RCU Usage In the Linux Kernel: One Decade Later

Paul E. McKenney
Linux Technology Center
IBM Beaverton

Silas Boyd-Wickizer
MIT CSAIL

Jonathan Walpole
Computer Science Department
Portland State University

Abstract

Read-copy update (RCU) has been used in the Linux kernel for more than a decade, raising the question of exactly what it is used for. To answer this question, we briefly survey use of RCU in the Linux kernel, addressing the why, where, and how of its usage. This document also includes a novel graphical depiction of the relationships among several patterns of RCU usage.

1 Introduction

This section presents the fundamental properties of RCU, which operates by maintaining multiple versions of objects. Readers access a given version, and updaters must leave that version intact until readers have released all references to it. Updaters must therefore receive some indication of when readers' accesses begin and end, which is accomplished using *RCU read-side critical sections* demarked by `rcu_read_lock()` and `rcu_read_unlock()`. Any time a given thread is not in an RCU read-side critical section, it is in a *quiescent state*.

Readers traverse all pointers to and within an RCU-protected data structure using `rcu_dereference()`, with each traversal contained within a single RCU read-side critical section. Updaters may also traverse the structure under any desired synchronization mechanism, including non-blocking synchronization, single updater thread, and transactional memory [12]. In the Linux kernel, RCU updaters normally use locking.

Any time period during which each thread[1] has been observed in at least one quiescent state is a *grace period*. Suppose an updater renders a given RCU-protected data item inaccessible to readers and then waits for one grace period. By definition, there can no longer be any RCU readers referencing that item. That item's memory can thus be safely reclaimed, for example, by freeing it.[2]

The `synchronize_rcu()` primitive waits for a grace period to elapse, and its asynchronous counterpart, `call_rcu()`, causes a specified function to be invoked after a subsequent grace period.

On weakly ordered systems, a reader accessing a data item that was concurrently initialized and then inserted into the structure could observe that item's pre-initialized value. As a result, new data items must be made accessible to readers using a `rcu_assign_pointer()` primitive. This primitive contains the instructions and directives required to prevent both the compiler and the CPU from reordering references. As noted earlier, readers must use `rcu_dereference()`, which also controls ordering as needed. Both primitives reduce to simple assignment statements on sequentially consistent systems. The `rcu_dereference()` primitive is a volatile access[3] on all systems other than DEC Alpha, where a memory barrier instruction is required [5].

Sections 2, 3, and 4 give the why, where, and how of the Linux kernel's RCU usage, respectively. Section 5 presents algorithmic transformations that increase RCU's applicability. Finally, Section 6 presents concluding remarks.

2 RCU Usage: Why?

This section gives a qualitative view of why RCU is used in the Linux kernel. Some quantitative data is given in Sections 3, 4, and elsewhere [7, 11]. This section augments the fundamental conceptual properties of RCU called out in Section 1 with practical requirements from the Linux kernel. These requirements result in some degree of specialization: Other environments might have

[1] Within the Linux kernel, the closest analog to a thread is a *task*.

[2] Waiting for a grace period to elapse before reclaiming memory implies that RCU-protected data structures are immune from the ABA problem [27]. In addition, placing non-blocking synchronization algorithms in RCU read-side critical sections permits the same simplifications permitted by the use of garbage collectors.

[3] Compilers implementing the C++11 standard may use a volatile `memory_order_consume` load.

different requirements met by a different RCU implementation, but nevertheless implementing the same fundamental abstraction.

RCU's read-side primitives are typically wait free and are exceedingly fast, with `rcu_read_lock()` and `rcu_read_unlock()` having exactly zero overhead in server-class kernel builds with `CONFIG_PREEMPT=n`. The wait-free nature of RCU's read-side primitives, combined with the fact that these primitives are required to succeed unconditionally, implies that RCU readers and updaters make forward progress concurrently. In this case, there are none of the spinning, blocking, rollbacks, or retries that can otherwise bedevil attempts to attain good performance, scalability, and real-time response on large systems.

In addition, real-time builds of the Linux kernel require that RCU read-side critical sections be preemptible. Whether real time or not, RCU read-side primitives must be usable from all non-idle environments within the kernel, including RCU read-side critical sections (`rcu_read_lock()` may be nested freely), as well as interrupt and non-maskable interrupt (NMI) handlers. The RCU implementation may not make any assumptions about the size and extent of RCU-protected data structures. This restriction permits use of any memory-allocation mechanism, including compile-time allocation. However, RCU implementations are permitted to assume that all RCU read-side critical sections are finite.

RCU is intended for read-mostly situations, so implementations may use expensive grace-period mechanisms with long latencies (e.g., milliseconds). However, Linux-kernel RCU implementations must use batching so as to satisfy multiple updates with one RCU grace period[4] and should also offer "expedited" grace-period primitives such as `synchronize_rcu_expedited()` [16].

The Linux kernel's RCU implmentation must tolerate CPU-hotplug operations and must avoid awakening CPUs that are in low-power states [21]. RCU implementations should issue warnings for excessively long RCU read-side critical sections. RCU must operate efficiency on systems with several thousand CPUs, and must not prevent real-time response in the low tens of microseconds on such systems. Finally, in real-time builds of the Linux kernel, RCU must prevent indefinite preemption within RCU read-side critical sections.

In short, while the fundamental conceptual properties of RCU allow a large number of implementations, some of which are quite simple [22], production-quality RCU implementations must meet a number of additional constraints in order to be useful within the Linux kernel.

The next section considers the extent of RCU usage in the Linux kernel.

[4] In the Linux kernel, it is not unusual for a single grace period to satisfy more than 1,000 updates [26].

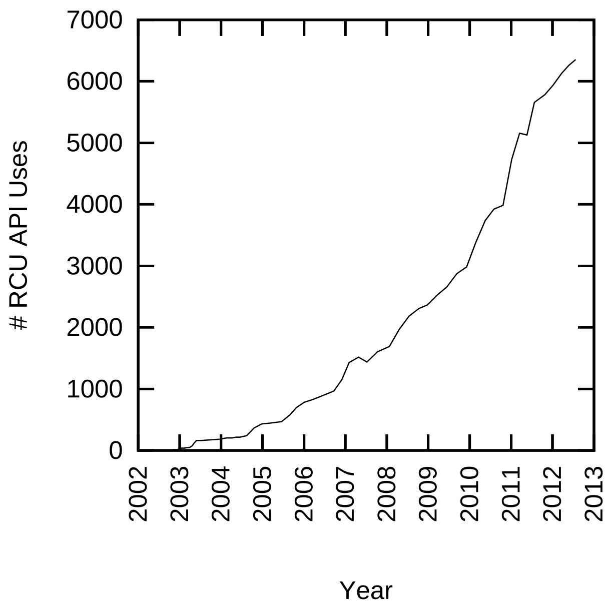

Figure 1: Linux-Kernel RCU Usage

3 RCU Usage: How Much and Where?

This section examines the usage of RCU in the Linux kernel over time, by subsystem within the kernel, and by type of RCU primitive. Figure 1 shows how the usage of RCU in the Linux kernel has increased over time. Although this increase has been quite large, it should be noted that there are more than ten times as many uses of locking as there are of RCU. This should not be too surprising, as RCU (1) is comparatively new to the Linux kernel, (2) is a specialized primitive, and (3) typically uses locking to mediate RCU updates.

Subsystem	Uses	LoC	Uses / KLoC
virt	65	6,400	10.16
ipc	35	8,116	4.31
net	3086	717,501	4.30
security	245	66,990	3.66
kernel	620	187,863	3.30
block	65	28,053	2.32
mm	186	86,486	2.15
lib	66	51,709	1.28
init	2	3,308	0.60
fs	595	1,014,373	0.59
include	266	512,880	0.52
crypto	12	56,913	0.21
drivers	859	8,059,951	0.11
arch	156	2,394,340	0.07
Total	6258	13,194,883	0.47

Table 1: Linux 3.4 RCU Usage by Subsystem

Type of Usage	API Usage
Annotation of RCU-protected pointers	250
Initialization and cleanup	256
Markers for RCU read-side critical sections	2920
RCU lockdep assertion	25
RCU pointer traversal	847
RCU pointer update	316
RCU list traversal	541
RCU list update	495
RCU grace period	608
Total	6258

Table 2: Linux 3.4 RCU Usage by Function

That said, Table 1 shows that RCU usage (excluding definitions) is spread widely across the Linux kernel. These counts also exclude indirect uses of RCU via wrapper functions or macros. Lines of code are computed by a simple count of text lines in the .c and .h files in the Linux source tree.

Linux's networking stack contains almost half of the most uses of RCU, despite networking comprising only about 5% of the kernel. Networking is well-suited to RCU due to its large proportion of read-mostly data describing network hardware and software configuration. Interestingly enough, the first uses of DYNIX/ptx's RCU equivalent [25] were also in networking.

However, virtualization (KVM) uses RCU most intensively, with fully one percent of its lines of code invoking RCU, 2.5 times that of the networking stack. Linux's drivers contain the second-greatest number of uses of RCU, but also have the second-lowest intensity. In fact, it has only been in the past few years that the drivers have made much use of RCU at all.

In short, RCU pervades the Linux kernel, with about one of every 2,000 lines of code being an RCU primitive. Within the individual subsystems, RCU usage ranges from about one of every 15,000 lines of code (architecture support) to about one out of every 100 lines of code (virtualization).

Table 2 lists the number of uses of RCU API members grouped into types of primitives having similar functionality.[5]. The `__rcu` tag is used to annotate RCU-protected pointers so that Linux's "sparse" [6] static-analysis tool can flag traversals of RCU-protected pointers that are not properly protected either by an RCU read-side critical section or an update-side lock. Markers for RCU read-side critical sections include the `rcu_read_lock()` and `rcu_read_unlock()` primitives introduced in Section 1. RCU lockdep assertions emit a run-time error if they are executed outside of an RCU read-side critical section. RCU pointer traversal includes `rcu_dereference()`, while RCU pointer update includes `rcu_assign_pointer()`, both introduced in Section 1. RCU list-traversal and list-update primitives operate on RCU-protected linked lists. Finally, RCU grace period primitives include the `synchronize_rcu()` and `call_rcu()` primitives called out in Section 1.

There are 1,388 (847+541) occurrences of primitives that traverse RCU-protected pointers (either independently or as part of a list iterator, respectively), which would correspond to 2776 RCU read-side critical sections assuming one traversal per critical section. However, there are instead 2920 markers for RCU read-side critical sections, which is more than 100 more than expected. There are two major reasons for this discrepancy: (1) Some RCU read-side critical sections have multiple `rcu_read_unlock()` primitives, and (2) RCU traversal primitives are often encapsulated in functions and macros not counted in Table 2, and these are invoked from multiple RCU read-side critical sections.

The higher-level list primitives are used frequently, in fact, 47% of the traversals and updates use them. Software-engineering assistance is also quite popular: Of the 847 pointer traversals, 777 (92%) are verified by the lockdep-RCU [17] dynamic-validation facility. However, of the 541 list traversals, only 6 are verified by lockdep-RCU (1%). This was a design choice: Verifying these traversals would add 42 additional RCU API members. There is some indication that adding verification would be worthwhile, however, given adding verification to the pointer traversals resulted in about a year of annotation pain, movement in this direction will be quite deliberate. Of the 495 list updates, 240 are adds, 240 are deletes, 13 are replacements, and 2 are splices (where an entire list is inserted into another list). The replacements usually replace a single element in the list with an updated copy: RCU updates typically make small changes to larger data structures.

This section has looked at the Linux kernel's use of RCU from a statistical viewpoint, and the next section looks at the higher-level semantics of RCU usage.

4 RCU Usage: How?

This section looks at uses of RCU in relation to the existing synchronization mechanisms that RCU replaces, as depicted in Figure 2. Each usage in the figure other than publish-subscribe[6] is described in one of the following sections, starting with those at the bottom of the figure.

[5] Full explanation of each member of the RCU API is beyond the scope of this document. More detail may be found elsewhere [18].

[6] Publish-subscribe, which is required for non-sequentially-consistent systems (in other words, for all commercially available systems), was discussed in Section 1.

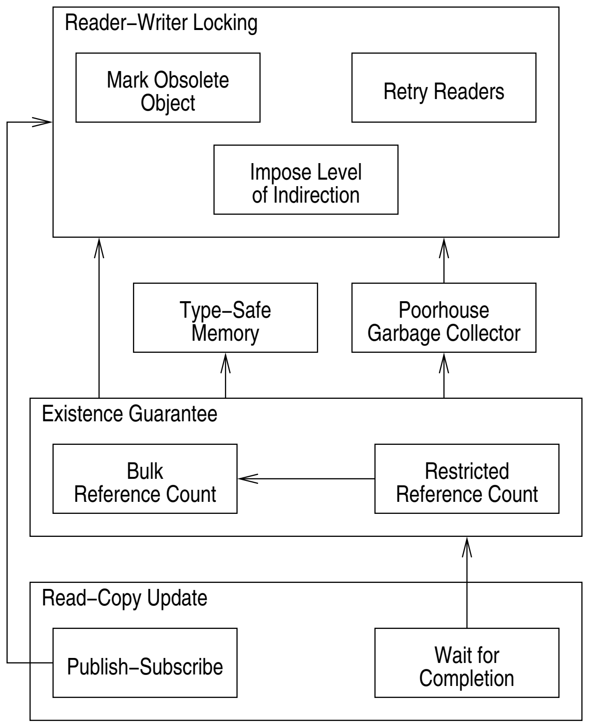

Figure 2: RCU Usage Relationships

4.1 Wait for Completion

Wait for completion uses the grace-period operations that wait for pre-existing RCU readers. This approach allows waiting for each of thousands of different things to complete without having to explicitly track each and every one of them, and without having to worry about the performance degradation, scalability limitations, complex deadlock scenarios, and memory-leak hazards that are inherent to explicit tracking.

In non-preemptive builds of the Linux kernel, anything that disables preemption, along with hardware operations and primitives that disable interrupts, serves as an RCU read-side critical section.[7] Therefore, RCU can interact with non-maskable interrupt (NMI) handlers, which is quite difficult with non-blocking synchronization and in general infeasible with locking. This approach has been called "Pure RCU" [14], and it is used in a number of places in the Linux kernel, typically as follows:

1. Make a change, for example, to the way that the OS reacts to an NMI.

2. Wait for all pre-existing read-side critical sections to completely complete by using the `synchronize_sched()` primitive. Subsequent RCU read-side critical sections are guaranteed to the change.

3. Clean up, for example, return status indicating that the change was successful.

[7] In preemptive builds, `rcu_read_lock_sched()`, `rcu_read_unlock_sched()`, `synchronize_sched()`, and `call_rcu_sched()` retain non-preemptive semantics.

More details, including code, may be found elsewhere [14, Section 6.3].

Although interacting with NMI handlers is perhaps the most unfamiliar use of wait for completion, it does have a number of other uses. For example, it has been used outside of the Linux kernel to sequence through updates to complex data structures while avoiding disrupting readers [12, 28]. It also forms the basis for higher-level uses of RCU, as will be seen in later sections.

4.2 Restricted Reference Count

Because grace periods are not allowed to complete while there is an RCU read-side critical section in progress, the RCU read-side primitives may be used as a restricted reference-counting mechanism for a given data item. Waiting for completion then waits for all pre-existing readers, thus in turn for all RCU-based reference counts for that item. For example, consider the following code fragment:

```
rcu_read_lock();  /* acquire reference. */
p = rcu_dereference(head);
/* do something with p. */
rcu_read_unlock();  /* release reference. */
```

The `rcu_read_lock()` primitive can be thought of as acquiring a reference to `p`, because a grace period starting after the `rcu_dereference()` assigns to `p` cannot possibly end until after we reach the matching `rcu_read_unlock()`. Therefore, the following code can safely delete the data item referenced by `p`:

```
spin_lock(&mylock);
p = head;
rcu_assign_pointer(head, NULL);
spin_unlock(&mylock);
/* Wait for all references to be released. */
synchronize_rcu();
kfree(p);
```

The assignment to `head` prevents readers from acquiring future references to `p`, and the `synchronize_rcu()` waits for all prior references to be released.

But why bother substituting RCU for reference counting? Part of the answer is performance, as shown in Figure 3, which shows data taken on a 16-CPU 3GHz Intel x86 system. The error bars for both traces span a single standard deviation in either direction.

The performance advantages of RCU are most pronounced for short critical sections, as shown Figure 4. Note however that each 3GHz CPU on this system can execute many thousands of instructions in the time required by a single reference-count acquisition-release

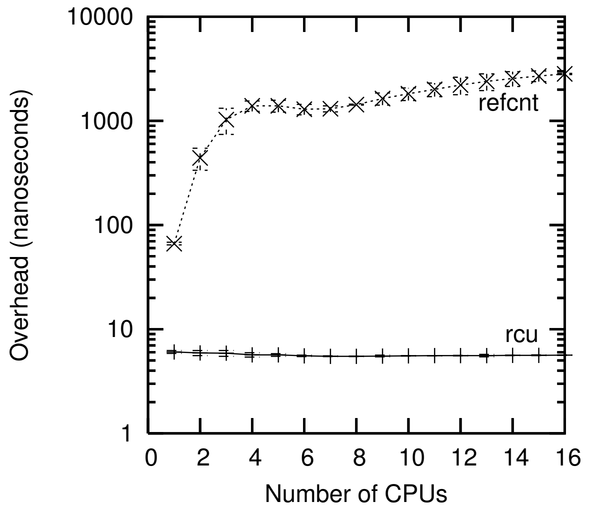

Figure 3: Performance of RCU vs. Reference Counting

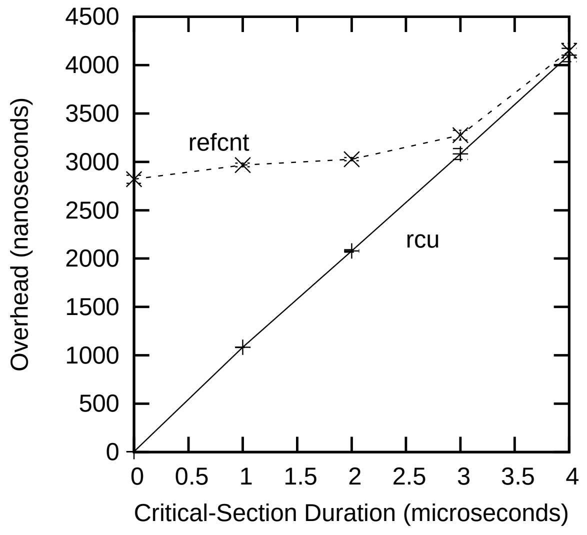

Figure 4: Latency of RCU vs. Reference Counting

pair. Furthermore, many system calls (and thus any RCU read-side critical sections that they contain) complete in a few microseconds.

However, the restriction against sleeping in an RCU read-side critical section can pose a challenge in some cases: If some of the code paths protected by reference counting can sleep, then RCU cannot be directly substited for reference counting. Linux's networking stack addresses this challenge by applying both reference counting and RCU to a given data item, with RCU sufficing for short critical sections that are guaranteed not to block, and with the heavier-weight reference count used for longer critical section, for example, those that include a transmit and/or receive operation. In such use cases, the protected data item cannot be freed until both: (1) A grace period has elapsed since the data item has been rendered inaccessible to readers and (2) the reference count decreases to zero [20].

However, RCU-based reference counters have the countervailing advantage of being cheaply extendable to cover multiple data items, as described in the following section.

4.3 Bulk Reference Count

As noted in the preceding section, traditional reference counters are usually associated with a specific data structure. It is of course possible to use a single global reference counter to protect larger data structures, but frequent references that counter thrashes the cache line containing the reference count. Such thrashing severely degrades performance, especially on large systems.

In contrast, RCU's light-weight read-side primitives permit RCU to be used as a bulk reference-counting mechanism with little or no performance penalty. In this bulk reference-counting approach, executing `rcu_read_lock()` acquires a reference on each and every RCU-protected data item in the structure.

Taking the concept of bulk reference counting to its logical extreme results in *existence guarantees*, which are the subject of the next section.

4.4 Existence Guarantees

Gamsa et al. [8, Section 5] describe how a mechanism resembling RCU can be used to provide these existence guarantees. The effect is that if any RCU-protected data element is accessed within an RCU read-side critical section, that data element is guaranteed to remain in existence for the duration of that critical section.

Figure 5 demonstrates RCU-based existence guarantees enabling per-element locking via a function that deletes an element from a hash table. Line 6 computes a hash function, and line 7 enters an RCU read-side critical section. If line 9 finds that the corresponding bucket of the hash table is empty or that the element present is not the one we wish to delete, then line 10 exits the RCU read-side critical section and line 11 indicates failure.[8]

Otherwise, line 13 acquires the update-side spinlock, and line 14 then checks that the element is still the one that we want. If so, line 15 leaves the RCU read-side critical section, line 16 removes it from the table, line 17 releases the lock, line 18 waits for all pre-existing RCU readers to complete, line 19 frees the newly removed el-

[8] Note that this is a very simple hash table with no chaining, so there is at most one element in a given bucket. Therefore, the element to be deleted will always be the first one in the list.

```
 1 int delete(int key)
 2 {
 3   struct element *p;
 4   int b;
 5
 6   b = hashfunction(key);
 7   rcu_read_lock();
 8   p = rcu_dereference(hashtable[b]);
 9   if (p == NULL || p->key != key) {
10     rcu_read_unlock();
11     return 0;
12   }
13   spin_lock(&p->lock);
14   if (hashtable[b] == p && p->key == key) {
15     rcu_read_unlock();
16     hashtable[b] = NULL;
17     spin_unlock(&p->lock);
18     synchronize_rcu();
19     kfree(p);
20     return 1;
21   }
22   spin_unlock(&p->lock);
23   rcu_read_unlock();
24   return 0;
25 }
```

Figure 5: Existence Guarantees for Per-Element Locking

ement, and line 20 indicates success. If the element is no longer the one we want, line 22 releases the lock, line 23 leaves the RCU read-side critical section, and line 24 indicates failure to delete the specified key.

Alert readers will recognize this as only a slight variation on the original "RCU is a way of waiting for things to finish" theme, which is addressed in Section 4.1.

RCU is used heavily within the Linux kernel for its existence guarantees, most famously by the System-V IPC implementation [1]. Existence guarantees can also be exploited to create type-safe memory, as described in the following section.

4.5 Type-Safe Memory

A number of lockless algorithms do not require that a given data element keep the same identity through a given RCU read-side critical section referencing it—but only if that data element retains the same type. In other words, these lockless algorithms can tolerate a given data element being freed and reallocated as the same type of structure while they are referencing it, but must prohibit a change in type. This guarantee, called "type-safe memory" in academic literature [9], Type-safe memory algorithms in the Linux kernel make use of slab caches, marking type-safe caches with `SLAB_DESTROY_BY_RCU` so that RCU is used when returning a freed-up slab to system memory. This use of RCU guarantees that any in-use element of such a slab will remain in that slab, thus retaining its type, for the duration of any pre-existing RCU read-side critical sections.

These algorithms typically use a validation step that checks to make sure that the newly referenced data structure really is the one that was requested [13, Section 2.5]. These validation checks require that portions of the data structure remain untouched by the free-reallocate process. Such validation checks are usually very hard to get right, and can hide subtle and difficult bugs.

Therefore, although type-safety-based lockless algorithms can be extremely helpful in a some difficult situations, existence guarantees should instead be used where feasible. That said, there are 12 places in the 3.4 Linux kernel that use `SLAB_DESTROY_BY_RCU` to guarantee type-safe memory, including the signal handling data structures, the virtual-memory system's reverse-mapping data structures, and networking.

4.6 Poorhouse Garbage Collector

Existence guarantees can be thought of as analogous to garbage collection (GC), but this line of thought can be misleading. Perhaps the best way to think of the relationship between RCU and automatic garbage collectors (GCs) is that RCU resembles a GC in that the *timing* of collection is automatically determined, but that RCU differs from a GC in that: (1) the programmer must manually indicate when a given data structure is eligible to be collected, and (2) the programmer must manually mark the RCU read-side critical sections where references might legitimately be held.

Despite these differences, the resemblance does go quite deep, and has appeared in at least one theoretical analysis of RCU. Furthermore, the first RCU-like mechanism I am aware of used a garbage collector to handle the grace periods. However, the Linux kernel community seems to find it easier to think of RCU in other terms, with the most popular approach being as a replacement for reader-writer locking, as discussed in the next section.

4.7 Reader-Writer Lock Replacement

Interestingly enough, wait for completion can also be used to replace some uses of reader-writer locking, and this is in fact the most common use of RCU within the Linux kernel. In fact, we will show that in some cases, it is possible to mechanically substitute RCU API members for the corresponding reader-writer lock API members. But first, why bother?

RCU's advantages include performance, deadlock immunity, and realtime latency. RCU's limitations include concurrent readers and updaters, low-priority RCU readers blocking high-priority threads waiting for a grace period to elapse, and grace periods that can extend for many milliseconds. These advantages and limitations are discussed in the following sections.

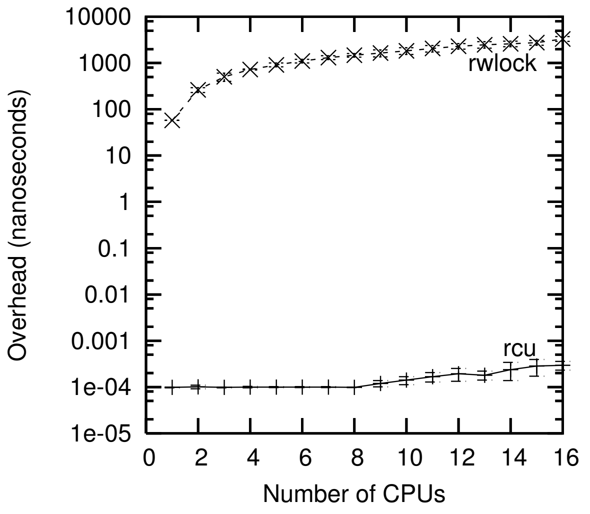

Figure 6: QSBR RCU vs. Reader-Writer Locking

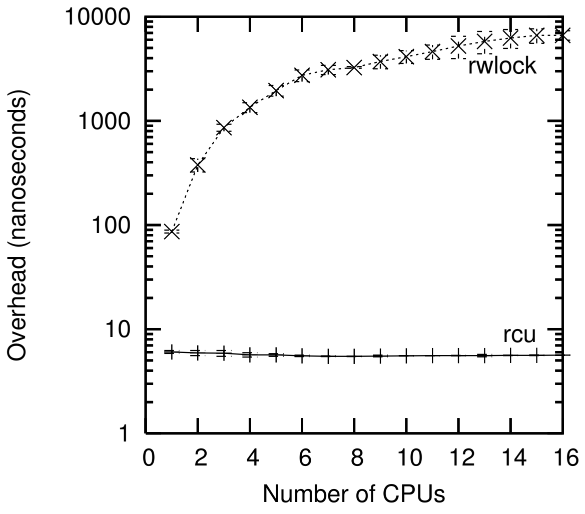

Figure 7: Preemptible RCU vs. Reader-Writer Locking

4.7.1 Performance

The read-side performance advantages of the quiescent-state-based-reclamation (QSBR) form of RCU [11] over reader-writer locking are shown in Figure 6,[9] again for a 16-CPU 3GHz x86 system, and again with error bars for both traces spaning a single standard deviation in either direction. Note that reader-writer locking is orders of magnitude slower than RCU on a single CPU, and is almost two *additional* orders of magnitude slower on 16 CPUs. In contrast, RCU scales quite well.

Although the QSBR form of RCU is extremely effective for throughput-oriented workloads, it can result in excessive scheduling latencies for aggressive real-time workloads, for which preemptible RCU was designed. Although preemptible RCU incurs some read-side overhead, it beats reader-writer locking by between one and three orders of magnitude, as shown in Figure 7.

The performance advantages of RCU become less significant as the overhead of the critical section increases, as shown in Figure 8 for a 16-CPU system, in which the y-axis represents the sum of the overhead of the read-side primitives and that of the critical section. The variance decreases as the critical-section duration increases due to decreasing levels of contention on the data structure implementing the reader-writer lock. However, note that many system calls (and thus any critical sections they contain) complete in well under a microsecond.

In addition to RCU's performance advantages, as discussed in the next section, RCU read-side primitives are almost entirely deadlock-immune.

[9] The measured value of 100 femtoseconds is an overestimate due to timing overhead. The true value is exactly zero.

4.7.2 Deadlock Immunity

Because RCU's read-side primitives are typically wait-free, they are also typically immune to deadlock.[10] In addition, RCU's deadlock immunity can greatly simplify design by removing the need for complex deadlock-avoidance code. In fact, a major early benefit of RCU in Sequent's DYNIX/ptx kernel was the removal of more than 10,000 lines of complex and difficult-to-test deadlock-avoidance code during a conversion of locking to RCU.

As a result of RCU's deadlock immunity, it is possible to unconditionally upgrade an RCU reader to an RCU updater, for example as follows:

```
rcu_read_lock();
list_for_each_entry_rcu(p, &head, list_field) {
  do_something_with(p);
  if (need_update(p)) {
    spin_lock(my_lock);
    do_update(p);
    spin_unlock(&my_lock);
  }
}
rcu_read_unlock();
```

In contrast, an analogous reader-writer-locking upgrade locking would deadlock. The wait-free nature of RCU's read-side primitives also benefits real-time workloads, as discussed in the following section.

4.7.3 Realtime Latency

RCU read-side primitives also offer excellent realtime latencies. In addition, as noted earlier, they are immune to

[10] One remaining deadlock scenario involves (illegally) placing `synchronize_rcu()` inside an RCU read-side critical section, while others involve the Linux scheduler's locks [19].

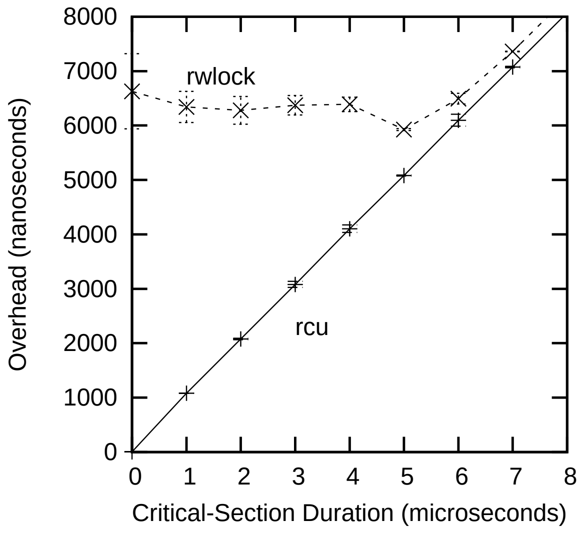

Figure 8: Comparison of RCU to Reader-Writer Locking as Function of Critical-Section Duration (16 CPUs)

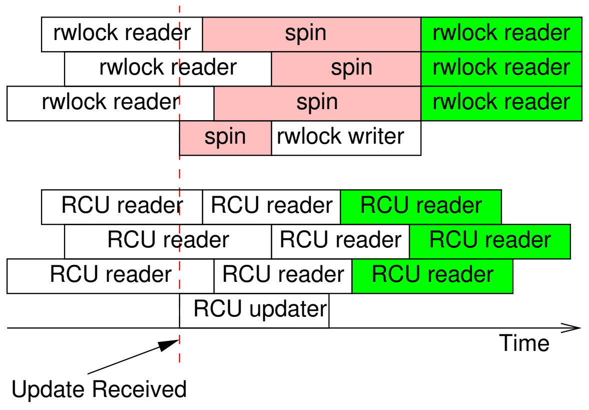

Figure 9: Latency of RCU vs. Reader-Writer Locking

priority inversion. For example, low-priority RCU readers cannot prevent a high-priority RCU updater from acquiring the update-side lock. Similarly, a low-priority RCU updater cannot prevent high-priority RCU readers from entering an RCU read-side critical section. Therefore, RCU can often provide reduced latency with respect to outside events, as shown by Figure 9 [25].

However, RCU is susceptible to more subtle priority-inversion scenarios, for example, a high-priority process blocked waiting for an RCU grace period to elapse can be blocked by low-priority RCU readers in -rt kernels. This can be solved by using RCU priority boosting [10, 15].

4.7.4 Concurrent RCU Readers and Updaters

RCU's performance advantages stem largely from the fact that readers and updaters can run concurrently, but this concurrency can be an obstacle because it means that RCU readers and updaters cannot in general be strictly serializable. This lack of strict serializability means that RCU readers might access stale data, and might even see inconsistencies, either of which can render conversion from reader-writer locking to RCU non-trivial. However, this design choice is inherent: Strictly serializable synchronization mechanisms cannot have all of: (1) concurrent readers and updaters, (2) readers that never impede updaters,[11] and (3) wait-free read-only transactions [2].[12] Because these three properties provide RCU's performance, scalability, and real-time response, giving up strict serializability is a good tradeoff.

Furthermore, in a surprisingly large number of situations, lack of strict serializability is a non-problem. The classic example is the networking routing table. Because routing updates can take considerable time to reach a given system (seconds or even minutes), the system will have been sending packets the wrong way for quite some time when the update arrives. It is usually not a problem to continue sending updates the wrong way for a few additional milliseconds. Furthermore, as noted earlier, RCU can offer reduce response times to outside events, such as routing changes.

Nevertheless, there are situations where inconsistency and stale data within the confines of the system cannot be tolerated. Section 5 discusses a number of transformations that allow algorithms to tolerate inconsistency and stale data.

4.7.5 Reader-Writer Locking vs. RCU: Code

In the best case, the conversion from reader-writer locking to RCU is quite simple, as shown in Figures 10, 11, and 12, all taken from Wikipedia [23]. More-complex transformations are covered in Section 5.

5 Algorithmic Transformations

RCU replacements for reader-writer locking permit readers and updaters to run concurrently, which poses problems when attempting to convert some reader-writer-locked algorithms to RCU. The following sections describe some ways or transforming such algorithms into forms that can be more readily converted to RCU. More details may be found elsewhere [14].

[11] Note that while RCU readers can impede reclamation, they do not typically impede the actual update itself.

[12] Attiya et al. focus on software transactional memory (STM), but locked updates with RCU readers maps to STM with invisible readers.

```
struct el {
  struct list_head lp;
  long key;
  spinlock_t mutex;
  int data;
  /* Other data fields */
};
DEFINE_RWLOCK(listmutex);
LIST_HEAD(head);
```

```
struct el {
  struct list_head lp;
  long key;
  spinlock_t mutex;
  int data;
  /* Other data fields */
};
DEFINE_SPINLOCK(listmutex);
LIST_HEAD(head);
```

Figure 10: Converting Reader-Writer Locking to RCU: Data

```
int search(long key, int *result)
{
  struct el *p;

  read_lock(&listmutex);
  list_for_each_entry(p, &head, lp) {
    if (p->key == key) {
      *result = p->data;
      read_unlock(&listmutex);
      return 1;
    }
  }
  read_unlock(&listmutex);
  return 0;
}
```

```
int search(long key, int *result)
{
  struct el *p;

  rcu_read_lock();
  list_for_each_entry_rcu(p, &head, lp) {
    if (p->key == key) {
      *result = p->data;
      rcu_read_unlock();
      return 1;
    }
  }
  rcu_read_unlock();
  return 0;
}
```

Figure 11: Converting Reader-Writer Locking to RCU: Search

```
int delete(long key)
{
  struct el *p;

  write_lock(&listmutex);
  list_for_each_entry(p, &head, lp) {
    if (p->key == key) {
      list_del(&p->lp);
      write_unlock(&listmutex);

      kfree(p);
      return 1;
    }
  }
  write_unlock(&listmutex);
  return 0;
}
```

```
int delete(long key)
{
  struct el *p;

  spin_lock(&listmutex);
  list_for_each_entry(p, &head, lp) {
    if (p->key == key) {
      list_del_rcu(&p->lp);
      spin_unlock(&listmutex);
      synchronize_rcu();
      kfree(p);
      return 1;
    }
  }
  spin_unlock(&listmutex);
  return 0;
}
```

Figure 12: Converting Reader-Writer Locking to RCU: Deletion

5.1 Impose Level of Indirection

Algorithms based on reader-writer locking may rely on atomic-to-readers coordinated changes to independent variables. Because RCU readers cannot exclude updaters, such algorithms cannot be directly converted to use RCU. However, in many cases they can be transformed by imposing a level of indirection, so that readers traverse an RCU-protected pointer in order to reach a structure containing these variables. Updaters can then publish an update update these variables by updating that single pointer, ensuring that each RCU reader sees a consistent view of the variables.

In many cases, the needed indirection is inherent in the linked data structures used in the Linux kernel, including linked lists, hash tables, and various search trees. In many such cases, RCU readers accessing obsolete data can be considered to be ordered immediately before the corresponding update. In addition, the common reader-writer-locking usage pattern where a result is passed out of a read-side critical section also results in use of obsolete data because an update can execute as soon as that critical section exits. This usage pattern is often the easiest to convert to RCU [14].

5.2 Mark Obsolete Object

Although imposing a level of indirection can ensure that each reader individually sees a consistent view of the protected data, in some cases still greater consistency is required. A case in point in the Linux kernel is the mapping of System V semaphore indentifiers to the corresponding in-kernel data structure. Consistency is automatically provided by co-locating all of a given semaphore's data into its data structure, but it is not permissible to allow one process to manipulate a semaphore that another process has just deleted.

The Linux kernel handles this situation by maintaining a `->deleted` flag for each semaphore, setting this flag when the corresponding semaphore is deleted. If a process finds that its semaphore identifier maps to a structure that has the `->deleted` flag set, it acts as if the mapping failed. Setting and checking the `->deleted` flag is protected by the same per-semaphore lock that is used to protect the individual semaphore operations [1]. This approach gives up some degree of concurrency between readers and writers as well as read-side wait-freedom in order to gain the strict serializability required in this situation [2]. However, the strict serializability applies only to operations on specific semaphores; the mapping operation itself need not be serializable with semaphore creation and deletion.

In short, the mark-obsolete-object approach protects the enclosing search structure in a highly performant and scalable manner with RCU, while avoiding stale data through use of locking on the individual data elements reachable via that search structure. In so doing, it introduces just enough serializability for the problem at hand.

```
do {
  seq = read_seqbegin(&update_lock);
  rcu_read_lock();
  read_something();
  rcu_read_unlock();
} while (read_seqretry(&update_lock, seq));
```

Figure 13: Retrying Readers

5.3 Retry Readers

In cases where the RCU read-side critical section is idempotent, one approach is to enclose it in a loop that checks for updates, retrying the critical section until is executes without concurrent updates. In the Linux kernel, this can be accomplished by combining RCU with sequence locking [3], as shown in Figure 13 and as used by the vfs layer in the Linux kernel [24]. In this combination, RCU ensures that traveral of any RCU-protected pointer will arrive at a valid data item, while sequence locking ensures that no updates occurred during the execution of the final RCU read-side critical section. This approach again gives up read-update concurrency and wait-free readers in order to gain only that degree serializability required by the problem at hand [2].

When updates are rare, this approach provides the excellent performance and scalability that has come to be associated with RCU. Of course, this approach is vulnerable to starvation in the face of high update rates, however, such starvation can be prevented by acquisition of the update-side lock after a large number of failed attempts to traverse the RCU read-side critical section.

6 Conclusions

This paper has listed reasons why RCU is used in the Linux kernel (Section 2), taken a census of RCU usage in version 3.4 of the Linux kernel (Section 3), and shown that it is possible to build higher-level constructs on top of RCU. These constructs include reference counting, existence guarantee, type-safe memory, and reader-writer-locking (Section 4). Algorithms are often adapted to RCU using the transformations listed in Section 5. Figure 2 on page 4 presents a novel depiction of the relationships among these constructs and transformations.

It is likely that the Linux community will continue to find new uses for RCU, both separately and in combination with other synchronization primitives.

Acknowledgements

@@@ We are indebted to Jim Wasko for his support of this effort.

Legal Statement

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] ATTIYA, H., HILLEL, E., AND MILANI, A. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2009), SPAA '09, ACM, pp. 69–78.

[3] BROWN, N. Meet the lockers. Available: `http://lwn.net/Articles/453685/` [Viewed September 2, 2011], August 2011.

[4] CLEMENTS, A., KAASHOEK, F., AND ZELDOVICH, N. Scalable address spaces using RCU balanced trees. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)* (London, UK, March 2012), ACM, pp. @@@–@@@.

[5] COMPAQ COMPUTER CORPORATION. Shared memory, threads, interprocess communication. Available: `http://www.openvms.compaq.com/wizard/wiz_2637.html` [Viewed: June 23, 2004], August 2001.

[6] CORBET, J. Finding kernel problems automatically. Linux Weekly News, June 2004.

[7] DESNOYERS, M., MCKENNEY, P. E., STERN, A., DAGENAIS, M. R., AND WALPOLE, J. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems 23* (2012), 375–382.

[8] 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].

[9] GREENWALD, M., AND CHERITON, D. R. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), USENIX Association, pp. 123–136.

[10] 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].

[11] 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.

[12] HOWARD, P. W., AND WALPOLE, J. A relativistic enhancement to software transactional memory. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism* (Berkeley, CA, USA, 2011), HotPar'11, USENIX Association, pp. 1–6.

[13] LANIN, V., AND SHASHA, D. A symmetric concurrent b-tree algorithm. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference* (Los Alamitos, CA, USA, 1986), IEEE Computer Society Press, pp. 380–389.

[14] 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].

[15] 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.

[16] McKenney, P. E. [PATCH -tip 0/3] expedited 'big hammer' RCU grace periods. Available: `http://lkml.org/lkml/2009/6/25/306` [Viewed August 16, 2009], June 2009.

[17] McKenney, P. E. Lockdep-RCU. Available: `https://lwn.net/Articles/371986/` [Viewed June 4, 2010], February 2010.

[18] McKenney, P. E. The RCU API, 2010 edition. Available: `http://lwn.net/Articles/418853/` [Viewed December 8, 2010], December 2010.

[19] McKenney, P. E. 3.0 and RCU: what went wrong. Available: `http://lwn.net/Articles/453002/` [Viewed July 27, 2011], July 2011.

[20] McKenney, P. E. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2012. Available: `http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html` [Viewed March 28, 2010].

[21] McKenney, P. E. Making rcu safe for battery-powered devices. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdynticks.2012.02.15b.pdf` [Viewed March 1, 2012], February 2012.

[22] 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].

[23] McKenney, P. E., Purcell, C., Algae, Schumin, B., Cornelius, G., Qwertyus, Conway, N., Sbw, Blainster, Rufus, C., Zoicon5, Anome, and Eisen, H. Read-copy update. Available: `http://en.wikipedia.org/wiki/Read-copy-update` [Viewed August 21, 2006], July 2006.

[24] McKenney, P. E., Sarma, D., and Soni, M. Scaling dcache with RCU. *Linux Journal 1*, 118 (January 2004), 38–46. Available: `http://www.linuxjournal.com/node/7124` [Viewed December 26, 2010].

[25] 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].

[26] 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. Available: `https://www.usenix.org/conference/2004-usenix-annual-technical-conference/making-rcu-safe-deep-sub-millisecond-response` [Viewed July 26, 2012].

[27] Treiber, R. K. Systems programming: Coping with parallelism. RJ 5118, Available: `http://domino.research.ibm.com/library/cyberdig.nsf/index.html` [Viewed January 23, 2007], April 1986.

[28] Triplett, J., McKenney, P. E., and Walpole, J. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX Annual Technical Conference* (Portland, OR USA, June 2011), The USENIX Association, pp. 145–158.