Introducing Technology Into the Linux Kernel: A Case Study*

Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@linux.vnet.ibm.com

Jonathan Walpole
Computer Science Department
Portland State University
walpole@cs.pdx.edu

ABSTRACT

There can be no doubt that a great many technologies have been added to Linux™ over the past ten years. What is less well-known is that it is often necessary to introduce a large amount of Linux into a given technology in order to successfully introduce that technology into Linux. This paper illustrates such an introduction of Linux into technology with Read-Copy Update (RCU). The RCU API's evolution over time clearly shows that Linux's extremely diverse set of workloads and platforms has changed RCU to a far greater degree than RCU has changed Linux—and it is reasonable to expect that other technologies that might be proposed for inclusion into Linux would face similar challenges. In addition, this paper presents a summary of lessons learned and an attempt to foresee what additional challenges Linux might present to RCU.

1. INTRODUCTION

Linux is an operating-system kernel that is used in a variety of platforms ranging from cellphones to super-computers, with more than an 80% share of the Top 500 Supercomputer Sites as of November 2007 [44], up from about 10% in late 2000. Although a very large amount of functionality has been added to the Linux kernel between 2000 and 2007, space constraints limit this paper to discussing but one specific niche technology, namely RCU. We shall see that the extreme diversity of Linux's platforms and workloads posed special challenges to RCU. It seems likely that this diversity would pose similar challenges to other technologies that might be proposed for inclusion into the Linux kernel.

Section 2 gives a brief overview of RCU, Section 3 gives a quantitative summary of RCU's use by the Linux kernel since its acceptance in late 2002, and Section 4 gives an overview of RCU and its environment in production systems prior to its acceptance into Linux. Section 5 reviews the evolution of the RCU API in the Linux kernel, and Section 6 delineates some of the forces underlying this evolution. Section 7 presents some speculation on the future evolution of the RCU API, and Section 8 presents concluding remarks. Finally, Section 9 contains RCU history subsequent to this paper's initial publication in SIGOPS Operating Systems Review [43].

2. OVERVIEW OF RCU

This paper does not discuss RCU itself in extreme depth, nor does it require that the reader possess any special knowledge of RCU (or, for that matter, of Linux). That said, this section provides a brief conceptual overview of RCU. People who are already familiar with RCU may wish to skip this section.

RCU is a specialized synchronization primitive that can be thought of as a replacement for reader-writer locking, but with extreme read-side performance, scalability, and determinism. These read-side benefits are achieved by allowing reads to run concurrently with a single update, in contrast with conventional locking primitives, which enforce strict mutual exclusion between readers on the one hand and updaters on the other. RCU provides readers a coherent view of shared data by maintaining multiple versions of objects and by ensuring that old versions are retained until all pre-existing RCU read-side critical sections complete. RCU implementations provide efficient and scalable mechanisms for publishing and reading new objects and for appropriately deferring collection of old objects. These implementations optimize the read paths at the expense of the update paths, in fact, in the limiting case of non-preemptible kernels, RCU's read-side primitives generate absolutely no code, and thus enjoy zero overhead.

This leads to the question "what exactly is RCU?", and, not infrequently, "how could RCU *possibly* work??", to say nothing of the assertion that RCU cannot possibly work. RCU is made up of three fundamental mechanisms, the first being a publish-subscribe mechanism used for insertion (Section 2.1), the second being a mechanism that defers storage reclamation until all RCU readers in progress during deletion have completed (Section 2.2), and the third being a multi-version mechanism that permits readers to tolerate concurrent insertion and deletion (Section 2.3). Section 2.4 presents a "toy" RCU implementation, and finally, Section 2.5 lists citations to which interested readers may refer in order to learn more about RCU.

2.1 Publish-Subscribe Mechanism

RCU updaters publish an object by first initializing the object, then storing a pointer to the object into a memory location accessible to all CPUs. RCU readers subscribe to an object by loading its pointer. This mechanism is simple, and would be completely trivial were it not for the fact that both CPUs and compilers freely re-order operations. For example, both compilers and weakly ordered CPUs might execute lines 2-5 in any order:

*Updated November 5, 2009

```
1 spin_lock(&mylock);
2 p->a = 1;
3 p->b = 2;
4 p->c = 3;
5 gp = p;
6 spin_unlock(&mylock);
```

In particular, if this section of code suffered from register pressure, and if the address of `gp` was already in a register, the compiler might choose to generate the code for line 5 first, freeing up the register for use in lines 2-4. Such reordering could fatally confuse concurrent readers, who might then see the old garbage values for `p->a`, `p->b`, and `p->c`. The `rcu_assign_pointer()` primitive directs both the compiler and the CPU to suppress such fatal reordering, as follows:

```
1 spin_lock(&mylock);
2 p->a = 1;
3 p->b = 2;
4 p->c = 3;
5 rcu_assign_pointer(gp, p);
6 spin_unlock(&mylock);
```

Read-side misordering is highly counter-intuitive, but still possible given value-speculation compiler optimizations and the DEC Alpha [26], although a full explanation is beyond the scope of this paper. The `rcu_dereference()` primitive is used to suppress such reordering by both compiler and CPU, as follows:

```
1 rcu_read_lock();
2 p = rcu_dereference(gp);
3 if (p != NULL) {
4   do_something_with(p->a, p->b, p->c);
5 }
6 rcu_read_unlock();
```

Line 2 has the same effect that `p=gp` would, but also suppresses any reordering that might otherwise be undertaken by CPU or compiler. The `rcu_read_lock()` and `rcu_read_unlock()` primitives delimit the RCU read-side critical section.

The `rcu_assign_pointer()` primitive publishes a structure, and the `rcu_dereference()` primitive subscribes to a previously published structure, allowing new structures to be inserted despite the presence of concurrent readers. However, it is also necessary to handle removal and deletion, as discussed in the following section.

2.2 Wait For Pre-Existing RCU Readers to Complete

The following code is a first attempt to replace the structure referenced by `gp`, freeing the old version:

```
1 spin_lock(&mylock);
2 p->a = 1;
3 p->b = 2;
4 p->c = 3;
5 q = gp;
6 rcu_assign_pointer(gp, p);
7 spin_unlock(&mylock);
8 kfree(q);
```

Note that `rcu_dereference()` is not needed on line 5 because holding `mylock` excludes all other updaters. However, it is possible that other CPUs might reference `gp` immediately before the assignment on line 6 above. Such an unlucky reading CPU might still be referencing the structure after it is freed (and perhaps immediately reallocated) on line 8 above. This problem can be avoided by using the `synchronize_rcu()` primitive as shown below:

```
1 spin_lock(&mylock);
2 p->a = 1;
3 p->b = 2;
4 p->c = 3;
5 q = gp;
6 rcu_assign_pointer(gp, p);
7 spin_unlock(&mylock);
8 synchronize_rcu();
9 kfree(q);
```

The `synchronize_rcu()` primitive waits for the completion of all pre-existing RCU read-side critical sections. Thus, any reading CPU holding a reference to the old value of `gp` will have released it before `synchronize_rcu()` returns. This approach has the effect of maintaining multiple versions, as described in the following section.

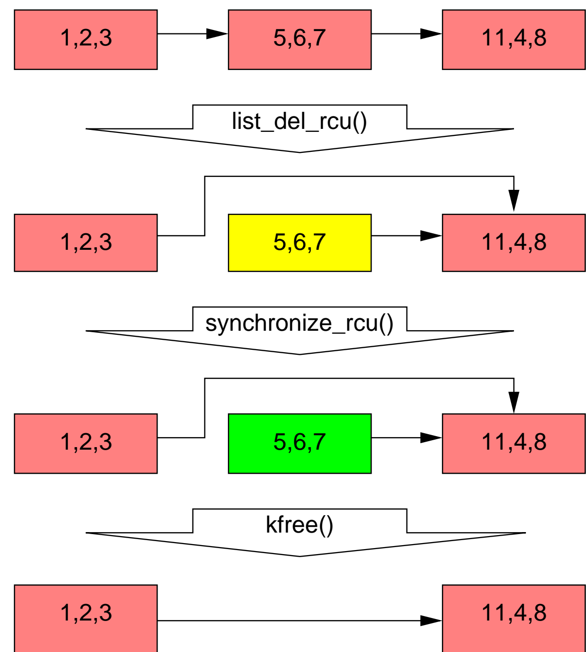

Figure 1: RCU Deletion From Linked List

2.3 Maintain Multiple Versions of Recently Updated Objects

Figure 1 shows the sequence of states of an RCU-protected linked list when the second element is being deleted and freed. The triples in each element represent the values of the fields `a`, `b`, and `c`, respectively. The red-shaded elements indicate that readers might be holding references to them.

The `list_del_rcu()` primitive unlinks the second element from the list, resulting in the state shown in the second row of Figure 1. The element is shaded yellow because although old RCU readers might still be referencing it, new RCU readers cannot obtain a reference. At this point, there are in effect two versions of the list: some RCU readers will see

{1,2,3 5,6,7 11,4,8}, while others will see {1,2,3 11,4,8}.

The `synchronize_rcu()` primitive blocks until all pre-existing RCU readers complete, after which point there can be no readers referencing the second element, as indicated by the green shading on the third row. At this point, all RCU readers see a single version of the list, namely, {1,2,3 11,4,8}. It is then safe to free that element, as shown on the last row.

This of course leads to the question of how one could possibly implement `synchronize_rcu()`, especially in cases where the read-side primitives generate no code. This question is taken up in the next section.

2.4 Toy RCU Implementation

Consider a non-preemptive kernel environment, where all threads run to block. In this case, it is illegal to block while holding a spinlock, as doing so can result in a deadlock situation where all the CPUs are spinning on a lock held by a blocked thread. The CPUs cannot acquire the lock until it is released, but the blocked thread cannot release until after at least one of the CPUs acquires the lock. This same restriction applies to RCU read-side critical sections, so that it is illegal to block while traversing an RCU-protected data structure.

This restriction is sufficient to admit the following trivial implementation of `synchronize_rcu()`:

```
1 void synchronize_rcu()
2 {
3   foreach_cpu(cpu)
4     run_on(cpu);
5 }
```

This code fragment simply runs on each CPU in turn. To see how this works, consider the situation once `synchronize_rcu()` has started running on CPU 0. Whatever was running on CPU 0 beforehand must have blocked, otherwise `synchronize_rcu()` could not have begun running on CPU 0. Because it is illegal to block within an RCU read-side critical section, all prior RCU read-side critical sections running on CPU 0 must have completed. This same line of reasoning applies to each of the other CPUs that `synchronize_rcu()` runs on, so that once `synchronize_rcu()` has completed, all prior RCU read-side critcial sections throughout the system must have completed.

Production-quality `synchronize_rcu()` implementations are more complex due to the need for performance and scalability, the need to preempt RCU read-side critical sections in real-time systems, and the need to tolerate CPUs being added to and removed from the system, for example, in order to conserve energy when the system is mostly idle.

2.5 Additional Information on RCU

Readers wishing more information on RCU are referred to a number of RCU-related publications covering fundamental concepts [42], usage [35], the Linux-kernel RCU API [34], implementation of the RCU infrastructure [1, 24, 28, 31, 32, 33], real-time adaptations of RCU [12, 16, 40, 38, 29, 57], and the performance of RCU [13, 25, 48, 50]. There are also a number of publications on other mechanisms that in some ways resemble RCU [10, 15, 18, 19, 20, 21, 22, 52, 53, 56, 58]. In addition, the Linux 2.4 kernel's use of the `brlock` per-CPU reader-writer locking primitive in the networking stack also has some resemblance to RCU. (The `brlock` primitive resembles Hsieh's and Weihl's scalable reader-writer locks [17].)

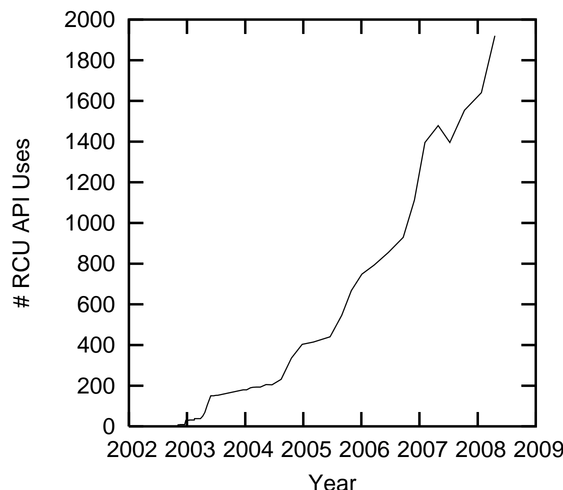

Figure 2: RCU API Usage in the Linux Kernel

3. RCU USAGE WITHIN LINUX

RCU's usage within the Linux kernel has increased rapidly over the past five years, as shown in Figure 2 [27]. In some cases, RCU has displaced other synchronization mechanisms in existing code (for example, `brlock` in the networking protocol stacks [47, 62, 63]), while in other cases it has been introduced with code implementing new functionality (for example, the audit system within SELinux [48]). Despite its rapid growth, RCU remains a niche technology, as shown by the comparison with locking in Figure 3. Nonetheless, RCU can be characterized as a reasonably successful niche technology within the Linux kernel. As such, it is useful to review the path RCU took in achieving this modest level of success, which was due more to RCU's being dramatically changed by Linux than by Linux being changed by RCU.

4. RCU BEFORE LINUX

Before Linux, production use of RCU-like mechanisms appears to have been confined to large data-processing systems such as the IBM mainframe's VM/XA [15] and Sequent's (now IBM's) Symmetry and NUMA-Q systems running the DYNIX/ptx operating system [41]. These were large (for the time) enterprise systems running parallel data-processing workloads. These systems normally ran in a protected networking environment, behind firewalls or client machines with restricted usage modes. The real-time response required of these machines is perhaps best exemplified by the TPC/A benchmark [64], which has the very soft real-time requirement that 90% of transactions complete in two seconds or less.

Back when the author was still foolish enough to believe that he knew all that there was to know about RCU, the RCU API for DYNIX/ptx [37] consisted of only the following members (translated to their Linux equivalents, where

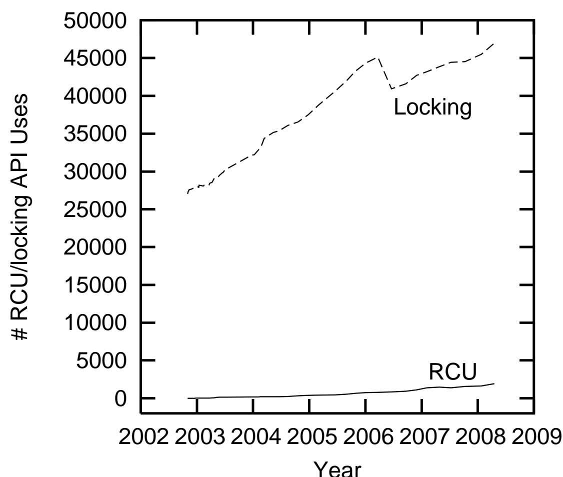

Figure 3: RCU API Usage in the Linux Kernel vs. Locking

available):

1. `rcu_read_lock()`, which marks the beginning of an RCU read-side critical section,

2. `rcu_read_unlock()`, which marks the end of an RCU read-side critical section,

3. `call_rcu()`, which invokes a specified function after all pre-existing RCU read-side critical sections have completed,

4. `kfree()`, which frees an unadorned block of memory,

5. `kmem_cache_free()`, which frees a typed block of memory, and

6. `kmem_deferred_free()`, which frees an untyped block of memory some time after all pre-existing RCU read-side critical sections have completed. (This primitive is not available in Linux, as it turns out to be simpler to open-code it.)

In addition, this variant of the RCU API made use of explict memory barriers [26].

The next section describes how the Linux kernel's RCU API evolved over time.

5. EVOLUTION OF THE RCU API

The RCU API has not evolved continuously, but rather in a manner reminiscent of punctuated equilibrium. Each burst of API change has had its own distinct motivation, including simplicity, distaste for explicit memory barriers, real-time response, memory conservation, networking loads, kernel modules that can be unloaded, and several specific algorithmic needs. Each such burst is covered in chronological order in the following sections, as summarized in Table 1.

Section	Cause of Change
5.1	Simplicity
5.2	Memory Barriers Unloved, Part I
5.3	Real-Time Systems, Part I
5.4	Small-Memory Systems
5.5	Heavy Networking Loads
5.6	Network-Based Denial-of-Service Attacks
5.7	Memory Barriers Unloved, Part II
5.8	Linux Accepts RCU's Namesake
5.9	Real-Time Systems, Part II
5.10	Unloadable Kernel Modules Use RCU
5.11	RCU Readers Must Block
5.12	RCU Readers of Lists Being Reaped
5.13	Summary of RCU API Evolution

Table 1: Evolution of the Linux-Kernel RCU API

5.1 Simplicity

The first force to act on the RCU API was the Linux community's unusually vociferous preference for simplicity. Of course, almost everyone prefers simplicity, but will normally accept a complex solution if it is the only solution known. In contrast, the Linux community often insists that a simpler solution be invented. Such a solution is in fact invented surprisingly frequently. Any technology being proposed for inclusion into Linux can therefore expect to face this challenge.

This demand for simplicity first manifested itself as a demand for an minimal API. Because the early RCU implementations used context switches to determine when pre-existing RCU read-side critical sections had completed, and because early Linux kernels were non-preemptible, the read-side primitives `rcu_read_lock()` and `rcu_read_unlock()` generated no code. This fact accounts for their extreme performance and scalabilty: free is indeed a very good price. However, it also caused the Linux community to insist that these primitives be eliminated (temporarily, as it turns out).

The `kmem_deferred_free()` primitive provided a safe way of freeing RCU-protected data elements in face of concurrent RCU readers. However, because open-coding this primitive turned out very nearly as simple as invoking `kmem_deferred_free()`, this primitive was also eliminated. In essence, an API designed to simplify the developer's life turned out to provide almost no simplification!

On the other hand, this drive for simplicity provided the easier-to-use `synchronize_kernel()` as an alternative to the asynchronous `call_rcu()`. The `synchronize_kernel()` primitive blocks until all pre-existing RCU read-side critical sections complete. However, `synchronize_kernel()` could not completely replace `call_rcu()` because as blocking is not permitted within interrupt handlers.

Although this vociferous demand for simplicity can be painful at times, it can also be quite beneficial, and might in fact account for much of Linux's popularity. After all, if you know only one way to implement something, it is likely to be neither the simplest nor the most optimal approach. The spirited discussions conducted in the Linux community often uncover solutions that are much simpler, faster, and more capable than those originally proposed. And in fact a number of RCU implementations were created during this time, including those from Andi Kleen, Rusty Russell, and

Andrea Arcangeli, with Dipankar Sarma doing the heavy lifting required to compare them and combine their best features into the implementation that was eventually accepted into the Linux kernel [1, 36, 39, 61].

5.2 Memory Barriers Unloved, Part I

Given an operating system with a small developer community and that runs on but a single CPU family, a good strategy might be to ensure that all the developers understand how to correctly use facilities such as memory barriers [11]. Memory barriers enforce ordering of memory references on weakly ordered multiprocessor systems and on systems with aggressive optimizing compilers, both of which can change the order in which code is executed. The correct use of memory barriers is somewhat of a black art, but an art that is critical to the creation of high-performance shared-memory-parallel operating-system kernels. Although only performance-critical technologies are likely to face this memory-barrier challenge, operating-system kernels tend to have more than their share of situations where performance is critical.

However, the above global-understanding strategy fails to scale, both with the number of maintainers and with the number of CPU architectures. In fact, in the case of memory barriers, a number of Linux-kernel maintainers have at times enforced a blanket policy rejecting any patch adding an explicit memory barrier to the Linux kernel. More recently, the `checkpatch.pl` patch-vetting script (found in the `scripts` directory in the Linux-kernel source tree) rejects any patch that adds a memory barrier lacking an explanatory comment.

In the case of Linux, with thousands of developers and more than 20 CPU families, an even better strategy is to do away with the need for explicit memory barriers, preferably by burying them into an easy-to-use higher-level API. To this end, Manfred Spraul recommended adding new RCU-protected linked-list primitives that contained any needed memory barriers [59], relieving Linux kernel developers of the need to consider RCU-related memory barriers, at least when using Linux's circular doubly linked lists. The primitives `list_for_each_entry_rcu()` (which iterates through the specified list), `list_add_rcu()` (which adds an item to the beginning of the specified list), `list_add_tail_rcu()` (which adds to the end), and `list_del_rcu()` (which deletes the specified item) were duly added to the RCU API.

These additions eliminated the need for explicit memory barriers in code using RCU-protected lists, freeing the kernel developers from the need to concern themselves with the memory model provided by the underlying hardware. This represented a great leap forward in readability of code using RCU-protected lists. Code using other types of RCU-protected data structures was dealt with later, as described in Section 5.7.

5.3 Real-Time Systems, Part I

Specially modified versions of Linux have been used for real-time computing since the mid-to-late 1990s. Of course, each such version of Linux was subtly different, and considerable development effort was expended creating very similar functionality. This situation called out for the addition of real-time functionality to the mainline Linux kernel.

An important step towards this goal occurred early in the 2.5 development effort with the introduction of the `CONFIG_PREEMPT` configuration option, which improved the Linux kernel's real-time response by introducing kernel preemption. Unfortunately, all of the Linux RCU implementations at that time assumed a non-preemptible kernel, as they relied on context switches to determine when all pre-existing RCU readers had completed. A simple fix was to cause RCU readers to disable preemption across RCU read-side critical sections, reintroducing the (nestable) `rcu_read_lock()` and `rcu_read_unlock()` primitives. The outermost `rcu_read_lock()` and `rcu_read_unlock()` disable and enable preemption, respectively.

This change enabled RCU to be used in real-time environments that required millisecond-scale scheduling latencies. It is reasonable to expect that many technologies will face real-time response challenges, particularly those based on backoff or retry techniques.

5.4 Small-Memory Systems

Linux is used on numerous embedded platforms, which often have tight constraints on system memory, particularly for platforms that are powered by batteries. These small-memory platforms will likely pose special challenges for any technology constructed using infinite-memory assumptions. On such platforms, Linux's circular doubly linked lists consume two pointers worth of memory per hash bucket, which can be problematic for large hash tables. Such large hash tables can also be problematic on large-memory systems with small CPU caches.

Andi Kleen therefore implemented `hlist`, which is a linear doubly linked list. Although each element still requires two pointers, one in each direction, each bucket of a hash table need only have a single pointer to the first element of the list, as opposed to the pair of pointers required for a list header for a circular doubly linked list. This reduction from two pointers per hash bucket down to a single pointer per hash bucket halves the memory consumed by the bucket array making up a large hash table, which can be the dominating factor in lightly loaded hash tables optimized for fast lookup. However, it also required Andi to also introduce the to the RCU API the primitives `hlist_for_each_entry_rcu()` (which iterates through the specified `hlist`), `hlist_del_rcu()` (which deletes the specified `hlist` element), `hlist_add_after_rcu()` (which adds a new entry after the specified `hlist` element), `hlist_add_before_rcu()` (which adds a new entry before the specified `hlist` element), and `hlist_add_head_rcu()` (which adds a new entry to the head of the specified `hlist`).

The addition of these APIs greatly improved performance of some filesystem workloads on systems with small CPU caches.

5.5 Heavy Networking Loads

As noted in Section 1, the Linux 2.4 networking stack used a `brlock` primitive based on per-CPU reader-writer locks. Updaters write-acquired and then immediately write-released this lock, guaranteeing that all pre-existing readers had completed, a use that is similar to RCU. Steve Hemminger therefore replaced this `brlock` primitive with RCU, introducing `synchronize_net()` to ease the transition (and also to ease a transition back, if need be). This primitive was retained after the transition proved successful, which permitted `brlock` to be eliminated. However, `synchronize_net()` remains a useful documentation aid, despite its being

simply a synonym for synchronize_kernel(): both primitives wait for all pre-existing RCU read-side critical sections to complete.

This change reduced the number of distinct synchronization primitives in the Linux kernel by eliminating brlock.

5.6 Network-Based Denial-of-Service Attacks

Enterprise systems are often protected from network-based attacks by firewalls. However, this strategy fails for Linux, because Linux often *is* the firewall. Robert Olsson found that extremely heavy networking loads from possible network-based denial-of-service attacks could, among other things, indefinitely postpone critical RCU-infrastructure operations ("grace periods"), resulting in exhaustion of free memory and subsequent system hangs. Dipankar Sarma worked with Robert to design a "bottom-half" variant of RCU that solved this problem, allowing Robert to pursue other problems exposed by such attacks. This solution added the rcu_read_lock_bh(), rcu_read_unlock_bh(), and call_rcu_bh() primitives to the RCU API. These new primitives are analogous to the rcu_read_lock(), rcu_read_unlock(), and call_rcu() primitives that were described in Section 4, the main difference being that the new call_rcu_bh() primitive completes much more quickly than the older call_rcu() primitive, reducing the amount of memory waiting for such completions, in turn preventing the denial-of-service attack from exhausting system memory.

Merging bottom-half RCU into the implementation of the normal RCU API proved infeasible due to the higher overhead of the new bottom-half read-side primitives, so the Linux kernel retains both the normal APIs (for example, rcu_read_lock()) and the bottom-half variants (for example, rcu_read_lock_bh()).

The addition of these bottom-half RCU primitives was a significant step in enabling Linux to survive network-based denial-of-service attacks, though we can expect such attacks to continue increasing in sophistication. Linux's heavy use in networking infrastructure can be expected to pose significant challenges to a broad range of technologies that might be put forward for inclusion into Linux.

5.7 Memory Barriers Unloved, Part II

Although the primitives described in Sections 5.2 and 5.4 eliminated the need for explicit memory barriers in RCU-protected linked lists, increasingly complex data structures appeared over time, including the RCU-protected trees introduced by Robert Olsson [50] and by Nick Piggin [51]. These more-complex RCU-protected data structures motivated eliminating explicit memory barriers for arbitrary RCU-protected data structures, which required addition of two more members of the RCU API, rcu_dereference() and rcu_assign_pointer(). The rcu_assign_pointer() primitive publishes a new data structure through an RCU-protected pointer, while rcu_dereference() subscribes to a previously published data structure.

The addition of these two primitives further reduced the need for explicit memory barriers in code using RCU, again improving the readability of such code.

5.8 Linux Accepts RCU's Namesake

The acronym "RCU" stands for "read-copy update". This name was chosen because RCU readers can access the RCU-protected data structure concurrently with copy-mediated updates. RCU's namesake is therefore the use case where the RCU updater carries out the following sequence of steps: (1) allocate a new element, (2) copy the old element to the new element, (3) update the new element, and finally (4) link the new element into the data structure in place of the old one. But as Murphy would have it, this use case turned out to be quite rare. Instead, most RCU updaters simply add elements to and delete elements from RCU-protected data structures, as opposed to updating existing elements.

It was not until the 2.6.11 kernel that Kaigai Kohei needed to use this technique to implement the Security-Enhanced Linux (SELinux) access-vector cache [48]. The two primitives list_replace_rcu() (which replaces an existing list element) and hlist_replace_rcu() (which replaces an existing hlist element) were therefore added to the RCU API.

These primitives have since found use in a number of other situations, providing a valuable addition to the Linux kernel developer's toolbox.

The key point of this particular change is that the Linux community is likely to continue its practice of accepting only those portions of a given technology that are immediately useful. In fact, there are a number of Linux-community members who put significant effort into pruning the Linux source base of code that is unused or otherwise unnecessary. Therefore, new technologies will frequently need to be introduced into Linux in an incremental fashion.

5.9 Real-Time Systems, Part II

The real-time functionality described in Section 5.3, although useful, proved insufficient for more-aggressively real-time systems. Therefore, a number of projects worked to improve Linux's real-time response [4]. Ingo Molnar's -rt patchset prevailed, but required that RCU read-side critical sections be preemptible [5], invalidating the basic RCU assumption that read-side critical sections be non-preemptible.

Although a rough-and-ready workaround was generated in due time [6], this workaround was prone to indefinite-postponement failures. Furthermore, a number of developers had used RCU strictly for its ability to wait until all interrupt and NMI handlers have completed, an ability that was an unintended side effect. This situation resulted in the deprecation of the synchronize_kernel() primitive in favor of the new synchronize_rcu() and synchronize_sched() primitives, the former for its conventional use (waiting for pre-existing RCU read-side critical sections to complete) and the latter for waiting for pre-existing preemption-disabled code sections (including interrupt and NMI handlers) to complete. In addition, the preempt_disable() and preempt_enable() primitives became members of the RCU API, as did a number of other primitives that disable and re-enable preemption.

In addition, this effort required substantial changes to the RCU implementation [29]. These changes helped to greatly improve the Linux kernel's real-time latencies, achieving latencies on the order of a few tens of *micro*seconds on quad-CPU blade-based systems. Of course, the need to achieve such aggressive scheduling latencies will likely pose severe challenges for any technology that has been developed without consideration of real-time response requirements.

5.10 Unloadable Kernel Modules Use RCU

The Linux kernel supports loadable kernel modules, which allow a small base kernel to dynamically load only that func-

tionality required by the system it is running on, conserving memory while also preserving the ability to adapt to a wide variety of hardware configurations. The Linux kernel also allows such modules to be *un*loaded, which removes the unloaded module's code and data from the kernel.

Such a module might use RCU's asynchronous `call_rcu()` interface, which can result in some of that module's functions ("RCU callbacks") being invoked at a later time, once all pre-existing RCU read-side critical sections have completed. Clearly, that module's code and data must remain in memory until all such RCU callbacks have been invoked, which means that module unloading must be delayed until after all of that module's RCU callbacks have completed. This requirement can be expected to affect any technology that relies on deferred processing.

When an RCU-using module appeared, an `rcu_barrier()` primitive [32], originally developed for ReiserFS by Dipankar Sarma, was added to the Linux 2.6.15 kernel. This primitive blocks until all RCU callbacks created by earlier calls to `call_rcu()` have been invoked, allowing the module to be safely unloaded.

This primitive permits Linux kernel modules using the `call_rcu()` primitive to be dynamically unloaded.

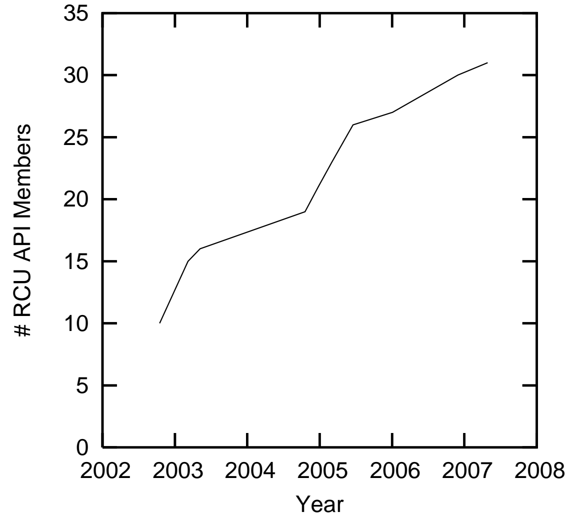

Figure 4: RCU API Growth Over Time

5.11 RCU Readers Must Block

People have asked for RCU readers to be able to block for well over a decade. This request has invariably indicated a lack of understanding of RCU.

That is, it indicated a lack of understanding of RCU until early 2006, when a group of Linux kernel developers really did need RCU readers to block. This meant creating a variant of RCU (named "SRCU") that permitted generalized blocking in read-side critical sections, but while avoiding the memory-exhaustion scenarios that would normally ensue [28]. Because the resulting implementation required slight changes to the RCU API, this also required adding the `srcu_read_lock()`, `srcu_read_unlock()`, and `synchronize_srcu()` primitives to the Linux 2.6.19 kernel. These primitives are roughly analogous to the `rcu_read_lock()`, `rcu_read_unlock()`, and `synchronize_kernel()` primitives described in Section 4.

Addition of these primitives permitted RCU to be used in situations requiring RCU's extremely low read-side overheads, but where readers might occasionally need to block. An example of such a situation would be a heavily used in-memory cache of a disk-based data structure with a high hit rate. The design of such a system can be simplified by use of SRCU without sacrificing performance or scalability.

Although this change was specific to RCU, it clearly illustrates how the wide usage of the Linux kernel can force unexpected changes into a given technology.

5.12 RCU Readers of Lists Being Reaped

One of the more unconventional features of RCU is that it allows readers and updaters to make forward progress even when running concurrently. This property is key to the high performance, unlimited scalability, and $O(1)$ computational complexity for RCU's read-side primitives, but can provide interesting challenges in some situations.

In particular, Corey Minyard needed to remove all elements of an RCU-protected circular doubly linked list with a single operation. Of course, the fact that RCU readers run concurrently with updaters means that readers might be referencing such a list at the time of full-list removal. Such removal must therefore be performed carefully, using the following steps:

1. Adjust the list so that new readers perceive it to be empty, but so that old readers still find the list header so that they terminate correctly upon reaching the end of the list.

2. Wait for all old readers to complete their scan of the list. RCU provides primitives such as `synchronize_rcu()` for this purpose.

3. Complete the removal process, linking the list into a new list header so that it may be processed further.

This process was packaged into the `list_splice_init_rcu()` primitive [46]. As with the change described in Section 5.11, this change is specific to RCU, but again demonstrates how the wide usage of the Linux kernel can force changes into a given technology.

5.13 Summary of RCU API Evolution

Seven years of exposure to Linux increased the size of the RCU API from the initial six components (seven if one counts explicit memory barriers) to 31 components as of the Linux 2.6.24 kernel, as shown in Figure 4. Any way you calculate it, this is an extremely large increase in size, an increase that was completely unexpected, given that the initial DYNIX/ptx RCU API had been running in production supporting large datacenter workloads for well over five years beforehand. Of course, the Linux kernel has a variety of internal software environments, including process context, interrupt context, and so on, which results in more than 50 API members for simple locking. However, DYNIX/ptx had a similar variety of internal software environments.

This situation therefore motivated a search for the reasons why a technology so well-suited for datacenter workloads should require so much change upon being introduced into

Linux. The lessons learned during this search are discussed in the next section.

6. LESSONS LEARNED

The experience of adapting RCU for Linux taught a number of valuable lessons. The first set of lessons pertained to working with free/open source software (FOSS) communities, and are covered in Section 6.1. Subsequent lessons are more specific to the evolution of the RCU API, including the wide range of workloads and platforms that Linux supports, the use of Linux in networking infrastructure, the economics of Linux's large community, and the high degree of innovation fostered by the Linux community. These RCU-specific lessons are covered by sections 6.2 through 6.5.

6.1 The Workings of FOSS Communities

Although my five-to-ten years with the Linux community by no means qualifies me as an expert on the FOSS communities, my combined experience with academia, research organizations, proprietary software development groups, standards organizations, and the Linux community does provide an interesting vantage point. This section leverages this vantage point to describe how one might work with the Linux community, drawing analogies with situations that might be more familiar to many readers.

For example, consider a researcher working with a motivated proprietary software vendor. Such a vendor might make a particular developer responsible for implementing the results of the research in the product. This sort of relationship can also happen in FOSS projects, for example, when Alexey Kuznetsov implemented stochastic fairness queuing [23] in Linux—my only involvement in this process was to send Alexey a copy of my paper. Although this process can work extremely well, it depends on a developer being ready, willing, and able to do the necessary work. Such a developer might or might not be available.

FOSS projects permit another approach, namely, the researcher downloading the project, making the necessary modifications, conducting experiments, and reporting the results. On the one hand, this approach eliminates the dependency on the developer, but on the other hand it does nothing to foster inclusion of the researcher's modifications into the FOSS project's official repository. Such inclusion does require more work, but can greatly ease collaboration and do much to advance the overall research agenda [3].

Although different projects have different criteria for inclusion, even within the confines of the Linux community, it is well worthwhile to consider the following questions:

1. Does the value of your contribution exceed the cost of its inclusion in the FOSS project and its maintenance thereafter?

2. Are there community members who are enthusiastic about your contribution?

3. If community members invest the time and effort required to include your contribution, are you willing to make a compensating investment of your time and effort into the project?

These questions are considered in the following sections.

6.1.1 *Code as Liability*

Source code is commonly considered to be an asset rather than a liability. After all, it can take great effort to produce, and can sometimes be sold for considerable amounts of money. The fallacy of the "source code as asset" viewpoint can be seen by considering the fate of those unfortunates who purchase a body of source code, but fail to retain the services of that code's developers. Without the developers, it is often impossible to properly service, support, and maintain the source code, possibly resulting in bankruptcy or legal action.

It is often better to instead view the source code as a *liability* that one might be willing to incur only if the corresponding functionality and performance is sufficiently valuable to the rest of the community. Only then can the community be expected to continually invest the time and effort required to fix the inevitable bugs, adapt the code to the inevitable changes in the enclosing software environment, support the users, add features demanded by those users, and so on.

In short, by incorporating your source-code contribution, the project will be making an ongoing investment of time in effort in order to give you your 15 minutes of fame and glory. What will you give the project in return?

6.1.2 *Community Enthusiasm*

The community might be willing (perhaps even happy) to accept your source-code contribution if enough community members are enthusiastic about the corresponding functionality or performance. These members might be either developers or users. This of course leads to the question of how to generate the necessary enthusiasm.

One approach is to identify the community's enthusiasm ahead of time, and then selecting a project that provides something that the community badly wants or needs. This approach can have the further advantage of garnering community help and support during your research and implementation, and greatly easing the inclusion process. You should nevertheless take care to learn the community's culture and coding style, and be prepared to interact with the community to shape your contribution into a form consistent with community expectations. You might also need to provide training and documentation to the community in order to ease their learning and support burden. With the proper preparation, this approach can be quite gratifying to both the researcher and the community.

Another approach is to "sell" your contribution to the community, either before or after creating it. Although I have seen situations where this has worked well, I must defer to those who have the relevant sales/marketing skills.

6.1.3 *Compensating Investments*

In order to get your contribution included into a FOSS project, you might need to invest some of your own time and effort to compensate the community for the time and effort that they will incur supporting and maintaining your contribution. This is no different than the situation with any given academic journal: authors are often expected to contribute their time and effort acting as reviewers, and may in addition need to be active members of the research community that the journal serves.

FOSS communities also value review: contributors may be expected to review and comment on the contributions of others, thereby increasing the quality and robustness of

the FOSS project. Such efforts also build trust: a history of on-point reviews, debugging, bug fixes, and contributions that are consistent with the aims and culture of the project will tend to increase the stature of the contributor within that community.

However, the typical FOSS review process differs from the typical academic process in a number of ways:

1. FOSS reviews are not anonymous.

2. Contributors can (and usually do) respond to FOSS reviews.

3. Multiple contributors with similar goals will often collaborate to produce a single contribution that meets all their goals. This can also happen in academic circles, but the open nature of the FOSS review process more directly encourages such alliances.

4. FOSS reviews can often be more harsh in tone than those in academic circles, though the effects of this harshness are balanced by the ability to respond directly and immediately.

A developer who contributes to the review process (including debugging and bug-fix efforts) will earn the community's trust, and thus meet less resistance to future contributions. This situation is not all that different from an academic community. As is the case with academic communities, trust in one community does not necessarily immediately translate to trust in another. In particular, trust within an academic community does not necessarily automatically translate to trust within the corresponding FOSS community, and vice versa. There are of course exceptions, one prominent example being Van Jacobson's high level of esteem within the Linux-kernel networking community.

Within FOSS communities, as within most organizations, trust is key. And, also like most organizations, one's level of trust is determined by one's words and (especially) actions over time.

6.1.4 FOSS Communities Summary

The advent of FOSS communities is critically important, as they have the potential of offering a publicly available bridge between research and practice by offering researchers a way to test their ideas in real-world software systems. The fact that the FOSS projects are publicly available also allows other researchers to easily replicate results, in happy contrast to proprietary software. FOSS communities also offer the prospect of academic research being applied to practice in a timely manner, increasing the impact of this research.

The following sections describe lessons that are more specific to the Linux kernel.

6.2 Wide Range of Workloads and Platforms

A key lesson learned from the RCU experience is that Linux runs an incredible variety of workloads on a wide variety of platforms, including embedded systems, cell phones, desktops, network processors, servers, and supercomputers. Each of these platforms brings its own set of issues and requirements, a number of which affected RCU's design and implementation. In particular, Linux runs real-time workloads, which required significant changes to RCU, as described in Sections 5.3 and 5.9. Furthermore, Linux is used in embedded systems with small memory, which also affected RCU as described in Section 5.4. Therefore, any technology that has been developed in a protected niche is likely to require substantial changes in order to operate safely and effectively in the less-protected Linux environment.

One advantage of the Linux kernel's FOSS nature is that not only is the source code freely available, but that design discussions are also freely available. In fact, design discussions are open to general participation, the only hard requirement being an ability to read and write English, but not necessarily to converse in spoken English. To their credit, many researchers are already taking advantage of this openness, using the Linux kernel as a platform for their research. It is hoped that such collaborations will help to narrow the researcher/practitioner divide, increasing the impact of research, while speeding the evolution of the Linux kernel.

6.3 Networking Infrastructure

In addition, Linux is heavily used in network infrastructure. As noted earlier, this means that Linux cannot be protected by a firewall because Linux *is* the firewall. Therefore, Linux must efficiently process networking loads that might bring a machine designed for a carefully cossetted datacenter environment to its knees. This requirement had significant effects on the RCU API, as noted in Sections 5.5 and 5.6.

6.4 Software Development Economics

Economics also plays a part. For example, there were only a few tens of kernel developers working on DYNIX/ptx, perhaps 50 at most. This means that an innovation in DYNIX/ptx that saved 1% of each developer's time would save at most six person-months per year. If the innovation itself required one person-year to implement, two years would be required to recover the cost of implementation. Because there would be very likely be higher-payoff investments of kernel-developer time, such an innovation might never be implemented.

In contrast, many thousands of people work with the Linux kernel. Even if we restrict our attention to people whose changes were accepted into the 2.6.24 release of the Linux kernel, we end up with 950 [7]. This smaller number does not count the number of people who read kernel code to provide support, to fix bugs in older versions of Linux, or to understand how best to implement Linux-based applications. However, if we nevertheless take 1,000 as an estimate of the number of full-time equivalent effort spent on the Linux kernel, our one-person-year innovation could save ten person-years per year, recovering the investment in not much more than a month.

This economic effect had a large effect on the RCU API, for example, in the form of the RCU-based list-manipulation primitives discussed in Section 5.2. Such primitives would likely have remained open-coded in operating systems with smaller communities. Because such primitives arguably increase productivity, it is interesting to speculate on the long-term prospects of Linux compared to kernels with communities having fewer active members. Similar effects on the RCU API are described in Sections 5.1 and 5.7.

6.5 Linux-Community Innovation

Arguably the largest effect was due to the highly innovative group of developers in the Linux community, who applied RCU in a number of ways, some completely unan-

ticipated. This is not necessarily to say that Linux developers are more innovative than those in other environments (though the Linux community by no means lacks extremely innovative and talented developers), but rather that the widespread sharing of viewpoints in the open design process does much to foster innovation. Some of the effects of this innovation on RCU are covered in 5.10, 5.11, and 5.12. In addition, the drive towards simplicity discussed in Section 5.1 forced incremental adoption of RCU, as described in Section 5.8.

One can argue that Linux-community innovation will continue to drive change into RCU, and into much else besides. Such future prospects for RCU are taken up in the next section.

7. FUTURE PROSPECTS

Has every conceivable change to RCU already been made, or will RCU continue to evolve? This second view seems most likely. And, although it is said that the best way to predict the future is to invent it, it is also true that thinking a bit about the future can help identify useful directions for invention. The following sections therefore consider RCU-related issues in the areas of power consumption, massively multicore systems, real-time response, new synchronization primitives, and, finally, RCU API orthogonality.

7.1 Power Consumption

There has been some progress over the past year better integrating preemptible RCU (the real-time variant) with the dynamic ticks power-conservation feature (selected by the `CONFIG_NO_HZ` Linux kernel parameter). The idea behind dynamic ticks is that idle CPUs should forgo the scheduling-clock interrupt, allowing them to reach deeper "sleep states", thus better conserving power.

Unfortunately, older versions of preemptible RCU would awaken all CPUs, including those in sleep states, whenever an RCU updater required an RCU grace period to elapse. This problem was addressed by a new interface between dynamic ticks and RCU that avoids waking sleeping CPUs [55]. This same technique could profitably be applied to other implementations of RCU as well.

7.2 Massively Multicore Systems

Increased CPU counts might require that the scalability of RCU's grace-period detection be improved (and much else besides). Although the original "classic" implementation of RCU has been modified to run efficiently on a 512-CPU Altix machine [60], other variants would require more work to run well on systems with large numbers of CPUs. Spraul's hierarchical approach has proven successful in other environments, and is likely to be the method of choice on very large systems.

However, if per-chip CPU counts were to rise without limit, then it is entirely possible that memory bandwidth and on-chip cache size will fail to increase sufficiently to permit some workloads from taking full advantage of all of the CPUs. A rational response to this situation might well be to simply avoid using some fraction of these (presumably extremely low-cost) CPUs. However, power-consumption issues might motivate high CPU utilization in order to minimize the total number of systems required for a given workload. If such a situation arises, memory conservation might well be required on large machines as well as on tiny embedded systems.

Such a need for memory conservation would further increase the motivation to shrink the `rcu_head` structure by tabulating RCU callback functions and encoding the resulting table index into this structure's pointer to next [40], thereby reducing the memory footprint of heavily replicated RCU-protected data structures, such as those making up the directory-entry cache.

Of course, increased focus on memory footprint would undoubtably affect many other aspects of the Linux kernel as well.

7.3 Real-Time Response

The need for improved real-time response has already had a large effect on RCU, as discussed in Sections 5.3 and 5.9. It seems likely that increasingly aggressive real-time response constraints will be applied to additional areas within the Linux kernel. For example, the advent of low-cost, high-capacity, and low-latency solid-state storage removes the traditional seek-time and rotational-latency barriers to real-time mass-storage access. This turn of events may well raise interest in real-time mass storage access, both with and without filesystems. RCU might well have a role to play in this arena, which in turn might require changes to RCU, either in the RCU infrastructure itself or in the way that RCU is used and applied.

Increasing bandwidths and decreasing latencies of data-communications hardware seem quite likely to have the same effect on Linux's protocol stacks.

In addition, as multi-core CPUs continue to gain popularity in real-time systems, we may possibly see a need for RCU in user applications, at least for those applications written in non-garbage-collected languages such as C and C++.

However, it is important to note that real-time RCU work to date has focused almost entirely on RCU's read-side primitives. When performing RCU updates, it is often necessary to defer destructive actions (such as freeing memory previously removed from an RCU-protected data structure) until all pre-existing RCU readers have completed. The length of time that such actions must be deferred is known as an "RCU grace period". In some RCU implementations, these grace periods can extend for tens of milliseconds [12], which can sometimes be inconveniently long [2, 65].

This situation inspired a new implementation of RCU that favors updates, named QRCU [30, 33, 49]. However, QRCU has not yet been accepted into the Linux kernel, and might or might not ever be. There has also been discussion of mechanisms to expedite grace-period computation for existing RCU implementations, perhaps providing RCU API members such as `synchronize_rcu_expedited()`. Many open questions remain in this area.

7.4 New Synchronization Primitives

RCU is a specialized synchronization primitive that is intended for use in read-mostly situations or in situations requiring deterministic readers. Because RCU is not so foolish as to attempt to be all things to all developers in all situations, it must interface conveniently and efficiently with other synchronization mechanisms. RCU has a long history of interfacing well with lock-based updates, and has also seen extensive use interfacing with atomic-instruction-based updates. There have also been some successful experiments in-

terfacing RCU with non-blocking-synchronization-based updates [24].

More recently, researchers have attempted to interface RCU to transactional-memory-based update, however, early attempts resulted in RCU readers unnecessarily aborting transactional updates [54]. More recent work avoids this problem, but restricts the transactional-memory implementation. Should transactional memory graduate from research to practice, more work will be needed to interface it efficiently with RCU.

7.5 RCU API Orthogonality

The RCU API is currently not orthogonal: a tabulation of the RCU API shown in Tables 2 and 3 reveals several missing members, namely `synchronize_rcu_bh()` (the synchronous counterpart to `call_rcu_bh()`), `call_rcu_sched()` (the asynchronous counterpart to `synchronize_sched()`), `rcu_barrier_sched()` (the barrier counterpart to `call_rcu_bh()`), and `rcu_barrier_bh()` (the barrier counterpart to `call_rcu_sched()`).

Of course, orthogonality alone is not necessarily sufficient to motivate adding code to the Linux kernel. However, a need for `call_rcu_sched()` and `rcu_barrier_sched()` has recently arisen [8], which might result in adding this pair of interfaces.

It is of course unwise to measure the capability of a technology such as RCU by the number of elements in its API. In fact, a reduction in the size of the RCU API would undoubtably be quite welcome. There is at least a theoretical possibility of substantial convergence [14], but the question remains open as to whether SRCU can subsume all the functionality of the other forms of RCU, while still providing adequate performance.

8. CONCLUSIONS

Introducing RCU into Linux forced dramatic and unexpected change into RCU, for example, expanding the RCU API from six members in 2001 to more than 30 members in 2007. There is reason to believe that Linux will continue forcing change into RCU, in particular, one welcome change would be a slower increase, or better yet a decrease, in the size of the RCU API.

These additions occurred a few at a time, and were motivated by the increasingly wide range of workloads and platforms that Linux supports, the use of RCU in networking infrastructure, the economics of Linux's large community, and the high degree of innovation fostered by the Linux community. To give only one example, although datacenters tend to be protected by firewalls, Linux cannot assume firewall protection because Linux *is* the firewall. Therefore, technologies developed in protected datacenter environments will likely require significant overhauls when being adapted to Linux, and it in fact seems likely that Linux will require continued changes to RCU. It seems likely that RCU's experience applies to other technologies that might be introduced to the Linux kernel.

Nevertheless, the opportunity to work on an artifact as widely used as is Linux is a rare privilege, and working on Linux's RCU implementation continues to be a deeply rewarding learning experience.

And, as the following section demonstrates, it is also an ongoing learning experience.

9. EPILOG

This section presents ongoing experience with RCU subsequent to initial publication of this paper in mid-2008 [43].

9.1 2.6.27 Linux Kernel

This release added the `call_rcu_sched()`, `rcu_barrier_sched()`, and `rcu_barrier_bh()` RCU API members, and predicted in Section 7.5.

9.2 2.6.28 Linux Kernel

One welcome change involved an actual reduction in the size of RCU's API with the removal of the `list_for_each_rcu()` primitive. This primitive is superseded by `list_for_each_entry_rcu()`, which has the advantage of iterating over structures rather than iterating over the pointer pairs making up a `list_head` structure (which, confusingly, acts as a list element as well as a list header). This change was accepted into the 2.6.28 Linux kernel.

Unfortunately, the 2.6.28 Linux kernel also added some RCU API members, namely, `rcu_read_lock_sched()` and `rcu_read_unlock_sched()`, which Section 7.5 somehow failed to anticipate. These APIs were added to promote readability. In the past, primitives to disable interrupts or preemption were used to mark the RCU read-side critical sections corresponding to `synchronize_sched()`. However, this practice led to bugs when developers removed the need to disable preemption or interrupts, but failed to notice the need for RCU protection. Use of `rcu_read_lock_sched()` will help prevent such bugs in the future.

9.3 2.6.29 Linux Kernel

A new more-scalable implementation, dubbed "Tree RCU", replaces the flat bitmap with a combining tree, and was accepted into the 2.6.29 Linux kernel. This implementation was inspired by the ever-growing core counts of modern multiprocessors (see Section 7.2, and is designed for many hundreds of CPUs. Its current architectural limit is 262,144 CPUs, which the authors (perhaps naïvely) believe to be sufficient for quite some time. This implementation also adopts preemptible RCU's improved dynamic-tick interface (see Section 7.1).

Mathieu Desnoyers added `rcu_read_lock_sched_notrace()` and `rcu_read_unlock_sched_notrace()`, which are required to permit the tracing code in the Linux kernel to use RCU. Without these APIs, attempts to trace RCU read-side critical sections lead to infinite recursion.

Eric Dumazet added a new type of RCU-protected list that allows single-bit markers to be stored in the list pointers. This type of list enables a number of lockless algorithms, including some reported on by Maged Michael [45]. Eric's work adds the `hlist_nulls_add_head_rcu()`, `hlist_nulls_del_rcu()`, `hlist_nulls_del_init_rcu()`, and `hlist_nulls_for_each_entry_rcu()`. It also adds a new structure named `hlist_nulls_node`.

Although it is strictly speaking not part of the Linux kernel, at about this same time, Mathieu Desnoyers announced his user-space RCU implementation [9]. This is an important first step towards the real-time user-level RCU implementation discussed in Section 7.3.

9.4 2.6.31 Linux Kernel

Jiri Pirko added `list_entry_rcu` and `list_entry_first_rcu()` primitives that encapsulate the `rcu_dereference()`

Category	Publish	Retract	Subscribe
Pointers	`rcu_assign_pointer()`	`rcu_assign_pointer(...,NULL)`	`rcu_dereference()`

Lists	`list_add_rcu()`
	`list_add_tail_rcu()`
	`list_replace_rcu()`
		`list_del_rcu()`	`list_for_each_entry_rcu()`

Hlists	`hlist_add_after_rcu()`
	`hlist_add_before_rcu()`
	`hlist_add_head_rcu()`
	`hlist_replace_rcu()`
		`hlist_del_rcu()`	`hlist_for_each_entry_rcu()`

Table 2: RCU Publish and Subscribe Primitives

	RCU Classic	RCU BH	RCU Sched	SRCU

Read-Side Primitives	`rcu_read_lock()`
	`rcu_read_unlock()`
		`rcu_read_lock_bh()`
		`rcu_read_unlock_`
		`bh()`
			`preempt_disable()`
			`preempt_enable()`
			and friends
				`srcu_read_lock()`
				`srcu_read_unlock()`

Update-Side Primitives (Synchronous)	`synchronize_rcu()` `synchronize_net()`		`synchronize_sched()`	`synchronize_srcu()`
Update-Side Primitives (Asynchronous)	`call_rcu()`	`call_rcu_bh()`		N/A
Update-Side Barriers	`rcu_barrier()`			N/A

Table 3: RCU Read- and Update-Side Primitives

RCU-subscription primitive into higher-level list-access primitives, which will hopefully eliminate a class of bugs.

In addition, the "Tree RCU" implementation was upgraded from "experimental" status.

9.5 2.6.32 Linux Kernel

Perhaps the largest change in this version of the Linux kernel is the removal of the old "Classic RCU" implementation. This implementation is superseded by the "Tree RCU" implementation.

This version saw a number of other changes, including:

1. The `synchronize_rcu_expedited()` RCU API member discussed in Section 7.3, along with `synchronize_sched_expedited()` and `synchronize_rcu_bh_expedited()`. These primitives are equivalent to their non-`_expedited()` counterparts, except that they take measures to expedite the grace period.

2. Add preemptible-RCU functionality to the "Tree RCU" implementation, thus removing one obstacle to real-time response from large multiprocessor machines running Linux.

3. This new "Tree Preemptible RCU" implementation obsoletes the old preemptible RCU implementation, which was removed from the Linux kernel.

9.6 What Comes After 2.6.32?

Although the "Tree RCU" implementation is quite capable, it does have a larger memory footprint than does "Classic RCU". A new RCU implementation, "Tiny RCU" (AKA "RCU: the Bloatwatch Edition"), has been designed, coded, and tested for small uniprocessor embedded systems. Thanks to a timely forward-port by David Howells, "Tiny RCU" is being considered for inclusion in 2.6.33.

Avi Kivity requested a version of `synchronize_srcu()` with expedited grace periods. An implementation has been designed, coded, and tested (by Marcelo Tosatti), and is being considered for inclusion in 2.6.33.

Acknowledgements

We are indebted to a number of the attendees of the 2008 Linux Developer Symposium - China for many valuable discussions stemming from a presentation on this topic, to Balbir Singh, David Kleikamp, and Matt Helsley for productive discussions, to Amos Waterland, Bryan Jacobson, Mark Brown, Ram Pai, Jonathan Walpole, and Val Henson for their careful review and thoughtful comments, and to Daniel Frye and Kathy Bennett for their support of this effort.

Of course, Paul also owes thanks to his upstream maintainer, Ingo Molnar, and to Linus Torvalds for sharing the Linux kernel with all of us.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851.

Legal Statement

This work represents the views of the authors and does not necessarily represent the view of IBM.

Linux is a copyright of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

10. REFERENCES

[1] Arcangeli, A., Cao, M., McKenney, P. E., and Sarma, D. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)* (June 2003), USENIX Association, pp. 297–310. Available: `http://www.rdrop.com/users/paulmck/RCU/rcu.FREENIX.2003.06.14.pdf` [Viewed November 21, 2007].

[2] Axboe, J. Re: [patch] cpufreq: mark `cpufreq_tsc()` as `core_initcall_sync`. Available: `http://lkml.org/lkml/2006/11/17/56` [Viewed May 28, 2007], November 2006.

[3] Ben-Yehuda, M., and Hensbergen, E. V. Open source as a foundation for systems research. *SIGOPS Oper. Syst. Rev. 42*, 1 (2008), 2–4.

[4] Corbet, J. Approaches to realtime Linux. Available: http://lwn.net/Articles/106010/ [Viewed March 25, 2008], October 2004.

[5] Corbet, J. Realtime preemption, part 2. Available: http://lwn.net/Articles/107269/ [Viewed March 25, 2008], October 2004.

[6] Corbet, J. Realtime preemption and read-copy-update. Available: http://lwn.net/Articles/129511/ [Viewed October 17, 2005], March 2005.

[7] Corbet, J. 2.6.24 - some statistics. Available: http://lwn.net/Articles/264440/ [Viewed March 25, 2008], January 2008.

[8] Desnoyers, M. Re: [patch 1/2] Linux kernel markers - support multiple probes. Available: http://lkml.org/lkml/2007/12/20/244 [Viewed March 27, 2008], December 2007.

[9] Desnoyers, M. [RFC git tree] userspace RCU (urcu) for Linux. Available: http://lkml.org/lkml/2009/2/5/572 git://lttng.org/userspace-rcu.git [Viewed February 20, 2009], February 2009.

[10] 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].

[11] Gharachorloo, K. Memory consistency models for shared-memory multiprocessors. Tech. Rep. CSL-TR-95-685, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA, December 1995. Available: http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-9.pdf [Viewed: October 11, 2004].

[12] 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].

[13] Hart, T. E., McKenney, P. E., Brown, A. D., and Walpole, J. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput. 67*, 12 (2007), 1270–1285.

[14] Hellwig, C. Re: [-mm PATCH 1/4] RCU: split classic rcu. Available: http://lkml.org/lkml/2006/9/28/160 [Viewed March 27, 2008], September 2006.

[15] Hennessy, J. P., Osisek, D. L., and Seigh II, J. W. Passive serialization in a multitasking environment. Tech. Rep. US Patent 4,809,168 (lapsed), US Patent and Trademark Office, Washington, DC, February 1989. Available: http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.htm&r=1&f=G&l=50&s1=4809168.PN.&OS=PN/4809168&RS=PN/4809168 [Viewed December 3, 2007].

[16] Houston, J. [RFC&PATCH] Alternative RCU implementation. Available: http://lkml.org/lkml/2004/8/30/87 [Viewed February 17, 2005], August 2004.

[17] Hsieh, W. C., and Weihl, W. E. Scalable reader-writer locks for parallel systems. In *Proceedings of the 6th International Parallel Processing Symposium* (Beverly Hills, CA, USA, March 1992), pp. 216–230.

[18] Jacobson, V. Avoid read-side locking via delayed free. private communication, September 1993.

[19] John, A. Dynamic vnodes – design and implementation. In *USENIX Winter 1995* (New Orleans, LA, January 1995), USENIX Association, pp. 11–23.

[20] 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].

[21] Liskov, B. Distributed programming in Argus. *Commun. ACM 31*, 3 (1988), 300–312.

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

[23] McKenney, P. E. Stochastic fairness queuing. In *IEEE INFOCOM'90 Proceedings* (San Francisco, June 1990), The Institute of Electrical and Electronics Engineers, Inc., pp. 733–740. Available: http://www.rdrop.com/users/paulmck/scalability/paper/sfq.2002.06.04.pdf [Viewed May 26, 2008].

[24] McKenney, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf [Viewed October 15, 2004].

[25] McKenney, P. E. RCU vs. locking performance on different CPUs. In *linux.conf.au* (Adelaide, Australia, January 2004). Available: http://www.linux.org.au/conf/2004/abstracts.html#90 http://www.rdrop.com/users/paulmck/RCU/lockperf.2004.01.17a.pdf [Viewed June 23, 2004].

[26] McKenney, P. E. Memory ordering in modern microprocessors, part I. *Linux Journal 1*, 136 (August 2005), 52–57. Available: http://www.linuxjournal.com/article/8211 http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.19a.pdf [Viewed November 30, 2007].

[27] McKenney, P. E. RCU Linux usage. Available: http://www.rdrop.com/users/paulmck/RCU/linuxusage.html [Viewed January 14, 2007], October 2006.

[28] McKenney, P. E. Sleepable RCU. Available: http://lwn.net/Articles/202847/ Revised: http://www.rdrop.com/users/paulmck/RCU/srcu.

2007.01.14a.pdf [Viewed August 21, 2006], October 2006.

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

[30] McKenney, P. E. [PATCH] QRCU with lockless fastpath. Available: http://lkml.org/lkml/2007/2/25/18 [Viewed March 27, 2008], February 2007.

[31] 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.

[32] McKenney, P. E. RCU and unloadable modules. Available: http://lwn.net/Articles/217484/ [Viewed November 22, 2007], January 2007.

[33] McKenney, P. E. Using Promela and Spin to verify parallel algorithms. Available: http://lwn.net/Articles/243851/ [Viewed September 8, 2007], August 2007.

[34] McKenney, P. E. RCU part 3: the RCU API. Available: http://lwn.net/Articles/264090/ [Viewed January 10, 2008], January 2008.

[35] McKenney, P. E. What is RCU? part 2: Usage. Available: http://lwn.net/Articles/263130/ [Viewed January 4, 2008], January 2008.

[36] McKenney, P. E., Appavoo, J., Kleen, A., Krieger, O., Russell, R., Sarma, D., and Soni, M. Read-copy update. In *Ottawa Linux Symposium* (July 2001). Available: http://www.linuxsymposium.org/2001/abstracts/readcopy.php http://www.rdrop.com/users/paulmck/RCU/rclock_OLS.2001.05.01c.pdf [Viewed June 23, 2004].

[37] McKenney, P. E., and Sarma, D. Read-copy update mutual exclusion in Linux. Available: http://lse.sourceforge.net/locking/rcu/rcupdate_doc.html [Viewed October 18, 2004], February 2001.

[38] 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].

[39] 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].

[40] 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].

[41] 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].

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

[43] McKenney, P. E., and Walpole, J. Introducing technology into the Linux kernel: a case study. *SIGOPS Oper. Syst. Rev. 42*, 5 (2008), 4–17.

[44] Meuer, H., Dongarra, J., Strohmaier, E., and Simon, H. Top 500 supercomputer sites: Operating system family share. Available: http://www.top500.org/stats/list/30/osfam [Viewed March 25, 2008], November 2007.

[45] Michael, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems 15*, 6 (June 2004), 491–504.

[46] Minyard, C., and McKenney, P. E. [PATCH] add an RCU version of list splicing. Available: http://lkml.org/lkml/2007/1/3/112 [Viewed May 28, 2007], January 2007.

[47] Molnar, I., and Miller, D. S. brlock. Available: http://www.tm.kernel.org/pub/linux/kernel/v2.3/patch-html/patch-2.3.49/linux_include_linux_brlock.h.html [Viewed September 3, 2004], March 2000.

[48] Morris, J. Recent developments in SELinux kernel performance. Available: http://www.livejournal.com/users/james_morris/2153.html [Viewed December 10, 2004], December 2004.

[49] Nesterov, O. Re: [patch] cpufreq: mark cpufreq_tsc() as core_initcall_sync. Available: http://lkml.org/lkml/2006/11/19/69 [Viewed May 28, 2007], November 2006.

[50] Olsson, R., and Nilsson, S. TRASH: A dynamic LC-trie and hash data structure. Available: http://www.nada.kth.se/~snilsson/public/papers/trash/trash.pdf [Viewed February 24, 2007], August 2006.

[51] Piggin, N. [patch 3/3] radix-tree: RCU lockless readside. Available: http://lkml.org/lkml/2006/6/20/238 [Viewed March 25, 2008], June 2006.

[52] Pugh, W. Concurrent maintenance of skip lists. Tech. Rep. CS-TR-2222.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, College Park, Maryland, June 1990.

[53] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *2nd Symposium on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, CA, October 1987), Association for Computing Machinery, pp. 31–39. Available:

http://citeseer.csail.mit.edu/cache/papers/cs/6535/http:zSzzSzwww.cs.cornell.eduzSzcs614-sp98zSzberkeley-262zSzmach-vm.pdf/rashid87machineindependent.pdf [Viewed February 17, 2005].

[54] Rossbach, C. J., Hofmann, O. S., Porter, D. E., Ramadan, H. E., Bhandari, A., and Witchel, E. TxLinux: Using and managing hardware transactional memory in an operating system. In *SOSP'07: Twenty-First ACM Symposium on Operating Systems Principles* (October 2007), ACM SIGOPS. Available: http://www.sosp2007.org/papers/sosp056-rossbach.pdf [Viewed October 21, 2007].

[55] Rostedt, S., and McKenney, P. E. [PATCH] add support for dynamic ticks and preempt rcu. Available: http://lkml.org/lkml/2008/1/29/208 [Viewed March 27, 2008], January 2008.

[56] Russell, R. Re: modular net drivers. Available: http://oss.sgi.com/projects/netdev/archive/2000-06/msg00250.html [Viewed April 10, 2006], June 2000.

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

[58] Seigh, J. Lock-free synchronization primitives. Available: http://sourceforge.net/projects/atomic-ptr-plus/ [Viewed August 8, 2005], July 2005.

[59] Spraul, M. Re: RFC: patch to allow lock-free traversal of lists with insertion. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=100264675012867&w=2 [Viewed June 23, 2004], October 2001.

[60] Spraul, M. [rfc] 0/5 rcu lock update. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=108546407726602&w=2 [Viewed June 23, 2004], May 2004.

[61] Torvalds, L. Linux 2.5.43. Available: http://lkml.org/lkml/2002/10/15/425 [Viewed March 30, 2008], October 2002.

[62] Torvalds, L. Linux 2.5.69. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=105209603501299&w=2 [Viewed June 23, 2004], May 2003.

[63] Torvalds, L. Linux 2.5.70. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=105400162802746&w=2 [Viewed June 23, 2004], May 2003.

[64] Transaction Processing Performance Council. TPC benchmark A. Available: http://www.tpc.org/tpca/default.asp [Viewed July 6, 2007], November 1989.

[65] Zijlstra, P., and Molnar, I. [PATCH 3/7] barrier: a scalable synchonisation barrier. Available: http://lkml.org/lkml/2007/1/28/34 [Viewed March 27, 2008], January 2007.