewise, taking into account that grace-period initialization might be in progress, so that the root and leaf rcu_node structures might be operating on different grace periods. However, there are a very large number of cases to consider, and the resulting code could therefore be complex and difficult to test thoroughly. Therefore, a simpler algorithm is desirable.

3.5.2 Helpful Constraints

Fortunately, there are a number of constraints that simplify the task significantly. First, the task referenced by the ->boost_tasks pointer can never precede that referenced by the ->gp_tasks pointer. Second, if a given ->

[4] But note that rcu_read_unlock()'s caller might have disabled interrupts, in which case they will obviously still be disabled.

[5] If no CPU has an RCU callback in flight, then there is no reason to do RCU priority boosting.

gp_tasks pointer is NULL, then the corresponding ->boost_tasks pointer must also be NULL. Third, if the root and leaf rcu_node structures are operating on different grace periods, the ->gp_tasks pointer for the leaf rcu_node structure must be NULL, because otherwise it would not be legal to be initializing the rcu_node structures for a new grace period. Finally, expedited grace periods should be rare and short-lived, so some sub-optimal handling of normal grace periods is permissible in that case.

These constraints rely on the following prioritization:

1. Most important: never delay an expedited grace period. After all, they are supposed to be expedited.

2. Where possible without undue complexity and without delaying an expedited grace period, avoid delaying a normal grace period.

3. Finally, when feasible, avoid redundant boosting of tasks. This is lowest priority because boosting should be rare, so the combination of priority boosting and offlining the last CPU of a given rcu_node structure should be extremely rare.

3.5.3 Simplified Approach

Given these constraints, a reasonably simple approach is summarized in Table 2. In all cases, the leaf rcu_node structure's list is emptied and its gp_task, ->boost_tasks, and ->exp_tasks pointers are set to NULL. Note that although the approach outlined in this section can delay normal grace periods and can cause some tasks to be repeatedly boosted, this can happen only when CPUs go offline, and even then only the first time that a given CPU goes offline during a given grace period. A CPU that goes offline for the second time during a given grace cannot have tasks blocking the current grace period.

The first column of Table 2 gives the case number. The second set of three columns give the initial state of the leaf rcu_node structure's three list pointers. A "Y" indicates that the corresponding pointer is non-NULL, an "X" indicates that the corresponding pointer might or might not be NULL, and a blank indicates that the corresponding pointer is NULL. An emboldened letter indicates that the corresponding operation (normal grace period or boosting) is handled sub-optimally in the corresponding case. The second set of three columns similarly gives the initial state of the root rcu_node structure's three list pointers. The final column summarizes the list-merging actions, which are covered in more detail in the following paragraphs.

In case 1, the leaf rcu_node structure's list has no blocked tasks that are relevant to the current grace periods, so these tasks may simply be spliced onto the head of the root rcu_node structure's list. Similarly, in case 2, the root rcu_node structure's list has no blocked tasks that are relevant to the current grace periods, so the leaf's tasks may simply be spliced onto the tail of the root rcu_node structure's list. In addition, in this second case, the root rcu_node structure's ->gp_tasks, ->boost_tasks, and ->exp_tasks are set to the corresponding values from the leaf rcu_node structure.

In both of these cases, neither of the grace periods are required to wait needlessly on tasks, nor are any tasks needlessly boosted.

However, in case 3 both rcu_node structure have at least one task blocking the current expedited grace period, and the root rcu_node structure might also have tasks blocking the current normal grace period, some of which might also need to be boosted. The list splicing is done in two steps: First, the head of the leaf rcu_node structure's list (up to but not including the first task blocking the current expedited grace period) is spliced onto the head of the root rcu_node structure's list, where it is guaranteed not to result in unnecessary work. Second, the remainder of the leaf rcu_node structure's list is spliced onto the tail of the root rcu_node structure's list. This second splicing might cause the current grace period to wait needlessly on these tasks, and further might cause these tasks to be needlessly boosted. This is preferable to delaying the current expedited grace period, which after all is supposed to be expedited, and is also preferable to the complexity that would be required for exact splicing.

Case 4 is spliced in the same manner as for case 3, and in addition the root rcu_node structure's ->exp_tasks pointer is set to the value of the leaf rcu_node structure's ->exp_tasks pointer. This again might cause the current normal grace period to wait unnecessarily on the tasks from the leaf rcu_node structure, but it is better to delay the normal grace period than to delay the expedited grace period.

Case 5 is the reverse of case 4, and is handled by splicing the leaf rcu_node structure's list to immediately precede the first task in the root rcu_node structure's list that is blocking the current expedited grace period. This has the same effect as for case 4, but with the roles of the root and leaf rcu_node structures reversed.

In case 6, both lists have tasks blocking the current normal grace period, and the root rcu_node structure further has at least one task blocking the current expedited grace period. This case is handled by splicing the entire leaf rcu_node structure's list to the head of the root rcu_node structure's list, then setting the root's ->gp_tasks pointer to that of the leaf. In addition, if the leaf's ->boost_tasks pointer is non-NULL, then its value is assigned to that of the root. This of course has the unfortunate side effect of making the current grace period wait on all of the tasks on the root rcu_node structure's list, but we cannot do better given that we don't know which of the root ->gp_tasks and ->exp_tasks pointers comes first in the list. Again, we choose to optimize for expedited grace periods.

Case 7 has tasks from both lists blocking the normal grace period, but none blocking the expedited grace period. This case is handled by splicing the head of the leaf rcu_node structure's list, up to but not including the task referenced by ->gp_tasks, to the head of the root rcu_node structure's list. The remainder of the leaf rcu_node structure's list is appended to the tail of the root rcu_node structure's list. This case avoids excessive waiting by the normal grace period, but might redundantly boost some of the tasks from the leaf rcu_node structure. Since boosting will normally be quite rare, this is a reasonable tradeoff to make.

Case 8 is the reverse of case 3. Here, the leaf rcu_node structure's list is spliced into the root rcu_node structure's list to precede the task referenced by the root's ->exp_tasks pointer. In addition, the root's ->gp_tasks, ->boost_tasks, and ->exp_tasks pointers to the values of the corresponding leaf pointers. This gives the same result as in case 3, but with the roles of root and leaf reversd.

Case 9 is the reverse of case 6. Here, the leaf rcu_node structure's list is spliced onto the tail of the root rcu_node

Case	Leaf			Root			List-Merge Action
	`->gp_tasks`	`->boost_tasks`	`->exp_tasks`	`->gp_tasks`	`->boost_tasks`	`->exp_tasks`	
1				X	X	X	Splice leaf at head of root.
2	X	X	X				Splice leaf at tail of root, set root's `->gp_tasks`, `->boost_tasks`, and `->exp_tasks` to those of leaf.
3			Y	**X**	**X**	Y	Splice leaf's range from head to `->exp_tasks` to the head of the root's list. Splice the remainder of leaf's list to the tail of root's list.
4			Y	**Y**	**X**		Splice leaf's range from head to `->exp_tasks` to the head of the root's list. Splice the remainder of leaf's list to root's tail and update root's `->exp_tasks` to that of leaf.
5	**Y**	**X**				Y	Splice leaf's list immediately precede `exp_task` in root's list. Set root's `->gp_tasks` and `->boost_tasks` to those of the leaf.
6	**Y**	**X**		Y	X	Y	Splice leaf's list to the head of root's list. Set root's `->gp_tasks` to that of the leaf. If the leaf's `->boost_tasks` is non-`NULL`, assign it to that of the root.
7	Y	X		Y	X		Splice leaf's range from head to `->gp_tasks` to the head of root's list. Splice the remainder of leaf's list to the tail of root's list.
8	**Y**	**X**	Y			Y	Splice leaf's list to precede the root's `->exp_tasks`. Set root's `->gp_tasks`, `->boost_tasks`, and `->exp_tasks` to those of the leaf.
9	Y	**X**	Y	**Y**	**X**		Splice the leaf's list to the tail of the root's list. Set the root's `->exp_tasks` to that of the leaf. If the root's `->boost_tasks` is `NULL`, set it to that of the leaf.
10	**Y**	**X**	Y	**Y**	**X**	Y	Splice the leaf's list to precede root's `->exp_tasks`. Set the root's `->exp_tasks` to that of the leaf. Set the root's `->gp_tasks` to the head of the list. If the root's `->boost_tasks` is `NULL`, set it to that of the leaf, otherwise if the leaf's `->boost_tasks` is non-`NULL` set the root's `->boost_tasks` to the head of the list.

Table 2: Leaf-to-Root List-Merge Cases

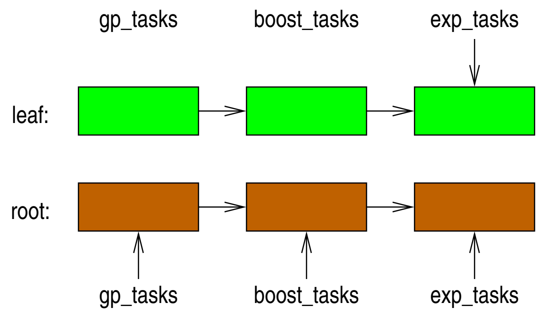

Figure 3: Initial List Configuration for Leaf and Root Nodes

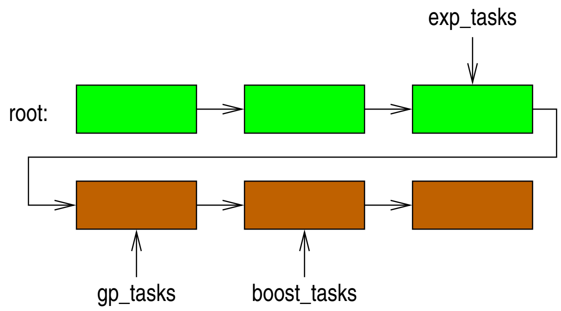

Figure 4: Merged List Configuration for Root Nodes

structure's list, and the root's `->exp_tasks` field is set to that of the leaf. In addition, if the root's `->boost_tasks` pointer is `NULL`, its value is taken from that of the leaf. This gives the same result as in case 6, but with the roles of the root and leaf reversed.

Case 10 has tasks in both `rcu_node` structures blocking both the normal and the expedited grace periods. Here, the leaf `rcu_node` structure's list is spliced into the root `rcu_node` structure's list to precede the task referenced by the root's `->exp_tasks` pointer. Set the root's `->gp_tasks` pointer to the head of the resulting list. If the root's `->boost_tasks` pointer is `NULL`, set it to that of the leaf, otherwise if both `->boost_tasks` pointers are non-`NULL`, set the root's `->boost_tasks` pointer to the head of the resulting list. This can cause the current normal grace period to wait unnecessarily for tasks and can result in boosting already-boosted tasks. Again, however, the expedited grace period is never unnecessarily delayed.

3.5.4 *Yet More Simplified Approach*

Although the above approach is simpler and faster than an optimal merge of the lists, it is still quite complex. This is inappropriate, especially given that the situation leading up to it is quite rare: the last of 64 CPUs corresponding to a given `rcu_node` structure having gone offline. In addition, the `synchronize_rcu_expedited()` function can sometimes fall back to using the slower `synchronize_rcu()` function, which makes it hard to justify all the complexity in the approach outlined in Section 3.5.3 just to make `synchronize_rcu_expedited()` a bit faster in this rare situation.

Therefore, this yet-more-simplified approach simply splices the leaf `rcu_node` structure's `->blkd_tasks` list at the head of the root structure's list. For each of the `->gp_tasks`, `->boost_tasks`, and `->exp_tasks` pointers, if a given leaf pointer is non-`NULL`, its value is assigned to the corresponding root pointer. For example, in the situation depicted in Figure 3, of the leaf structure's pointers, only `->exp_tasks` pointer is non-`NULL` while all of the root structure's pointers are non-`NULL`. Therefore, when the leaf's `->blkd_tasks` list is merged onto that of the root, the resulting `->exp_tasks` pointer is taken from the leaf, while the `->gp_tasks` and `->boost_tasks` pointers are taken from the root, as shown in Figure 4.

Pseudo-code for this operation is as follows:

1. (The leaf `rcu_node` structure's `->lock` field will already have been acquired by the caller, and interrupts will have been disabled.)

2. For each element of the leaf's `->blkd_tasks` list:

   (a) Obtain a pointer to the task corresponding to the current list element.

   (b) Acquire the root `rcu_node` structure's `->lock`.

   (c) Remove the current element from the leaf's `->blkd_tasks` list.

   (d) Set the task's `->rcu_blocked_node` pointer to reference the root `rcu_node` structure.

   (e) Add the current element to the head of the root `rcu_node` structure's `->blkd_tasks` list.

   (f) Release the root `rcu_node` structure's `->lock`.

3. If the leaf `rcu_node` structure's `->gp_tasks` pointer is non-`NULL`, assign its value to the root's `->gp_tasks` pointer.

4. If the leaf `rcu_node` structure's `->boost_tasks` pointer is non-`NULL`, assign its value to the root's `->boost_tasks` pointer.

5. If the leaf `rcu_node` structure's `->exp_tasks` pointer is non-`NULL`, assign its value to the root's `->exp_tasks` pointer.

6. Set the leaf `rcu_node` structure's `->gp_tasks`, `->boost_tasks`, and `->exp_tasks` pointers all to `NULL`.

This provides reasonably efficient operation, while reducing the amount of code in the kernel.

3.6 Boosting Callback Processing

Initial rcutorture tests showed that boosting the tasks preempted in RCU read-side critical sections is insufficient: it is also necessary to boost RCU callback invocation. invocation. Otherwise, the grace period might complete (in `TINY_PREEMPT_RCU`, but unless the callbacks are invoked, no memory will actually be freed.

Unfortunately, RCU callbacks are currently invoked within softirq context, with overflow to ksoftirqd, which cannot be conveniently boosted. We therefore create separate kthreads for each CPU that can be readily boosted. The notion of separate tasks for callback processing is not new, in fact this concept appeared early in RCU's Linux implementation [9].

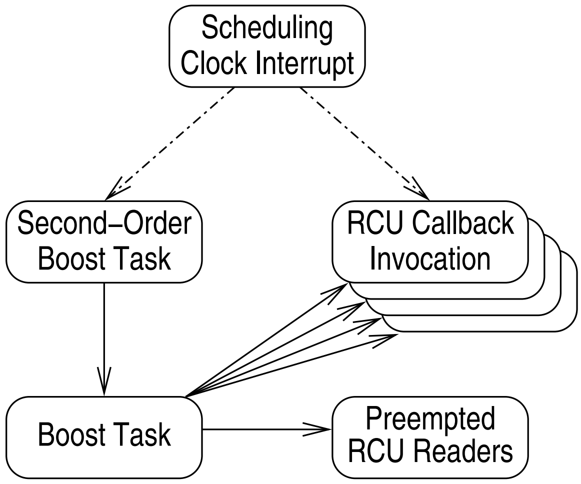

Figure 5: Relationship of Priority-Boosting Tasks

These are related as shown in Figure 5. The scheduling-clock interrupt code will cause RCU callback invocation to run, but does not adjust the priority. Instead, the second-order boost task is invoked, which boosts any preempted RCU readers as well as any of the RCU callback invocation kthreads that have at least one callback blocked by the current grace period (or any earlier grace period).

3.7 Statistics

Statistics are useful for debugging, performance analysis, and for evaluating the effectiveness of priority boosting in given situations. The following statistics need to be gathered:

1. The number of RCU read-side critical sections that have blocked, collected on a per-`rcu_node` basis in `TREE_PREEMPT_RCU` and globally in `TINY_PREEMPT_RCU`.

2. The number of RCU read-side critical sections that have been boosted, also collected on a per-`rcu_node` basis in `TREE_PREEMPT_RCU` and globally in `TINY_PREEMPT_RCU`.

3. The number of RCU read-side critical sections that have unboosted themselves, also collected on a per-`rcu_node` basis in `TREE_PREEMPT_RCU` and globally in `TINY_PREEMPT_RCU`.

In `TREE_PREEMPT_RCU`, these statistics will be output via a new file in the `sysfs rcu` directory. In `TINY_PREEMPT_RCU`, statistics will be added to a new `kernel/rcutiny_trace.c` file, in a manner similar to `kernel/rcutree_trace.c`. As with `TREE_PREEMPT_RCU`, tracing in `TINY_PREEMPT_RCU` will be optional in order to keep the memory footprint small. Unlike `TREE_PREEMPT_RCU`, when tracing is disabled in `TINY_PREEMPT_RCU`, the data will not be collected, again in order to keep the memory footprint small.

3.8 Changes to Scheduler Code

One of the biggest benefits of the new design worked out with Peter Zijlstra and Thomas Gleixner is that it does not require any changes to the scheduler. Woo-hoo!!! ;-)

4. TESTING

Testing will be carried out by the existing rcutorture module in the Linux kernel. This module will be modified as follows:

1. Add a `test_boost` module parameter, which defaults to one. A value of one says to test RCU priority boosting only if the specified flavor of RCU supports this notion, while a value of two says to test RCU priority boosting even if the specified flavor of RCU does not support this notion. This latter is useful for testing the test.

2. Add a `can_boost` flag to the `rcu_torture_ops` structure.

3. If the values of `test_boost` and `can_boost` indicate that boosting should be tested, a high-priority realtime task is spawned, one per CPU. These tasks periodically run in unison, periodically registering callbacks and checking for their completion. The time that each thread should wait is controlled by the new `rcu_boost_delay_jiffies()` function – if a given grace period does not complete in twice that value plus (say) ten jiffies, the thread complains that RCU priority boosting is not working.

In addition, early versions of the patches will be made available on LKML, with the hope that people who have experienced OOM issues will try it out and report results.

5. IMPLEMENTATION

This work will be implemented in the following stages:

1. Negotiate an appropriate interface to the scheduler. This task has been started, and thus far has resulted in the improved design noted in Section 3.8. However, additional adjustments are likely to be required.

2. Implement the rcutorture changes, verifying that it is possible to reliably preempt low-priority RCU readers.

3. Fix some performance bugs in `TINY_PREEMPT_RCU`'s `rcu_preempt_check_callbacks()` function, test, and submit the patch.

4. Implement RCU priority boosting in `TINY_PREEMPT_RCU`, including tracing, test and submit patch along with the rcutorture patch.

5. Move `TREE_PREEMPT_RCU` to a single blocked-tasks list, in order to allow common code to boost both implementations. Test and submit the patch.

6. Implement RCU priority boosting in `TREE_PREEMPT_RCU`, including tracing, test and submit the patch.

7. Fixes and issues located during the review process.

8. Update `Documentation/RCU/` in-tree documentation to note new kernel configuration parameters, rcutorture options, and stall-warning resolutions.

9. Produce community documentation, for example, an LWN article. This should preferably be combined with that for `TINY_PREEMPT_RCU` and `TREE_PREEMPT_RCU`, possibly as a series.

6. CONCLUSIONS

Although many common real-time programming methodologies avoid long-term stalling of low-priority RCU readers, for example, by mandating strict limits on CPU utilization, there are other methodologies that use CPU-bound real-time processes. In addition, even given strict limits on CPU utilization, it can be very helpful if bugs that violate these limits do not hang the system.

Therefore, RCU priority boosting can be helpful to a wide range of real-time programming methodologies, especially for small-memory machines that cannot ride out overlong RCU grace periods. This RCU priority-boosting design is a great improvement over earlier attempts [7] in that no changes to the scheduler are required, in the common case, only those tasks blocking the current grace period are boosted, and the algorithm and data structures are much simpler. These improvements are largely due to improvements in the design and implementation of preemptible RCU, as previous implementations were notoriously complex [6].

Acknowledgments

Many thanks to Peter Zijlstra, Thomas Gleixner, and Darren Hart for reviewing my previous design and suggesting the greatly improved design discussed in Section 2.3, which among other things has the nice side effect of not requiring any new fields in the task structure and not requiring changes to the scheduler.

Legal Statement

7. REFERENCES

[1] Desnoyers, M., McKenney, P. E., Stern, A., Dagenais, M. R., and Walpole, J. User-level implementations of read-copy update. Submitted to IEEE TPDS, December 2009.

[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. Available: `http://www.usenix.org/events/osdi99/full_papers/gamsa/gamsa.pdf` [Viewed August 30, 2006].

[3] Guniguntala, D., McKenney, P. E., Triplett, J., and Walpole, J. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal 47*, 2 (May 2008), 221–236. Available: `http://www.research.ibm.com/journal/sj/472/guniguntala.pdf` [Viewed April 24, 2008].

[4] Kung, H. T., and Lehman, Q. Concurrent maintenance of binary search trees. *ACM Transactions on Database Systems 5*, 3 (September 1980), 354–382. Available: `http://portal.acm.org/citation.cfm?id=320619&dl=GUIDE,` [Viewed December 3, 2007].

[5] Massalin, H. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, New York, NY, 1992.

[6] McKenney, P. E. The design of preemptible read-copy-update. Available: `http://lwn.net/Articles/253651/` [Viewed October 25, 2007], October 2007.

[7] McKenney, P. E. Priority-boosting RCU read-side critical sections. Available: `http://lwn.net/Articles/220677/` Revised: `http://www.rdrop.com/users/paulmck/RCU/RCUbooststate.2007.04.16a.pdf` [Viewed September 7, 2007], February 2007.

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

[9] 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].

[10] McKenney, P. E., Sarma, D., Molnar, I., and Bhattacharya, S. Extending RCU for realtime and embedded workloads. In *Ottawa Linux Symposium* (July 2006), pp. v2 123–138. Available: `http://www.linuxsymposium.org/2006/view_abstract.php?content_key=184` `http://www.rdrop.com/users/paulmck/RCU/OLSrtRCU.2006.08.11a.pdf` [Viewed January 1, 2007].

[11] McKenney, P. E., and Slingwine, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518. Available: `http://www.rdrop.com/users/paulmck/RCU/rclockpdcsproof.pdf` [Viewed December 3, 2007].

[12] McKenney, P. E., and Walpole, J. What is RCU, fundamentally? Available: `http://lwn.net/Articles/262464/` [Viewed December 27, 2007], December 2007.