Read Copy Update

Paul E. McKenney
Linux Technology Center
IBM Beaverton
pmckenne@us.ibm.com, http://www.rdrop.com/users/paulmck

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

Andrea Arcangeli
SuSE Labs
andrea@suse.de

Andi Kleen
SuSE Labs
ak@suse.de

Orran Krieger
IBM T. J. Watson Research Center
okrieg@us.ibm.com, http://www.eecg.toronto.edu/~okrieg

Rusty Russell
Linux Technology Center
IBM Canberra
rusty@rustcorp.com.au, prussell@au.ibm.com

Abstract

Read-copy update is a mechanism for constructing highly scalable algorithms for accessing and modifying read-mostly data structures, while avoiding cacheline bouncing, memory contention, and deadlocks that plague highly scalable operating system implementations. In particular, code that performs read-only accesses may be written without any locks, atomic instructions, or writes to shared cachelines, even in the face of concurrent updates. We reported on the basic mechanism last year, and have produced a number of Linux^{TM} patches implementing and exploiting read copy update.

This paper evaluates performance of a number of read copy update implementations for non-preemptive Linux kernels, and outlines a new implementation targeted to preemptive Linux kernels.

1 Introduction

The past year has seen much discussion of read-copy update and the design and coding of a number of read-copy-update implementations. These implementations make a number of different tradeoffs, and this paper takes a first step towards evaluating them.

Comparison of read-copy update to other concurrent update mechanisms has been done elsewhere [McK01b, Linder02a]. These comparisons have shown that read-copy update can greatly simplify and inprove performance of code accessing read-mostly linked-list data structures (such as FD

The views expressed in this paper are the authors' only, and should not be attributed to SuSE or IBM.

management tables and dcache data structures). Evaluation of read-copy update in other environments has shown that the read-copy update can also improve performance of code modifying linked-list data structures when there is a high system-wide aggregate update rate across all such data structures [McK98a].

Section 2 fills in some background on read-copy update. Section 3 gives an overview of the design choices of the Linux read-copy update non-preemptive implementations. Section 4 compares performance and complexity of these implementations, with emphasis on the grace-period latency that determines the incremental memory overhead compared to non-read-copy-update locking algorithms. Section 5 overviews the implementations, focusing on `call_rcu()`, scheduler instrumentation, and timer processing. Section 5 also describes how the *rcu* algorithm may be adapted to a preemptible kernel. Section 6 describes future plans, Appendix A provides implementation details, and Appendix B discusses memory ordering issues encountered when inserting into a read-copy-protected data structure.

2 Background

This section gives a brief overview of read-copy update, more details are available elsewhere [McK98a, McK01a, McK01b]. Section 2.1 contains a glossary of read-copy-update-related terms, Section 2.2 presents concepts, Section 2.3 presents the read-copy-update API, Section 2.4 describes the IP route cache patch that uses read-copy update, Section 2.5 describes the module race reduction patch that uses read-copy update, and Section 2.6 gives an overview of how read-copy update may be used in a preemptive kernel.

2.1 Glossary

Live Variable: A variable that might be accessed before it is modified, so that its current value has some possibility of influencing future execution state.

Dead Variable: A variable that will be modified before it is next accessed, so that its current value cannot possibly have any influence over future execution state.

Temporary Variable: A variable that is only live inside a critical section. One example is a auto variable used as a pointer while traversing a linked list.

Permanent Variable: A variable that is live outside of critical sections. One example would be the header for a linked list.[1]

Quiescent State: A point in the code where all of the current entity's temporary variables that were in use before a specified time are dead. In a non-preemptive Linux kernel, a context switch is a quiescent state for CPUs. In a preemptive Linux kernel, a voluntary context switch is a quiescent state, but for threads. In this paper, quiescent states are global events, as opposed to being associated with a specific data structure.

Grace Period: Time interval during which all entities (CPUs or tasks, as appropriate) pass through at least one quiescent state. Note that any time interval containing a grace period is itself a grace period.

The key point underlying read-copy update is that if you remove all permanent-variable references to a given item, then wait for a grace period to expire, there can be no remaining references to that item. The item can then be safely freed up. This process is described in more detail in the next section.

2.2 Concepts

Read-copy update allows lock-free read-only access to data structures that are being concurrently modified. The accessing code needs neither locks nor atomic instructions, and can often be written as if the data structure were unchanging, in a "CS 101" style. Read-copy update is typically applied to linked data structures where the read side code traverses links through the data structure in a single direction.

Without special action on the update side, the read side would be prone to races with deletions, as illustrated in Figure 1, which shows two tasks searching

[1] Yes, it is possible for the same variable to be temporary sometimes and permanent at other times. However, this can lead to confusion, so is not generally recommended.

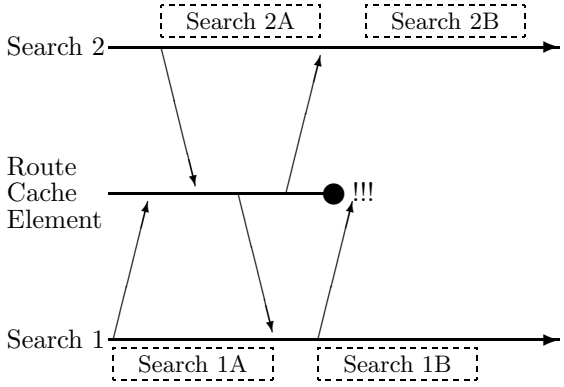

Figure 1: Race Between Deletion and Search

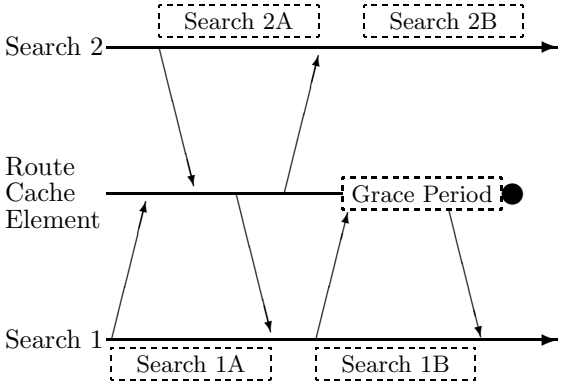

Figure 2: Read-Copy Update Handling Race

a list that contains an element that is concurrently deleted by a third task (signified by the line labelled "Route Cache Element"). To handle such race conditions, the update side uses a two-phase update discipline:

1. Remove permanent-variable pointers to the item being deleted.
2. After a grace period has elapsed, free up the item's memory.

The grace period is not a fixed time duration, but is instead inferred by checking for per-CPU quiescent states, such as context switches. Since kernel threads are prohibited from holding locks across a context switch, they also prohibited from holding pointers to data structures protected by those locks across context switches–after all, the entire data structure could well be deleted by some other CPU at any time the lock is not held.

Therefore, a simple implementation of read-copy update might declare the grace period over once it observed each CPU performing a context switch. Now, the first phase removed all global pointers to the item being deleted, and kernel threads are not permitted to hold references to the item across a context switch. Therefore, CPUs that have performed a context switch after the completion of the first phase have no way to gain a reference to the item being deleted. Thus, once all CPUs have performed a context switch, it is safe to free up the item being deleted from the list.

With this approach, searches already in progress when the first phase executes might (or might not)

```
void synchronize_kernel(void);
struct rcu_head {
        struct list_head list;
        void (*func)(void *obj);
        void *arg;
};
void call_rcu(struct rcu_head *head,
              void (*func)(void *arg),
              void *arg);
```

Figure 3: Read-Copy Update API

see the item being deleted. However, searches that start after the first phase completes are guaranteed to never reference this item. Therefore, the item may be safely freed once all searches in progress at the end of the first phase have completed, as shown in Figure 2.

Efficient mechanisms for determining the duration of the grace period are key to read-copy update.

2.3 Read-Copy Update API

Figure 3 shows the external API for read-copy update. The `synchronize_kernel()` function blocks for a full grace period. This is a simple, easy-to-use function, but imposes expensive context-switch overhead on its caller. It may not be called with locks held or from BH/IRQ context.

Another approach, taken by `call_rcu()` is to schedule a function to be called after the end of a full grace period. Since `call_rcu()` never sleeps, it may be called with locks held or from BH (and perhaps also IRQ) context. The `call_rcu()` function uses

```
1 void delete(struct el *p)
2 {
3    spin_lock(&list_lock);
4    p->next->prev = p->prev;
5    p->prev->next = p->next;
6    spin_unlock(&list_lock);
7    call_rcu(&p->my_rcu_head, my_free, p);
8 }
```

Figure 4: Read-Copy Dequeue From Doubly-Linked List

its `struct rcu_head` argument to store the specified callback function and argument, and the read-copy-update subsystem then uses this struct to schedule the callback invocation. An `rcu_head` is often placed within a structure being protected by read-copy update.

A typical use of `call_rcu` is shown in Figure 4, where an element is deleted from a circular doubly linked list with a header element. Here `my_free()` is a wrapper around `kfree()`, and the lock is used only to serialize concurrent calls to `delete()`. Since the element's `next` and `prev` pointers are unaffected, and since `my_free()` is not called until a grace period has elapsed, non-sleeping reading tasks may traverse the list concurrently with the deletion of the element without danger of a NULL pointer or a pointer to the freelist. This is a common read-copy-update idiom: `kfree()` is replaced by a `call_rcu()` to a function that is a wrapper around `kfree()`.

2.4 Read-Copy Update and IP Route Cache

Read-copy update has been used in a number of OSes, including several patches to Linux [McK01b, Linder02a]. This section describes how read-copy update may be used in the Linux IP route cache. This modification was done to validate the RCU implementations, rather than in response to a known performance problem in the IP route cache.

The Linux IP route cache uses a reader-writer lock, so multiple searches may proceed in parallel. However, the multiple readers' lock acquisitions result in the cacheline bouncing. Read-copy update may be used to eliminate this read side cacheline bouncing:

1. Delete all calls to `read_lock()`, `read_unlock()`, `read_lock_bh()`, and `read_unlock_bh()`.
2. Replace all calls to `write_lock()`, `write_unlock()`, `write_lock_bh()`, and `write_unlock_bh()` with the corresponding member of the `spin_lock()` family of primitives.
3. Add `rmb()` primitives on the read side between the fetch of the pointer and its dereferencing. These should be replaced by `read_barrier_depends()` when it becomes available.
4. Replace all calls to `dst_free()` with a call to `call_rcu()` which causes `dst_free()` to be invoked after the end of a following grace period, as shown in Figure 5.

```
1 @@ -314,13 +314,13 @@
2  static inline void rt_free(
3                       struct rtable *rt)
4  {
5 -    dst_free(&rt->u.dst);
6 +    call_rcu(&rt->u.dst.rcu_head,
7 +             (void (*)(void *))dst_free,
8 +              &rt->u.dst);
9  }
10
11  static inline void rt_drop(
12                       struct rtable *rt)
13  {
14      ip_rt_put(rt);
15 -    dst_free(&rt->u.dst);
16 +    call_rcu(&rt->u.dst.rcu_head,
17 +             (void (*)(void *))dst_free,
18 +              &rt->u.dst);
19  }
```

Figure 5: dst_free() Modifications

This results in a significant decrease in `ip_route_output_key()` overhead during a workload that transmits a fixed number of random-sized IP packets to a single destination, as shown in Figure 6. This workload was run on an 8-CPU 700MHz Pentium^{TM} III Xeon^{TM} with 1MB L2 cache and 6GB of memory.

Figure 7 shows the total non-idle kernel profile ticks for this same workload. This data shows the IP route cache speedup is real; it is not happening at the expense of other processing in the system. The overall speedup is quite small, as expected, given that the change was not motivated by a known per-

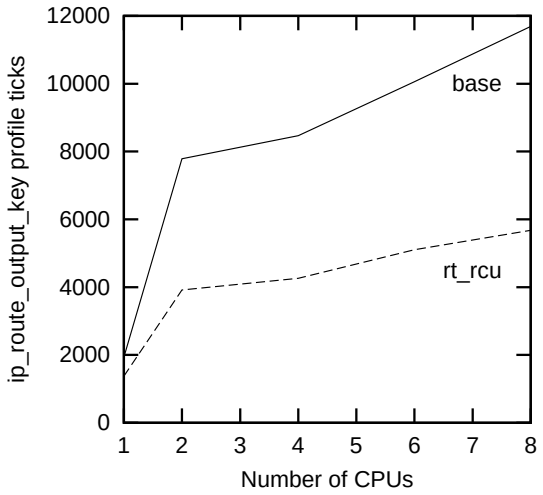

Figure 6: IP Route Cache Speedup Using rcu

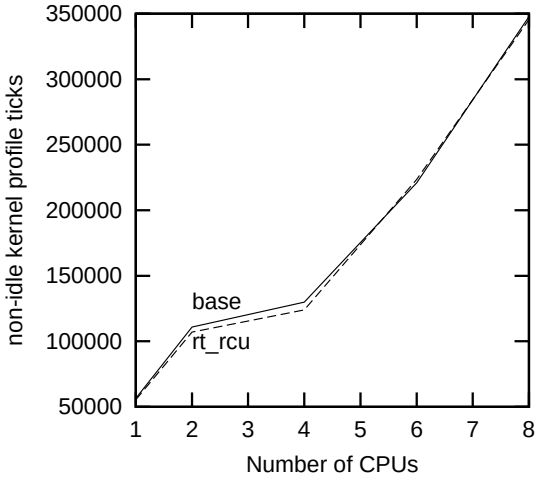

Figure 7: IP Route Cache System Performance Using rcu

formance problem.[2] More compelling Linux-based read-copy-update results include a 30% improvement for FD management [McK01b] and a 25% improvement for dcache management [Blanchard02a, Linder02a]

2.5 Read-Copy Update and Module Race Reduction

Linux 2.4 is subject to races between module unloading and use of that module. These races can result in the racing code that is attempting to use the module holding a reference to newly freed memory, most likely resulting in an "oops".

One way to reduce the likelihood of these races occurring is to wait for a grace period after removing the module structure from the `module_list` before `kfree()`ing it in `free_module()` [Kleen02a]. Races can still occur, but the race's window has been decreased substantially. The change is a one-liner (not counting comments), as shown in Figure 8.

As noted earlier, this change does not address all the module-unloading problems. However, we hope that it can be a basis for a full solution. This approach is now being used in production in SuSE Linux.

2.6 Read-Copy Update and Preemption

Preemption has recently been added to Linux in 2.5.4. The addition of preemption means that read side kernel code is subject to involuntary context switches. If not taken into account, this leads to premature flagging of the ends of grace periods. There are two ways to handle preemption: (1) explicitly disabling preemption over read side code segments, and (2) considering only *voluntary* context switches to be quiescent states.

Explicitly disabling preemption over read side code segments adds unwanted overhead to reading processes, and removes some of the latency benefits provided by preemption. In contrast, considering only voluntary context switches to be quiescent states allows the kernel to reap the full benefit of reduced latency. This scheme for tracking only voluntary context switches is inspired by the K42 implementation [Gamsa99].[3] The main drawback is increased length of grace periods. This paper focuses on the voluntary context switches option and its effects.

[2]However, we will be measuring this patch on various workloads as Linux's scaling continues to improve.

3 Read-Copy Update Implementations

As noted earlier, the key to read-copy update is a CPU-efficient mechanism for determining the required duration of the grace period. This mechanism is permitted to overestimate the grace-period duration, but the greater the overestimation, the greater the amount of memory that will be consumed by waiting callbacks. There are a number of simple and efficient algorithms to determine grace-period duration, and this paper reviews a number of them.

There are a number of design parameters for a read-copy update implementation:

1. Batching. Many implementations batch requests, so that a single grace-period identification can satisfy multiple requests. Batching is particularly important for implementations with heavyweight grace period identification mechanisms. Although there have been implementations without batching [McK01a], all implementations described in this paper do batching.

2. Deducing the length of the grace period. The simplest mechanisms force a grace period by a reschedule on all CPUs in non-preemptive kernels. However, this approach is relatively expensive, particularly if extended to cope with preemptible kernels. More efficient implementations use something like per-CPU quiescent-state counters to deduce when the natural course of events has resulted in the expiration of a grace period.

3. Polling mechanism. Implementations that deduce when a grace period has ended must use some mechanism to be informed of this event:

[3]K42's extensive use of blocking locks and short-lived threads results in use of thread termination rather than voluntary context switch as the K42 quiescent state. In addition, Linux migrates preempted tasks to other CPUs, which requires special tracking of tasks that have been preempted since their last voluntary context switch.

(a) Adding explicit checks to code corresponding to quiescent states, for example, *rcu-sched*'s hooks in the Linux scheduler shown in Figure 29. Explicit checks allow fast response to quiescent states, but add overhead when there are no read-copy callbacks in flight.

(b) Adding counters to code corresponding to quiescent states, and using kernel daemons to check the counters, as shown in Figure 13. This approach adds some complexity, but greatly reduces the overhead when there are no read-copy callbacks in flight.

(c) As above, but use tasklets instead of kernel daemons to do the checking. This further reduces the overhead, but uses more exotic features of Linux.

(d) As above, but use a per-CPU timer handler [Sarma02a] instead of tasklets to do the checking. It is not yet clear which of tasklets and timer handlers are preferable.

If the implementation forces the end of the grace period, it must similarly use a mechanism for doing so:

(a) Scheduling a thread on each CPU in turn. This has the advantage of immediacy, but cannot be used from BH or IRQ, and gains no performance benefit from batching.

(b) Reserving a kernel daemon that, upon request, schedules itself on each CPU in turn. This permits batching and use from BH and IRQ, but is more complex.

4. Request queuing. Requests may be queued globally or on a per-CPU basis. Grace periods must of course always be detected globally, but per-CPU queuing can reduce the CPU overhead incurred by `call_rcu()`. This is a classic performance/complexity tradeoff. The correct choice depends on the workload.

5. Quiescent state definition. For non-preemptive kernels, context switch is a popular choice. For preemptive Linux kernels (such as Linux 2.5), voluntary context switch may instead be used.

6. Environments. If `call_rcu()` use is prohibited in the BH or IRQ contexts, then more kernel functionality is available to the implementor of `call_rcu()`, and less overhead is incurred.

```
 1 @@ -1065,6 +1066,12 @@
 2           p->next = mod->next;
 3       }
 4       spin_unlock_irqrestore(&modlist_lock,
 5                              flags);
 6
 7 +     /* Wait for all other cpus to go
 8 +      * through a context switch. This
 9 +      * doesn't plug all module unload
10 +      * races, but at least some of
11 +      * them and makes the window much
12 +      * smaller.
13 +      */
14 +     synchronize_kernel();
15
16           /* And free the memory.  */
```

Figure 8: Module Unloading

Section 5 describes a number of Linux implementations of read-copy update, summarized in Table 1.

All the implementations in Table 1 except *rcu-preempt* assume a run-to-block kernel. Section 5.7 describes *rcu-preempt*, which operates efficiently in a preemptive kernel.

The "QS" column lists the quiescent states that each algorithm tracks, "I" for idle-loop execution, "C" for context switch, and "U" for user-mode execution.

The "BH/IRQ Safe" column indicates whether code running in BH/IRQ context may safely delete elements of a read-copy-update-protected data structure that is accessed by base-level code with interrupts enabled. The *rcu-poll* implementation is BH safe, but is IRQ unsafe by choice, in order to eliminate the overhead of interrupt disabling and enabling that would otherwise be incurred on each call to `call_rcu()`. If a strong need arises for use of `call_rcu()` from IRQ context, trivial changes to *rcu-poll* will render it IRQ safe.

The read-copy-update implementations discussed in this paper choose different points in this design space. These implementations are freely available [LSE]. The *X-rcu*, *rcu*, and *rcu-ltimer* implementations are similar to the ptx^{TM} implementation, using per-CPU timers, kernel daemons, and architecture-dependent timer support, respectively. The *rcu-taskq* implementation is an extremely compact implementation in which a kernel task forces per-CPU kernel daemons to run on their respective CPUs. The *rcu-sched* implementation uses ring counters within the Linux scheduler, and boasts an extremely low overhead `call_rcu()` implementation. The *rcu-poll* implementation is designed for minimal overhead when there are no outstanding read-copy callbacks, and boasts very low `call_rcu()` latencies. Finally, the *rcu-preempt* implementation adapts the *rcu* implementation to work correctly in preemptible kernels. We will adapt some of the other implementations for preemptible use, as well. These implementations are described in more detail in Section 5 and Appendix A.

4 Performance and Complexity Comparisons

Table 2 shows the amount of overhead incurred by each implementation when there is no read-copy update activity in the system. The *rcu-taskq* implementation does best by this measure, with absolutely no overhead. The *rcu-poll* and *rcu-preempt* are next, with but a single local non-atomic increment in the scheduler. The *rcu-preempt* also incurs overhead on each preemption, as *rcu-poll* likely will once it is adapted to run in a preemptive kernel. The other implementations incur timer overhead under idle conditions.

An important figure of merit for a read-copy-update implementation is the grace period latency. The greater the latency, the more memory is waiting on the internal lists for the current grace period to end. On the other hand, longer latency results in higher efficiency, since the per-callback-batch processing is done less frequently, spreading the overhead over more `call_rcu()` requests. The best tradeoff depends on the workload: systems with very infrequent `call_rcu()` invocations would prefer small latency in order to conserve memory, while systems with very frequent `call_rcu()` invocations would prefer larger latencies in order to amortize the overhead of detecting a grace period over more `call_rcu()` invocations.

This latency depends on worst-case kernel code-path length, the workload, and the details of the read-copy-update implementation. Figure 9 shows the `call_rcu()` latency for the different read-copy update algorithms as a function of offered load to the dbench benchmark. It was run on an 8-CPU 700MHz Xeon system with 1MB L2 caches and 6GB of memory using the dcache-rcu patch [LSE]. The winner by far is *rcu-poll*, which keeps latencies below 10 milliseconds (and below 250 *microseconds* on an idle system) by allowing quiescent states to be detected in parallel and by its aggressive forcing of scheduling when a grace period is required (see Figure 10, which shows the same data on a semilog plot). Therefore, *rcu-poll* is preferable on systems that invoke `call_rcu()` infrequently. The *X-rcu*, *rcu-ltimer*, and *rcu* implementations have larger latencies that are well bounded as the number of clients increase. These algorithms are thus preferable on systems that have very high rates of `call_rcu()` invocation.

Name	Batch?	Deduce	Poll	Queuing	QS	BH/IRQ Safe
X-rcu	Yes	counters	timers	per-CPU	IC	Yes
rcu	Yes	counters	daemons	per-CPU	C	Yes
rcu-poll	Yes	counters	tasklet	global	C	BH Only
rcu-ltimer	Yes	counters	tasklets	per-CPU	IUC	Yes
rcu-taskq	Yes	No	daemons	global	C	Yes
rcu-sched	Yes	counter ring	N/A	per-CPU-rrupt	IC	Yes
rcu-preempt	Yes	counters	timers	per-CPU	IC	Yes

Table 1: Read-Copy Implementations

	RCU Idle Memory Refs			
Name	Switch	Preempt	Timer	Timer Type
X-rcu	1 local		8 local + 1 global + 1 timer	50ms per CPU
rcu	1 local		2 local + 1 global read + 1 global write + 1 timer + #CPU * up()	50ms global
rcu-poll	1 local			
rcu-ltimer	1 local		7 local + 1 global + 1 tasklet	per CPU
rcu-taskq				
rcu-sched	1 global read			
rcu-preempt	1 local	6 local		

Table 2: Read-Copy Idle Overhead

The *rcu-sched* algorithm exhibited very large latencies (14.5 seconds at 8 clients and 57.7 seconds at 4 clients), which we are investigating. The *rcu-taskq* algorithm's latencies increases with increasing numbers of clients, because this algorithm requires the CPUs to pass through quiescent states sequentially, and because keventd (which runs the taskq's) runs at low priority.

Read-copy update can pose a tradeoff between latency and overhead, since increased latency increases the number of callbacks that are serviced by a single grace period. To evaluate this tradeoff, Figure 11 compares the performance of the chat benchmark with 20 rooms and 500 messages on a 4-CPU 700MHz Pentium III Xeon system with 1MB L2 caches and 1GB memory. This benchmark was run using the read-copy-update-based IP-route-cache and FD management patches [LSE]. These results show little sensitivity to the read-copy-update algorithm. We are collecting more data on other workloads.

Table 3 shows the number of lines in each algorithm's patch. The "All Archs" column gives the size of the patch applied to all architectures currently in the kernel, while the "One Arch" column gives the size of each patch applied to only one architecture. Architecture-independent patches will have the same number in both columns. The *rcu-taskq* implementation is the simplest, and so might be a good place to start looking at read-copy-update implementations.

The *rcu-ltimer* patch works only on the i386 architecture, so the figure for "All Archs" is an estimate based on the i386-specific portion of the patch, which simply invokes `RCU_PROCESS_CALLBACKS()` from the `smp_local_timer_interrupt()` function. The *rcu-sched* patch contains code to guard against architectures that shut down their CPUs when idle.

	Size of Unified Diffs	
Name	All Archs	One Arch
rcu-taskq-2.5.3-1.patch	237	237
rcu-poll-2.5.3-1.patch	378	378
X-rcu-2.5.3-4.patch	424	424
rcu-sched-2.5.3-1.patch	575	333
rcu-2.5.3-2.patch	603	603
rcu-preempt-2.5.8-3.patch	682	682
rcu-ltimer-2.5.3-1.patch	*742*	514

Table 3: Read-Copy Implementation Complexity

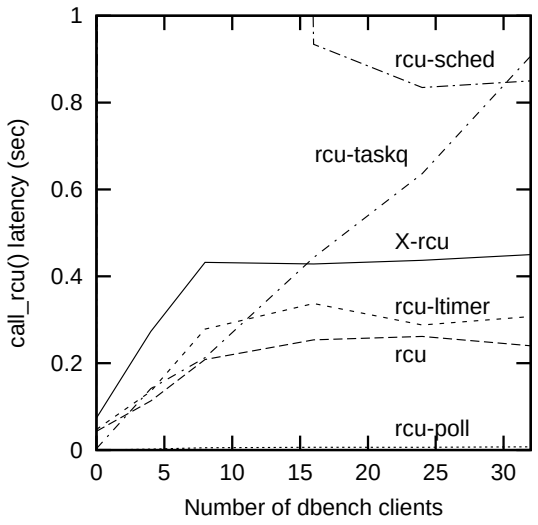

Figure 9: call_rcu() Latency Under dbench Load

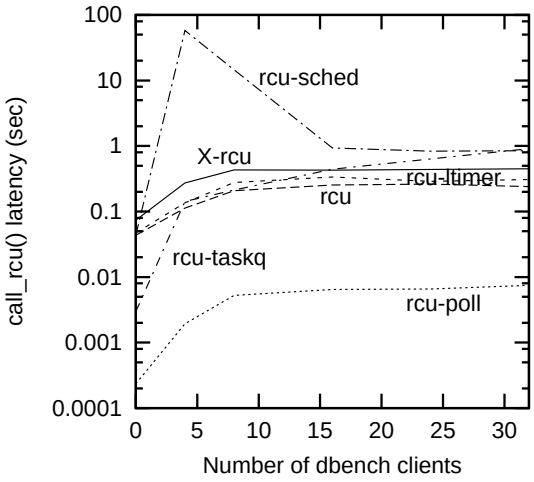

Figure 10: call_rcu() Latency Under dbench Load (logscale)

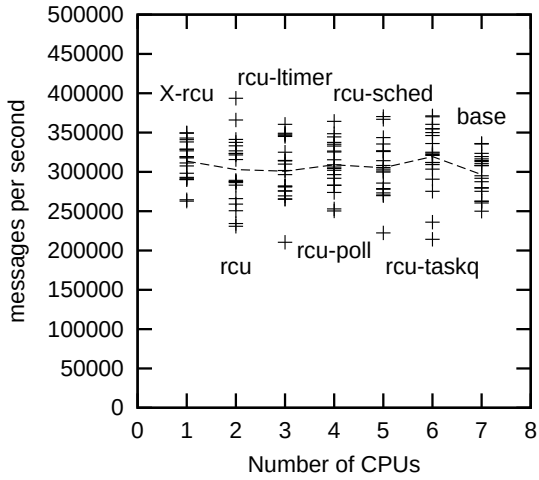

Figure 11: RCU Performance on Chat Benchmark

5 Read-Copy Update Implementation Overviews

The following sections summarize the `call_rcu()` implementation, the quiescent-state instrumentation (usually in the scheduler), and the high-level timer processing. More details on the more-complex implementations may be found in Appendix A, and patches for each may be found on the Linux Scalability Effort website [LSE].

5.1 X-rcu

X-rcu is loosely based on the ptx read-copy-update implementation. It uses a per-CPU context switch counter to instrument this quiescent state, uses per-CPU queues to track callbacks, and per-CPU timers to track quiescent states as needed to find the end of grace periods. The timers further check for running from idle, which is a second quiescent state. Dipankar Sarma implemented this variant to evaluate the use of timers rather than the kernel daemons or architecture-dependent timer hooks used by the *rcu* and *rcu-ltimer* implementations.

This implementation depends on patches that have not yet appeared in 2.4, 2.5, or both. The required patches include:

1. Rusty Russell's per-CPU data area patch [Russell02a] permits more natural maintenance of per-CPU data. It permits the context switch counter to be maintained separately from the rest of the per-CPU state, which avoids some nasty header file cyclic dependencies between *interrupt.h*, *fs.h*, and *sched.h*. This separation means that *rcupdate.h* need not include *interrupt.h*, which makes it easier to include *rcupdate.h* in lower-level kernel subsystems, such as dcache. This patch recently was accepted into the Linux 2.5 kernel.

2. Per-CPU timer support [Sarma02a]. This patch enhances Ingo Molnar's smptimers patch to guarantee that timers queued in a CPU always get executed on the same CPU where they were enqueued. This guarantee allows per-CPU quiescent state checking to be performed in a clean and architecture independent way. In addition, timers have significantly lower overhead than kernel daemons.

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(void *arg),
 3               void *arg)
 4 {
 5         unsigned long flags;
 6
 7         head->func = func;
 8         head->arg = arg;
 9         local_irq_save(flags);
10         list_add_tail(&head->list,
11                &this_cpu(rcu_nextlist));
12         local_irq_restore(flags);
13 }
```

Figure 12: *X-rcu* call_rcu() Implementation

```
1 @@ -685,6 +686,7 @@
2  switch_tasks:
3         prefetch(next);
4         prev->work.need_resched = 0;
5 +       per_cpu(rcu_qsctr, prev->cpu)++;
6
7         if (likely(prev != next)) {
8                 rq->nr_switches++;
```

Figure 13: *X-rcu* Scheduler Instrumentation

The `call_rcu()` function constructs the callback and enqueues it onto the current CPU's `rcu_nextlist`, as shown in Figure 12.

Figure 13 shows how the scheduler is instrumented. The added line 5 compiles to a local increment, with no locking, atomic operations, or cacheline bouncing.

Figure 14 shows the processing done by the per-CPU timer handler, currently set up to execute every 5 jiffies on each CPU. This code detects idle-loop execution and counts this as a quiescent state. It then invokes `rcu_process_callbacks()` to advance callbacks as ends of grace periods are detected. This callback advancement is described in Appendix A.1.

```
1 static void rcu_percpu_tick(void)
2 {
3         /* Check for idle loop */
4         if (task_idle(current))
5                 this_cpu(rcu_qsctr)++;
6         rcu_process_callbacks();
7 }
```

Figure 14: *X-rcu* Timer Processing

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(void *arg),
 3               void *arg)
 4 {
 5     int cpu = cpu_number_map(
 6                 smp_processor_id());
 7     unsigned long flags;
 8
 9     head->func = func;
10     head->arg = arg;
11     local_irq_save(flags);
12     list_add_tail(&head->list,
13                   &RCU_nxtlist(cpu));
14     local_irq_restore(flags);
15     tasklet_schedule(&RCU_tasklet(cpu));
16 }
```

Figure 15: *rcu* call_rcu() Implementation

5.2 rcu

The *rcu* patch is also based on the ptx algorithm. Unlike the *X-rcu* patch described in Section 5.1, *rcu* has minimal dependencies on other patches. It is otherwise quite similar, using per-CPU queues of callbacks and context-switch counters instrumenting the quiescent states. However, it uses per-CPU kernel daemons to periodically check for the end of grace periods, which means that it cannot easily check for the CPU having been idle. These daemons are awakened by a timer that is scheduled only when there is at least one callback in the system. Dipankar Sarma implemented this variant to evaluate use of kernel daemons rather than architecture-dependent timer hooks.

The `call_rcu()` function simply constructs the callback, enqueues it onto the current CPU's `RCU_nxtlist`, then schedules the current CPU's tasklet, as shown in Figure 15.

The scheduler is instrumented as shown in Figure 16. As with *X-rcu*, this is a local increment without locking, atomic instructions, or cacheline bouncing, but, due to the lack of a per-CPU data area, array-indexing instructions are required.

The code that performs periodic RCU processing is shown in Figure 17. UP kernels invoke it directly from the timeout handler, while SMP kernels invoke it from *krcud* daemons that are awakened by the timeout handler.

```
1 @@ -685,6 +686,7 @@
2  switch_tasks:
3   prefetch(next);
4   prev->work.need_resched = 0;
5 +  RCU_qsctr(prev->cpu)++;
6
7   if (likely(prev != next)) {
8   rq->nr_switches++;
```

Figure 16: *rcu* Scheduler Instrumentation

```
1 static void rcu_percpu_tick_common(void)
2 {
3     rcu_process_callbacks(0);
4 }
```

Figure 17: *rcu* Timer Processing

Details of *rcu*'s callback processing are discussed in Appendix A.2.

5.3 rcu-poll

The *rcu-poll* algorithm was written by Andrea Arcangeli and Dipankar Sarma. It appears in the "-aa" series of kernels and in recent SuSE releases. Unlike the *X-rcu* and *rcu* algorithms, *rcu-poll* uses a single set of lists to process read-copy-update callbacks, which are processed by a single tasklet. This results in more cacheline bouncing than do the other algorithms, but is considerably shorter and simpler, and, as noted earlier, boasts extremely short average grace-period latencies and low incremental overheads when there are no read-copy update callbacks in flight.

The `call_rcu()` function constructs the callback, enqueues it onto a global `rcu_nxtlist`, then schedules the tasklet, as shown in Figure 18.

The scheduler is instrumented in much the same way as for the previous algorithms, as shown in Figure 19.

Periodic RCU processing is handled by a single tasklet, whose body is shown in Figure 20. This tasklet invokes `rcu_prepare_polling()` to snapshot each CPU's quiescent state counters if polling is not yet in progress and if there are pending callbacks. If polling has already been started, it instead invokes `rcu_polling()` to check to see if the grace period has ended. This ensures all CPUs have

```
1 void call_rcu(struct rcu_head *head,
2               void (*func)(void *arg),
3               void *arg)
4 {
5     head->func = func;
6     head->arg = arg;
7
8     spin_lock_bh(&rcu_lock);
9     list_add(&head->list, &rcu_nxtlist);
10    spin_unlock_bh(&rcu_lock);
11
12    tasklet_hi_schedule(&rcu_tasklet);
13 }
```

Figure 18: *rcu-poll* call_rcu() Implementation

```
1 @@ -685,6 +686,7 @@
2  switch_tasks:
3          prefetch(next);
4          prev->work.need_resched = 0;
5 +        RCU_quiescent(prev->cpu)++;
6
7          if (likely(prev != next)) {
8                  rq->nr_switches++;
```

Figure 19: *rcu-poll* Scheduler Instrumentation

passed through their quiescent states via the context switch.

Details of *rcu-poll*'s callback processing are discussed in Appendix A.3.

5.4 rcu-ltimer

The *rcu-ltimer* implementation is similar to *X-rcu* and *rcu*, but it inserts calls to `RCU_PROCESS_CALLBACKS()` into `do_timer()` and into the architecture-specific `smp_local_timer_interrupt()` functions, instead of using timers or a kernel daemon to check for the ends of grace periods. This allows *rcu-ltimer* to count user-mode execution as a quiescent state, in addition to the idle loop and context switch. The current patch is fully implemented only on the i386 architecture. Dipankar Sarma implemented this variant to obtain the closest analog to the ptx implementation.

The `call_rcu()` function constructs the callback and enqueues it onto a per-CPU `RCU_nxtlist`, as shown in Figure 21.

```
 1 static void rcu_process_callbacks(
 2                   unsigned long data)
 3 {
 4     int stop;
 5
 6     spin_lock(&rcu_lock);
 7     if (!rcu_polling_in_progress)
 8         stop = rcu_prepare_polling();
 8     else
 9         stop = rcu_polling();
10     spin_unlock(&rcu_lock);
11
12     if (!stop)
13         tasklet_hi_schedule(&rcu_tasklet);
14 }
```

Figure 20: *rcu-poll* Tasklet Body

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(void *arg),
 3               void *arg)
 4 {
 5     int cpu = cpu_number_map(
 6                  smp_processor_id());
 7
 8     head->func = func;
 9     head->arg = arg;
10     local_bh_disable();
11     list_add_tail(&head->list,
12                   &RCU_nxtlist(cpu));
13     local_bh_enable();
14 }
```

Figure 21: *rcu-ltimer* call_rcu() Implementation

```
 1 @@ -685,6 +686,7 @@
 2  switch_tasks:
 3          prefetch(next);
 4          prev->work.need_resched = 0;
 5 +        RCU_qsctr(prev->cpu)++;
 6
 7          if (likely(prev != next)) {
 8                  rq->nr_switches++;
```

Figure 22: *rcu-ltimer* Scheduler Instrumentation

```
 1 #define RCU_PROCESS_CALLBACKS(cpu,regs) \
 2   do { \
 3      if (user_mode(regs) || idle_cpu(cpu)) \
 4          RCU_qsctr(cpu)++; \
 5          if ((RCU_tasklet(cpu).state & \
 6              ((1 << TASKLET_STATE_SCHED) | \
 7                (1 << TASKLET_STATE_RUN))) \
 8                    == 0) \
 9              tasklet_schedule(
10                  &RCU_tasklet(cpu)); \
11   } while(0)
```

Figure 23: *rcu-ltimer* Timer Processing

The scheduler is instrumented in much the same way as for the previous algorithms, as shown in Figure 22.

Periodic RCU processing is handled by per-CPU tasklets, which are invoked as shown in Figure 23. Lines 3-4 note a quiescent state if the CPU was interrupted from user mode or the idle loop. Lines 5-10 schedule this CPU's tasklet if it is not already either scheduled or running. This tasklet invokes `rcu_process_callbacks()`, which is described in more detail in Appendix A.4.

5.5 rcu-taskq

Dipankar Sarma implemented the *rcu-taskq* algorithm to obtain a minimal efficient implementation. And this implementation does in fact have the smallest patch, using a single task and a global set of callback queues. The task forces each of a set of per-CPU kernel daemons to schedule itself; when each done so, the grace period has expired. This implementation thus directly forces quiescent states, unlike the other implementations, which instead measure naturally occurring quiescent states. Its grace-period latency increases with increasing load on the system, as noted earlier, but is the only implemen-

```
 1 void call_rcu(struct rcu_head * head,
 2               void (*func)(void * arg),
 3               void * arg)
 4 {
 5     unsigned long flags;
 6     int start = 0;
 7
 8     head->func = func;
 9     head->arg = arg;
10
11     spin_lock_irqsave(&rcu_lock, flags);
12     if (list_empty(&rcu_wait_list))
13         start = 1;
14     list_add(&head->list, &rcu_wait_list);
15     spin_unlock_irqrestore(&rcu_lock, flags);
16
17     if (start)
18         schedule_task(&rcu_task);
19 }
```

Figure 24: *rcu-taskq* call_rcu() Implementation

tation with absolutely zero load on the system when there are no read-copy callbacks in flight.

Figure 24 shows the `call_rcu()` implementation. Lines 8-9 initialize the callback, lines 11 and 15 handle locking, lines 12-13 record the initial list state, and line 14 adds the callback to the `rcu_wait_list`. Lines 17-18 start the task if lines 12-13 found the list initially empty.

The task started by `call_rcu()` invokes the function `process_pending_rcus()`, shown in Figure 25. Lines 8-10 snapshot `rcu_wait_list` into a local list. Line 13 then invokes `wait_for_rcu()` to wait for a full grace period to elapse. Finally, lines 15-23 invoke the callbacks from the local list.

Figure 26 shows `wait_for_rcu()`. Lines 6-10 awaken the *krcud* daemons for the other CPUs, and lines 11-13 wait for these daemons to respond.

Figure 27 shows the code for the *krcud* daemons. Lines 6-20 initialize the daemon, set its priority high, blocking signals, binding to the corresponding CPU, setting the task name, initializing the task name, and informing the `spawn_krcud()` task that the daemon is ready to process requests. Lines 22-26 process each request, alternately sleeping on the `krcud_sema` and waking up the `process_pending_rcus()` task.

```
 1 static void process_pending_rcus(
 2                   void *arg)
 3 {
 4     LIST_HEAD(rcu_current_list);
 5     struct list_head * entry;
 6
 7     spin_lock_irq(&rcu_lock);
 8     list_splice(&rcu_wait_list,
 9                 rcu_current_list.prev);
10     INIT_LIST_HEAD(&rcu_wait_list);
11     spin_unlock_irq(&rcu_lock);
12
13     wait_for_rcu();
14
15     while ((entry = rcu_current_list.prev)
16               != &rcu_current_list) {
17         struct rcu_head * head;
18
19         list_del(entry);
20         head = list_entry(entry,
21                   struct rcu_head, list);
22         head->func(head->arg);
23     }
24 }
```

Figure 25: *rcu-taskq* process_pending_rcus() Implementation

```
 1 static void wait_for_rcu(void)
 2 {
 3     int cpu;
 4     int count;
 5
 6     for (cpu = 0; cpu < smp_num_cpus; cpu++) {
 7         if (cpu == smp_processor_id())
 8             continue;
 9         up(&krcud_sema(cpu));
10     }
11     count = 0;
12     while (count++ < smp_num_cpus - 1)
13         down(&rcu_sema);
14 }
```

Figure 26: *rcu-taskq* wait_for_rcu() Implementation

```
 1 static int krcud(void * __bind_cpu)
 2 {
 3     int bind_cpu = *(int *) __bind_cpu;
 4     int cpu = cpu_logical_map(bind_cpu);
 5
 6     daemonize();
 7     current->policy = SCHED_FIFO;
 8     current->rt_priority = 1001 +
 9       sys_sched_get_priority_max(SCHED_FIFO);
10
11     sigfillset(&current->blocked);
12
13     /* Migrate to the right CPU */
14     set_cpus_allowed(current, 1UL << cpu);
15
16     sprintf(current->comm,
17             "krcud_CPU%d", bind_cpu);
18     sema_init(&krcud_sema(cpu), 0);
19
20     krcud_task(cpu) = current;
21
22     for (;;) {
23         while (down_interruptible(
24                  &krcud_sema(cpu)));
25         up(&rcu_sema);
26     }
27 }
```

Figure 27: *rcu-taskq* krcud() Implementation

5.6 rcu-sched

The *rcu-sched* implementation was developed by Rusty Russell [Russell01d], with a goal of minimizing `call_rcu()` overhead. It uses a ring of per-CPU counters, and each CPU sets its counter to one greater than that of its neighbor on each pass through the scheduler when read-copy-update callbacks are pending. Thus, when a given CPU sees its neighbor's counter change, it is guaranteed that each CPU has passed through the scheduler (a quiescent state) since the given CPU last incremented its own counter.

This implementation also maintains not just per-CPU callback queues, but two sets of per-CPU-per-IRQ callback queues. This allows the queues to be accesses without the need for either locks (per-CPU) or for interrupt masking (per-IRQ). One set of these queues accumulates new callbacks from `call_rcu()`, while the other set holds callbacks waiting for the end of the current grace period.

Finally, this implementation places checks in the idle loop in order to ensure that idle CPUs do not indefinitely delay the end of the grace period. This has the beneficial side effect of causing idle-loop execution to be a quiescent state without using the active entities (tasklets, timers, kernel daemons) used by the other implementations.

Figure 28 shows the `call_rcu()` function. Lines 9-10 initialize the `rcu_head` callback. Lines 11-14 determine the interrupt state, which is used later as an index to the array of lists of callbacks. Lines 17-18 find the right queue for the callback. The `rcu_batch[cpu].queueing` is a bit that contains the index of the half of the array that is accumulating new callbacks. The sense of this bit is reversed in `rcu_batch_done()` at the end of each grace period. Line 20 increments the number of pending callbacks, which signals the scheduler to start looking for a grace period, and lines 23-24 enqueues the callback.

Figure 29 shows the first patch to the scheduler. Lines 12-13 check to see if there are read-copy-update callbacks pending, and, if so, branch to the `rcu_process` label in the second patch shown in Figure 30

Lines 8-10 of Figure 30 set local variable `c` to one greater than the previous CPU's ring counter. If

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(void *data),
 3               void *data)
 4 {
 5     unsigned cpu = smp_processor_id();
 6     unsigned state;
 7     struct rcu_head **headp;
 8
 9     head->func = func;
10     head->data = data;
11     if (in_interrupt()) {
12         if (in_irq()) state = 2;
13         else state = 1;
14     } else state = 0;
15
16     /* Figure out which queue we're on. */
17     headp = &rcu_batch[cpu].head[
18              rcu_batch[cpu].queueing][state];
19
20     atomic_inc(&rcu_pending);
21     /* Prepend to this CPU's list:
22        no locks needed. */
23     head->next = *headp;
24     *headp = head;
25 }
```

Figure 28: *rcu-sched* call_rcu() Implementation

```
 1 @@ -634,10 +639,16 @@
 2      prio_array_t *array;
 3      list_t *queue;
 4      int idx;
 5 +    int c, this_cpu;
 6
 7      if (unlikely(in_interrupt()))
 8          BUG();
 9      release_kernel_lock(prev,
10              smp_processor_id());
11 +
12 +    if (unlikely(is_rcu_pending()))
13 +        goto rcu_process;
14 +
15 +rcu_process_back:
16      spin_lock_irq(&rq->lock);
17
18          switch (prev->state) {
```

Figure 29: *rcu-sched* Scheduler Instrumentation, Part 1

```
 1 @@ -700,6 +711,23 @@
 2      }
 3      spin_unlock_irq(&rq->lock);
 4
 5 +rcu_process:
 6 +    /* Avoid cache line effects
 7 +       if value hasn't changed */
 8 +    this_cpu = smp_processor_id();
 9 +    c = ring_count((this_cpu + 1) %
10 +           smp_num_cpus) + 1;
11 +    if (c != ring_count(this_cpu)) {
12 +        /* Do subtraction to
13 +           avoid int wrap corner case */
14 +        if (c - finished_count(this_cpu)
15 +                                >= 0) {
16 +            /* Avoid reentry: temporarily
17 +               set finish_count
18 +               far in the future */
19 +            finished_count(this_cpu) =
20 +                           c + INT_MAX;
21 +            rcu_batch_done();
22 +            finished_count(this_cpu) =
23 +                           c + smp_num_cpus;
24 +        }
25 +        ring_count(this_cpu) = c;
26 +    }
27 +    goto rcu_process_back;
28 +
29      reacquire_kernel_lock(current);
30      return;
31  }
```

Figure 30: *rcu-sched* Scheduler Instrumentation, Part 2

`c` is different than this CPU's ring count, a grace period has ended, and is handled by lines 16-23. Line 11 checks for scheduler reentry, and if this has not occurred, lines 19-23 invoke `rcu_batch_done()`, protecting against scheduler re-entry by manipulating this CPU's `finished_count`. Line 25 updates this CPU's ring count, which will result in the next CPU seeing the end of a grace period. Line 27 returns control to the mainline scheduler.

Figure 31 shows how the idle loop is instrumented to prevent architectures that shut down CPUs on idle from indefinitely extending the grace period. The other implementations get this effect through use of timers or forced context switches.

Figure 32 shows `rcu_batch_done()`, which is invoked from the scheduler at the end of a grace period. Line 7-8 pick up a pointer to this CPU's set of read-copy-update callback queues. Lines 11-22 invoke all the callbacks in each of this CPU's

```
 1 @@ -84,7 +85,8 @@
 2     get into the scheduler unnecessarily.  */
 3     long oldval = xchg(
 4           &current->work.need_resched, -1UL);
 5     if (!oldval)
 6 -       while (current->work.need_resched < 0);
 7 +       while (current->work.need_resched < 0
 8 +              && !is_rcu_pending());
 9     schedule();
10     check_pgt_cache();
11 }
```

Figure 31: *rcu-sched* Idle Loop Instrumentation

callback queues (one for each possible IRQ level) that was waiting for the current grace period to expire (selected by `!mybatch->queueing`), and empty each list. Line 25 swaps the sets of queues, so that the callbacks previously waiting for a new grace period to begin are now waiting for the now-current grace period, and the newly emptied queues will now accept new callbacks registered by future calls to `call_rcu()`.

```
 1 void rcu_batch_done(void)
 2 {
 3     struct rcu_head *i, *next;
 4     struct rcu_batch *mybatch;
 5     unsigned int q;
 6
 7     mybatch =
 8         &rcu_batch[smp_processor_id()];
 9     /* Call callbacks: probably delete
10        themselves, may schedule. */
11     for (q = 0; q < 3; q++) {
12         for (i = mybatch->head[
13                    !mybatch->queueing][q];
14              i;
15              i = next) {
16             next = i->next;
17             i->func(i->data);
18             atomic_dec(&rcu_pending);
19         }
20         mybatch->head[
21            !mybatch->queueing][q] = NULL;
22     }
23
24     /* Start queueing on this batch. */
25     mybatch->queueing = !mybatch->queueing;
26 }
```

Figure 32: *rcu-sched* rcu_batch_done()

5.7 Preemptible Algorithm

With the addition of preemption to the Linux kernel, read-copy update must also handle preemption. Rusty Russell [Russell01b] produced such a patch, but it requires scanning all tasks on the runqueue, a job made more complex by the addition of the multi-queue scheduler.

Dipankar Sarma created a prototype preemptible algorithm that is similar to *rcu*,[4] but adds per-CPU counts of preempted tasks, which operate in a manner in some ways similar to the generation mechanism in K42 [Gamsa99]. The key concept is that a preemptible kernel must track tasks rather than CPUs. However, to avoid potentially expensive scans of the task list or the runqueues, the tasks are tracked on a per-CPU basis. When a task returns from a voluntary context switch (or is created), it is implicitly associated with the CPU that it starts running on. No matter how many times the task is preempted, from a read-copy-update perspective, it remains affiliated with that CPU, even if it is migrated to other CPUs. Once it performs a voluntary context switch, it gives up its affiliation.

[4]However, as noted earlier, this preemptible version of *rcu* has greatly reduced CPU overhead when there are no read-copy callbacks in the system.

However, no additional work is done (over that done by a non-preemptible kernel running a non-preemptible implementation of read-copy update) until that task is preempted. The task then increments a per-CPU counter, which remains incremented until the task executes a voluntary context switch, possibly by exiting. The task then decrements that same per-CPU counter, even if the task is running on some other CPU at the time.

Of course, if there is a lot of preemption, it might be that a particular CPU *always* has at least one preempted task affiliated with it. However, the end of a grace period is marked not by the absence of tasks, but by each of the tasks that was either running or preempted at the start of the grace period having either exited or voluntarily switched context.

This distinction is maintained by providing each CPU with a pair of counters, a "next" counter that is incremented by tasks returning from their voluntary context switch onto the corresponding CPU, and a "current" counter that is only decremented. Note that the "next" counter will be also decremented whenever a task resumes execution quickly enough after being preempted. The end of the grace

```
1 extern atomic_t
2    rcu_preempt_cntr[2] __per_cpu_data;
3 extern atomic_t
4    *curr_preempt_cntr __per_cpu_data;
5 extern atomic_t
6    *next_preempt_cntr __per_cpu_data;
```

Figure 33: rcu_preempt Per-CPU Counters

period occurs when all CPUs' "current" counters reach zero.[5] The roles of the counters in each pair are now reversed in order to start the next grace period, just after the base *rcu* portion of the algorithm moves the callbacks in the `rcu_nextlist` to `rcu_currlist`.

Each CPU's pair of counters is as shown in Figure 33, along with the pair of pointers that handle the reversing of their roles. The `next_preempt_cntr` pointer points to the element of `rcu_preempt_cntr[]` that is atomically incremented (by a new `rcu_preempt_get()` function) when task affiliated with this CPU is preempted for the first time since its preceding voluntary context switch. The task records this pointer in a new `cpu_preempt_cntr` pointer in its task structure, which is initially NULL. After the task resumes and voluntarily relinquishes the CPU[6], it atomically decrements the counter pointed to by its `cpu_preempt_cntr`, using a new `rcu_preempt_put()` function, then NULLs its `cpu_preempt_cntr` pointer.

The `curr_preempt_cntr` pointer points to the element of `rcu_preempt_cntr[]` that `next_preempt_cntr` does not point to. This element of the array contains the number of tasks affiliated with this CPU that were first preempted before the beginning of the current grace period, and that must resume and voluntarily relinquish a CPU before the current grace period can expire. When this CPU becomes aware of the end of the current grace period, it exchanges the values of `next_preempt_cntr` and `curr_preempt_cntr`, so that the elements of the `rcu_preempt_cntr[]` array exchange roles.

[5] Unless one of the CPUs has been running a task continuously since before the start of the grace period, but this case is handled by the base *rcu* portion of the implementation.

[6] Possibly after having been preempted several more times along the way. This is why the counter cannot be decremented immediately when the task is resumed, but must instead wait for the task to voluntarily relinquish the CPU.

The rest of the callback processing is very similar to that of the *rcu* algorithm. The major difference is that `rcu_check_quiescent_state()` must check that all tasks preempted on this CPU prior to the current grace period have voluntarily relinquished the CPU.

6 Conclusions and Future Plans

Andrea Arcangeli's *rcu-poll* implementation exhibits the best `call_rcu()` latency, and is therefore a good implementation for workloads that do not have high aggregate `call_rcu()` invocation rates. The longer (but well-bounded) `call_rcu()` latencies of the *X-rcu*, *rcu-ltimer*, and *rcu* implementations may make them preferable for systems with higher `call_rcu()` invocation rates.

We are continuing our work on preemptible read-copy-update implementations, in order to obtain the best implementation compatible with the 2.5 kernel. Finally, we are continuing our measurements with various workloads, which we expect will evolve as the 2.5 kernel evolves. In particular, we will measure performance under heavy *call_rcu()* load.

7 Acknowledgments

We owe thanks to Martin Bligh and Hanna Linder for their able assistance with the machines we used to gather the data shown in this paper, and to Hans Tannenberger and Gerrit Huizenga for arranging access to these machines. We are especially grateful to Maneesh Soni and Hanna Linder for their efforts with read-copy update, and to Jonathan Appavoo for many enlightening discussions. We are indebted to Dan Frye, Randy Kalmeta, Hugh Blemings, and Manish Gupta for their support of this effort.

References

[Blanchard02a] A. Blanchard *some RCU dcache and ratcache results*, Linux-Kernel Mailing List, March 2002. http://marc.theaimsgroup.com/?l=linux-kernel&m=101637107412972&w=2.

[Compaq01] Compaq Computer Corporation *Shared Memory, Threads, Interprocess Communication*, Ask The Wizard, August 2001. http://www.openvms.compaq.com/wizard/wiz_2637.html.

[Gamsa99] B. Gamsa, O. Kreiger, J. Appavoo, and M. Stumm. *Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system*, Proceedings of the 3rd Symposium on Operating System Design and Implementation, New Orleans, LA, February, 1999.

[Linder02a] H. Linder, D. Sarma, and Maneesh Soni. *Scalability of the Directory Entry Cache*, To appear in Ottawa Linux Symposium, June 2002.

[Kleen02a] A. Kleen *Reduce Module Races*, kernel.org, January 2002. ftp://ftp.us.kernel.org/pub/linux/kernel/people/andrea/kernels/v2.4/v2.4.19pre7aa2/00_reduce-module-races-1.

[LSE] D. Sarma et al. *Linux Scaling Effort (LSE)*, SourceForge Project, April 2002. http://prdownloads.sourceforge.net/lse/.

[McK98a] P. E. McKenney and J. D. Slingwine. *Read-copy update: using execution history to solve concurrency problems*, Parallel and Distributed Computing and Systems, October 1998. (revised version available at http://www.rdrop.com/users/paulmck/rclockpdcsproof.pdf).

[McK01a] P. E. McKenney and D. Sarma. *Read-Copy Mutual Exclusion in Linux*, http://lse.sourceforge.net/locking/rcu/rcupdate_doc.html, February 2001.

[McK01b] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, M. Soni. *Read-Copy Update*, Ottawa Linux Symposium, July 2001. (revised version available at http://www.rdrop.com/users/paulmck/rclock/rclock_OLS.2001.05.01c.pdf).

[McK01c] P. E. McKenney, et al. *RFC: patch to allow lock-free traversal of lists with insertion*, LKML, October 2001. http://www.ussg.iu.edu/hypermail/linux/kernel/0110.1/0239.html.

[McK01d] P. E. McKenney, et al. *Data Dependencies and wmb()*, LSE, October 2001. http://lse.sourceforge.net/locking/wmbdd.html.

[Russell01b] R. Russell. *Re: [PATCH for 2.5] preemptible kernel*, http://www.uwsg.indiana.edu/hypermail/linux/kernel/0103.3/1070.html, March 2001.

[Russell01d] R. Russell *Re: 2.4.10pre7aa1*, Linux-Kernel Mailing List, September 2001. http://www.ussg.iu.edu/hypermail/linux/kernel/0109.2/0021.html.

[Russell02a] R. Russell *Re: [PATCH] per-cpu areas for 2.5.3-pre6*, Linux-Kernel Mailing List, February 2002. http://marc.theaimsgroup.com/?l=linux-kernel&m=101255391528359&w=2.

[Sarma02a] D. Sarma *[RFC][PATCH] Ingo's smptimers patch experiment*, Linux-Kernel Mailing List, February 2002. http://marc.theaimsgroup.com/?l=linux-kernel&m=101301053225522&w=2.

[Sarma02b] D. Sarma *[PATCH] memory barriers*, Linux-Kernel Mailing List, March 2002. http://www.ussg.iu.edu/hypermail/linux/kernel/0203.2/1604.html.

8 Trademarks

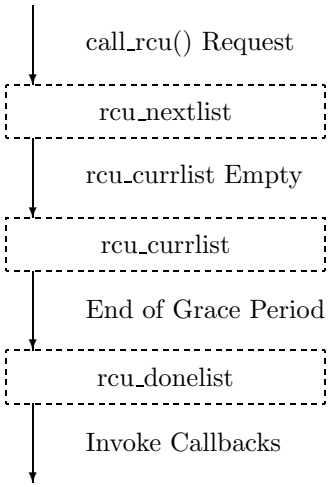

Figure 34: RCU Callback Flow

Appendix

A Implementation Details

These appendices contain more implementation details of the various algorithms.

A.1 *X-rcu* Callback Processing

This section describes the *X-rcu* callback processing. The processing proceeds as shown in Figure 34.

The `rcu_process_callbacks()` function shown in Figure 35 handles the overall flow. Lines 3-12 move callbacks from `rcu_currlist` to `rcu_donelist` after the end of a grace period. Line 14 invokes `rcu_move_next_batch()` (shown in Figure 36), which moves callbacks from `rcu_nextlist` to `rcu_currlist`, initiating grace-period detection if needed. Line 16 calls `rcu_check_quiescent_state()`, which checks to see if the current CPU has passed through a quiescent state since the beginning of the current grace period. Lines 18-22 call `rcu_invoke_callbacks()` to invoke any callbacks in `rcu_donelist`.

The `rcu_move_next_batch()` function shown in Figure 36 disables local interrupts (line 3), and then checks to see if `rcu_currlist` is empty and `rcu_nextlist` is not (lines 4-7). If so, it moves the

```
 1 static void rcu_process_callbacks(void)
 2 {
 3     if (!list_empty(
 4            &this_cpu(rcu_currlist)) &&
 5         RCU_BATCH_GT(rcu_currbatch,
 6                      this_cpu(rcu_batch))) {
 7         list_splice(
 8             &this_cpu(rcu_currlist),
 9             &this_cpu(rcu_donelist));
10         INIT_LIST_HEAD(
11             &this_cpu(rcu_currlist));
12     }
13
14     rcu_move_next_batch();
15
16     rcu_check_quiescent_state();
17
18     if (!list_empty(
19             &this_cpu(rcu_donelist))) {
20         rcu_invoke_callbacks(
21             &this_cpu(rcu_donelist));
22     }
23 }
```

Figure 35: *X-rcu* rcu_process_callbacks()

contents of `rcu_nextlist` to `rcu_currlist` (lines 8 and 9), then re-enables interrupts (line 12). It then obtains a new RCU batch number (lines 18-19) and registers it using `rcu_reg_batch()` (line 20, see Figure 39 for this function's definition) under the `rcu_lock`.

If lines 4-5 find `rcu_currlist` to be nonempty, `rcu_move_next_batch()` simply re-enables interrupts and returns (line 23).

The `rcu_check_quiescent_state()` function shown in Figure 37 checks to see if the current CPU has gone through a quiescent state, and, if so, publicizes it.

Lines 6-8 check to see if this CPU has already passed through a quiescent state during the current grace period, and, if so, line 6 simply returns. Lines 17-22 check to see if this is the first that this CPU has heard of the current grace period, and, if so, lines 19-20 take a snapshot of this CPU's context-switch counter in `rcu_last_qsctr` and returns. Lines 23-26 check to see if this CPU has passed through a quiescent state since the snapshot, and, if not, line 25 simply returns.

Execution reaches line 29 when this CPU first determines that it has passed through a quiescent state

```
 1 static void rcu_move_next_batch(void)
 2 {
 3     local_irq_disable();
 4     if (!list_empty(
 5              &this_cpu(rcu_nextlist)) &&
 6         list_empty(
 7              &this_cpu(rcu_currlist))) {
 8         list_splice(&this_cpu(rcu_nextlist),
 9                     &this_cpu(rcu_currlist));
10         INIT_LIST_HEAD(
11                     &this_cpu(rcu_nextlist));
12         local_irq_enable();
13
14         /*
15          * start the next batch of callbacks
16          */
17         spin_lock(&rcu_lock);
18         this_cpu(rcu_batch) =
19                       rcu_currbatch + 1;
20         rcu_reg_batch(this_cpu(rcu_batch));
21         spin_unlock(&rcu_lock);
22     } else {
23         local_irq_enable();
24     }
25 }
```

Figure 36: *X-rcu* rcu_move_next_batch()

in the current grace period. Lines 28-44 publish this fact under the global `rcu_lock`, which possibly marks the end of the current grace period. Line 33 clears this CPU's bit in `rcu_cpumask`, which publicizes the fact that this CPU has passed through a quiescent state during the current grace period. Lines 34-35 set `rcu_last_qsctr` to an invalid quantity, which will indicate that this CPU is not yet aware of the next grace period. If there are other CPUs that have not yet passed through their quiescent states, then lines 36-41 release the `rcu_lock` and return. Execution reaches line 42 if this CPU is the last one to detect that it has passed through a quiescent state during the current grace period, which marks the end of the grace period. Line 42 increments `rcu_currbatch`, which signals the end of the grace period. Line 43 invokes `rcu_reg_batch()` to initiate a new grace period if needed, and line 36 releases the `rcu_lock`.

Figure 38 shows `rcu_invoke_callbacks()`, which simply loops through the list of callbacks, invoking each in turn.

Figure 39 shows `rcu_reg_batch()`, which publicizes the beginning of a new grace period, if needed. Lines 4-7 check to see if the batch number of the

```
 1 static void rcu_check_quiescent_state(void)
 2 {
 3     int cpu = cpu_number_map(
 4                  smp_processor_id());
 5
 6     if (!test_bit(cpu, &rcu_cpumask)) {
 7         return;
 8     }
 9
10     /*
11      * May race with rcu per-cpu tick -
12      * in the worst case
13      * we may miss one quiescent state
14      * of that CPU. That is tolerable.
15      * So no need to disable interrupts.
16      */
17     if (this_cpu(rcu_last_qsctr) ==
18                      RCU_QSCTR_INVALID) {
19         this_cpu(rcu_last_qsctr) =
20                      this_cpu(rcu_qsctr);
21         return;
22     }
23     if (this_cpu(rcu_qsctr) ==
24              this_cpu(rcu_last_qsctr)) {
25         return;
26     }
27
28     spin_lock(&rcu_lock);
29     if (!test_bit(cpu, &rcu_cpumask)) {
30         spin_unlock(&rcu_lock);
31         return;
32     }
33     clear_bit(cpu, &rcu_cpumask);
34     this_cpu(rcu_last_qsctr) =
35                      RCU_QSCTR_INVALID;
36     if (rcu_cpumask != 0) {
37         /* All CPUs haven't gone
38            through a quiescent state */
39         spin_unlock(&rcu_lock);
40         return;
41     }
42     rcu_currbatch++;
43     rcu_reg_batch(rcu_maxbatch);
44     spin_unlock(&rcu_lock);
45 }
```

Figure 37: *X-rcu* rcu_check_quiescent_state()

```
 1 static inline void rcu_invoke_callbacks(
 2                   struct list_head *list)
 3 {
 4     struct list_head *entry;
 5     struct rcu_head *head;
 6
 7     while (!list_empty(list)) {
 8         entry = list->next;
 9         list_del(entry);
10         head = list_entry(entry,
11                   struct rcu_head, list);
12         head->func(head->arg);
13     }
14 }
```

Figure 38: *X-rcu* rcu_invoke_callbacks()

```
 1 static inline void rcu_reg_batch(
 2                 rcu_batch_t newbatch)
 3 {
 4     if (RCU_BATCH_LT(rcu_maxbatch,
 5                          newbatch)) {
 6         rcu_maxbatch = newbatch;
 7     }
 8     if (RCU_BATCH_LT(rcu_maxbatch,
 9             rcu_currbatch) ||
10         (rcu_cpumask != 0)) {
11         return;
12     }
13     rcu_cpumask = cpu_online_map;
14 }
```

Figure 39: *X-rcu* rcu_reg_batch()

requested grace period is larger than that of the largest-numbered grace period that has been requested thus far (the `RCU_BATCH_LT()` macro handles wraparound). If so, line 6 publicizes the new maximum batch number. If the largest-numbered grace period requested thus far has already completed or if a grace period is currently in progress, lines 8-12 simply return. Otherwise, line 13 sets `rcu_cpumask` to indicate that all CPUs need to pass through a quiescent state, which publicizes the start of a new grace period.

A.2 *rcu* Callback Processing

The *rcu* algorithm's callback processing is very similar to that of the *X-rcu* algorithm, shown in Appendix A.1. Differences include:

```
 1 static int rcu_prepare_polling(void)
 2 {
 3     int stop;
 4     int i;
 5
 6 #ifdef DEBUG
 7     if (!list_empty(&rcu_curlist))
 8         BUG();
 9 #endif
10
11     stop = 1;
12     if (!list_empty(&rcu_nxtlist)) {
13         list_splice(&rcu_nxtlist, &rcu_curlist);
14         INIT_LIST_HEAD(&rcu_nxtlist);
15
16         rcu_polling_in_progress = 1;
17
18         for (i = 0; i < smp_num_cpus; i++) {
19             int cpu = cpu_logical_map(i);
20
21             rcu_qsmask |= 1UL << cpu;
22             rcu_quiescent_checkpoint[cpu] =
23                         RCU_quiescent(cpu);
24             force_cpu_reschedule(cpu);
25         }
26         stop = 0;
27     }
28
29     return stop;
30 }
```

Figure 40: *rcu-poll* rcu_prepare_polling()

1. *rcu* must explicitly index into arrays containing per-CPU elements, while *X-rcu* directly accesses the per-CPU data area.

2. *rcu*'s `rcu_process_callbacks()` contains code that clears the current CPU's bit from `rcu_active_cpumask`.

3. *rcu*'s `rcu_move_next_batch()` contains code that sets the current CPU's bit in `rcu_active_cpumask` and schedules the timer if there are RCU callbacks active and the timer is not already scheduled.

A.3 *rcu-poll* Callback Processing

Figure 40 shows the `rcu_prepare_polling()` function. This function relies on `rcu_process_callbacks()` (see Figure 20) acquiring the `rcu_lock`. Lines 12-27 check to see if there are callbacks waiting in `rcu_nxtlist`, and, if

```
 1 static int rcu_polling(void)
 2 {
 3     int i;
 4     int stop;
 5
 6     for (i = 0; i < smp_num_cpus; i++) {
 7         int cpu = cpu_logical_map(i);
 8
 9         if (rcu_qsmask & (1UL << cpu))
10             if (rcu_quiescent_checkpoint[cpu]
11                   != RCU_quiescent(cpu))
12                 rcu_qsmask &= ~(1UL << cpu);
13     }
14
15     stop = 0;
16     if (!rcu_qsmask)
17         stop = rcu_completion();
18
19     return stop;
20 }
```

Figure 41: *rcu-poll* rcu_polling()

so, starts a grace period. Lines 13-14 move the list from `rcu_nxtlist` to `rcu_curlist`. Line 16 records the fact that a grace period is now in progress. Lines 18-25 mark each CPU (other than the current one) as needing to go through a quiescent state, take a snapshot of each CPU's context-switch counter, and expedite a context switch. Line 26 indicates that grace-period polling needs to continue – if `rcu_nxtlist` had been empty, polling would cease until the next `call_rcu()` invocation.

Figure 41 shows the `rcu_polling()` function. Lines 6-13 check each CPU that has not yet been observed passing through a quiescent state (as indicated by the `rcu_qsmask` check at line 9) to see if that CPU's `RCU_quiescent` counter has advanced since the `rcu_prepare_polling()` started the current grace period. If it has, then that CPU has recently passed through a quiescent state, so line 12 clears its bit from `rcu_qsmask`. Line 16 then checks to see if all CPUs have now passed through their quiescent states. If so, line 17 invokes `rcu_completion()` to mark the end of the grace period. If another grace period is required, `rcu_completion` will have started it, and will then return zero to signal that grace-period polling should continue.

Figure 42 shows the `rcu_completion()` function that is invoked at the end of a grace period. Line 5 records the fact that a grace period is no longer in progress, line 6 invokes `rcu_invoke_callbacks()`

```
 1 static int rcu_completion(void)
 2 {
 3     int stop;
 4
 5     rcu_polling_in_progress = 0;
 6     rcu_invoke_callbacks();
 7
 8     stop = rcu_prepare_polling();
 9
10     return stop;
11 }
```

Figure 42: *rcu-poll* rcu_completion()

```
 1 static void rcu_invoke_callbacks(void)
 2 {
 3     struct list_head *entry;
 4     struct rcu_head *head;
 5
 6 #ifdef DEBUG
 7     if (list_empty(&rcu_curlist))
 8         BUG();
 9 #endif
10
11     entry = rcu_curlist.prev;
12     do {
13         head = list_entry(entry,
14                  struct rcu_head, list);
15         entry = entry->prev;
16
17         head->func(head->arg);
18     } while (entry != &rcu_curlist);
19
20     INIT_LIST_HEAD(&rcu_curlist);
21 }
```

Figure 43: *rcu-poll* rcu_invoke_callbacks()

to invoke the callbacks, and line 8 starts a new grace period, if required.

Figure 43 shows the `rcu_invoke_callbacks()` function. This is similar to that shown for *X-rcu* in Figure 38, but processes a single global list rather than a per-CPU list, and removes elements from the list in a slightly different manner.

A.4 *rcu-ltimer* Callback Processing

This implementation is closest to that in ptx, and is thus driven from timer handlers, as noted in Section 5.4. The `rcu_process_callbacks()` function, shown in Figure 44 is invoked on every timer tick

```
 1 static void rcu_process_callbacks(
 2                 unsigned long data)
 3 {
 4     int cpu = cpu_number_map(
 5                   smp_processor_id());
 6
 7     if ((!list_empty(&RCU_curlist(cpu)) &&
 8          RCU_BATCH_LT(RCU_batch(cpu),
 9                 rcu_ctrlblk.curbatch)) ||
10         (list_empty(&RCU_curlist(cpu)) &&
11             !list_empty(&RCU_nxtlist(cpu))) ||
12         test_bit(cpu,
13                 &rcu_ctrlblk.rcu_cpu_mask))
14             rcu_check_callbacks();
15 }
```

Figure 44: *rcu-ltimer* rcu_process_callbacks()

to process the per-CPU callback lists. This function invokes `rcu_check_callbacks()` if any of the following are true:

1. There are callbacks in `RCU_curlist` and the corresponding grace period has expired (lines 7-9).
2. There are no callbacks in `RCU_curlist`, but there are some in `RCU_nxtlist` waiting to start a grace period (lines 10-11).
3. This CPU has not yet passed through a quiescent state for the current grace period (line 12-13).

Figure 45 shows `rcu_check_callbacks()` advances callbacks for the current CPU through the lists. Lines 7-13 check to see if the grace period corresponding to callbacks in this CPU's `RCU_curlist` has expired, and, if so, moves the contents of this list to the local variable `list`. Lines 15-29 check to see if this CPU's `RCU_curlist` is empty and if there are callbacks in this CPU's `RCU_nxtlist` waiting to start a grace period, and, if so, moves them from `RCU_nxtlist` to `RCU_curlist` on lines 17-19 and requests a new grace period in lines 24-28. Line 30 then checks to see if this CPU has passed through a quiescent state. Lines 31-32 invoke any callbacks on local variable `list`.

Figure 46 shows `rcu_reg_batch()`, which schedules a new grace period if required. Lines 4-7 check to see if the new batch number is larger than the largest seen thus far, and, if so, records the new maximum batch number on line 6. Lines 8-10 check to see if

```
 1 static void rcu_check_callbacks(void)
 2 {
 3     int cpu = cpu_number_map(
 4                   smp_processor_id());
 5     LIST_HEAD(list);
 6
 7     if (!list_empty(&RCU_curlist(cpu)) &&
 8         RCU_BATCH_GT(rcu_ctrlblk.curbatch,
 9                          RCU_batch(cpu))) {
10         list_splice(&RCU_curlist(cpu),
11                     &list);
12         INIT_LIST_HEAD(&RCU_curlist(cpu));
13     }
14
15     if (!list_empty(&RCU_nxtlist(cpu)) &&
16         list_empty(&RCU_curlist(cpu))) {
17         list_splice(&RCU_nxtlist(cpu),
18                     &RCU_curlist(cpu));
19         INIT_LIST_HEAD(&RCU_nxtlist(cpu));
20
21         /*
22          * start the next batch of callbacks
23          */
24         spin_lock(&rcu_ctrlblk.mutex);
25         RCU_batch(cpu) =
26             rcu_ctrlblk.curbatch + 1;
27         rcu_reg_batch(RCU_batch(cpu));
28         spin_unlock(&rcu_ctrlblk.mutex);
29     }
30     rcu_check_quiescent_state();
31     if (!list_empty(&list))
32         rcu_invoke_callbacks(&list);
33 }
```

Figure 45: *rcu-ltimer* rcu_check_callbacks()

```
 1 static void rcu_reg_batch(
 2              rcu_batch_t newbatch)
 3 {
 4     if (RCU_BATCH_LT(rcu_ctrlblk.maxbatch,
 5                                newbatch)) {
 6         rcu_ctrlblk.maxbatch = newbatch;
 7     }
 8     if (RCU_BATCH_LT(rcu_ctrlblk.maxbatch,
 9                      rcu_ctrlblk.curbatch) ||
10         (rcu_ctrlblk.rcu_cpu_mask != 0)) {
11         return;
12     }
13     rcu_ctrlblk.rcu_cpu_mask =
14                          cpu_online_map;
15 }
```

Figure 46: *rcu-ltimer* rcu_reg_batch()

the grace period corresponding to the largest batch number has already expired (lines 8-9), or if a grace period is already in progress (line 10), and, in either case, simply returns. Otherwise, lines 13-14 record the fact that all CPUs need to go through a quiescent state for the new grace period. As before, the `RCU_BATCH_LT()` macros check for batch-number wraparound.

Figure 47 shows how `rcu_check_quiescent_state()` checks that the current CPU has passed through a quiescent state since the beginning of the current grace period. Lines 6-9 check to see if this CPU has already passed through a quiescent state, and, if so, simply returns. Lines 19-20 checks to see if this CPU is unaware of the current grace period, and, if so, snapshots the current quiescent-state counter on lines 21-22 and then returns. Lines 25-28 checks to see if this CPU has passed through a quiescent state since it became aware of the current grace period, and, if not, simply returns. Execution reaches line 30 the first time that this CPU realizes that it has passed through a quiescent state since it became aware of the current grace period. Lines 36 and 37 publish the fact that this CPU has passed through a quiescent state. Lines 38-41 check to see if this is the last CPU to pass through a quiescent state, thus ending the grace period, and returning if not. Line 42 publicizes the end of the grace period, and line 43 invokes `rcu_reg_batch()` to start a new grace period, if one is needed.

```
 1 static void rcu_check_quiescent_state(void)
 2 {
 3     int cpu = cpu_number_map(
 4                     smp_processor_id());
 5
 6     if (!test_bit(cpu,
 7             &rcu_ctrlblk.rcu_cpu_mask)) {
 8         return;
 9     }
10
11     /*
12      * Races with local timer interrupt -
13      * in the worst case
14      * we may miss one quiescent state
15      * of that CPU. That is
16      * tolerable. So no need
17      * to disable interrupts.
18      */
19     if (RCU_last_qsctr(cpu) ==
20                 RCU_QSCTR_INVALID) {
21         RCU_last_qsctr(cpu) =
22                     RCU_qsctr(cpu);
23         return;
24     }
25     if (RCU_qsctr(cpu) ==
26              RCU_last_qsctr(cpu)) {
27         return;
28     }
29
30     spin_lock(&rcu_ctrlblk.mutex);
31     if (!test_bit(cpu,
32         &rcu_ctrlblk.rcu_cpu_mask)) {
33         spin_unlock(&rcu_ctrlblk.mutex);
34         return;
35     }
36     clear_bit(cpu, &rcu_ctrlblk.rcu_cpu_mask);
37     RCU_last_qsctr(cpu) = RCU_QSCTR_INVALID;
38     if (rcu_ctrlblk.rcu_cpu_mask != 0) {
39         spin_unlock(&rcu_ctrlblk.mutex);
40         return;
41     }
42     rcu_ctrlblk.curbatch++;
43     rcu_reg_batch(rcu_ctrlblk.maxbatch);
44     spin_unlock(&rcu_ctrlblk.mutex);
45 }
```

Figure 47: *rcu-ltimer* rcu_check_quiescent_state()

```
 1 struct el *insert(long key, long data)
 2 {
 3     struct el *p;
 4     p = kmalloc(sizeof(*p), GPF_ATOMIC);
 5     spin_lock(&mutex);
 6     p->next = head.next;
 7     p->key = key;
 8     p->data = data;
 9     wmb();
10     head.next = p;
11     spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
15 {
16     struct el *p;
17     p = head.next;
18     while (p != &head) {
19         /* BUG ON ALPHA!!! */
20         if (p->key == key) {
21             return (p);
22         }
23         p = p->next;
24     };
25     return (NULL);
26 }
```

Figure 48: Insert and Lock-Free Search

B Memory Ordering Issues

This paper has heretofore focused on lock-free search on lists subject to concurrent deletion. Insertion poses additional problems on systems with very weak memory ordering, as noted in recent discussions on LKML [McK01c]. This appendix focuses on these problems and some solutions.

Some of these problems may be addressed by using the `wmb()` primitive as shown on line 9 of Figure 48. This `wmb()` guarantees that the element initialization in lines 6-8 is not executed before the element is added to the list on line 10. On many (*but not all*) CPUs, this is sufficient, and the lock-free search on lines 14-26 will then operate correctly.

However, some CPUs, such as Alpha, have extremely weak memory ordering such that the code on line 20 of Figure 48 could see the old garbage values that were present before the initialization on lines 6-8.

Figure 49 shows how this can happen on an aggressively parallel machine with partitioned caches, so

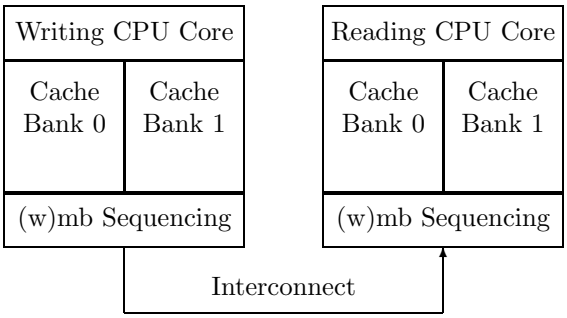

Figure 49: Why rmb() is Required

that alternating caches lines are processed by the different partitions of the caches. Assume that the list header `head` will be processed by cache bank 0 and that the new element will be processed by cache bank 1. On Alpha, the `wmb()` will guarantee that the cache invalidates performed by lines 6-8 of Figure 48 will reach the interconnect before that of line 10 does, but makes absolutely no guarantee about the order in which the new values will reach the reading CPU's core. For example, it is possible that the reading CPU's cache bank 1 is very busy, but cache bank 0 is idle. This could result in the cache invalidates for the new element being delayed, so that the reading CPU gets the new value for the pointer, but sees the old cached values for the new element. See Compaq's Alpha documentation [Compaq01] for more information, or if you think we are just making all this up.

This can be fixed in an implementation-independent manner by inserting an `rmb()` between the pointer fetch and dereference, as shown on line 19 of Figure 50. However, this imposes unneeded overhead on systems (such as i386, IA64, PPC, and SPARC) that respect data dependencies on the read side. A `read_barrier_depends()` primitive has been proposed to eliminate overhead no these systems [Sarma02b]. It is also possible to implement a software barrier that could be used in place of `wmb()`, which would force all reading CPUs to see the writing CPU's writes in order[McK01d]. However, this approach is deemed to impose excessive overhead on extremely weakly ordered CPUs such as Alpha.[7]

For the moment, `rmb()` should be used on lock-free code paths traversing lists subject to concurrent insertion.

[7] CPUs that respect data dependencies would define such a barrier to simply be `wmb()`.

```
 1 struct el *insert(long key, long data)
 2 {
 3     struct el *p;
 4     p = kmalloc(sizeof(*p), GPF_ATOMIC);
 5     spin_lock(&mutex);
 6     p->next = head.next;
 7     p->key = key;
 8     p->data = data;
 9     wmb();
10     head.next = p;
11     spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
15 {
16     struct el *p;
17     p = head.next;
18     while (p != &head) {
19         rmb();
20         if (p->key == key) {
21             return (p);
22         }
23         p = p->next;
24     };
25     return (NULL);
26 }
```

Figure 50: Safe Insert and Lock-Free Search