

Supplementary Material for User-Level Implementations of Read-Copy Update

Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais and Jonathan Walpole

Abstract—Read-copy update (RCU) is a synchronization technique that often replaces reader-writer locking because RCU's read-side primitives are both wait-free and an order of magnitude faster than uncontended locking. Although RCU updates are relatively heavy weight, the importance of read-side performance is increasing as computing systems become more responsive to changes in their environments.

RCU is heavily used in several kernel-level environments. Unfortunately, kernel-level implementations use facilities that are often unavailable to user applications. The few prior user-level RCU implementations either provided inefficient read-side primitives or restricted the application architecture. This paper fills this gap by describing efficient and flexible RCU implementations based on primitives commonly available to user-level applications.

Finally, this paper compares these RCU implementations with each other and with standard locking, which enables choosing the best mechanism for a given workload. This work opens the door to widespread user-application use of RCU.

Index Terms—D.4.1.f Synchronization < D.4.1 Process Management < D.4 Operating Systems < D Software/Software Engineering, D.4.1.g Threads < D.4.1 Process Management < D.4 Operating Systems < D Software/Software Engineering, D.4.1.a Concurrency < D.4.1 Process Management < D.4 Operating Systems < D Software/Software Engineering

I. INTRODUCTION

READ-COPY UPDATE (RCU) is a synchronization technique that was added to the Linux kernel in October of 2002. In contrast with conventional locking techniques that ensure mutual exclusion among all threads, or with reader-writer locks that allow readers to proceed concurrently with each other, but not with updaters, RCU permits both readers and updaters to make concurrent forward progress. RCU ensures that reads are coherent by maintaining multiple versions of objects and ensuring that each version remains intact until the completion of all RCU read-side critical sections that might reference that version. RCU defines and uses efficient and scalable mechanisms for publishing and reading new

versions of an object and for deferring reclamation of old versions. These mechanisms distribute the work between read and update paths so as to make read paths extremely fast, typically more than an order of magnitude faster than uncontended locking. RCU's light-weight read paths support the increasing need to track read-mostly connectivity, hardware-configuration, and security-policy data. Other mechanisms must be used to coordinate among multiple writers, for example locking, transactions, non-blocking synchronization, or single designated updater thread.

Techniques similar to RCU have appeared in several operating-system kernels [1, 2, 3, 4, 5], and, as shown in Figure 1, RCU is heavily used in the Linux kernel [6]. One reason RCU is heavily used is that it eases lock-based programming when the locks themselves are dynamically created and destroyed, which occurs frequently in concurrent programs. However, RCU is not heavily used in applications, in part because prior user-level RCU-like algorithms severely constrained application design [7], incurred heavy read-side overhead [8, 9], or relied on sequential consistency and garbage collection [10, 11]. The popularity of RCU in operating-system kernels owes much to the fact that kernels can accommodate the global constraints imposed by the high-performance quiescent-state based reclamation (QSBR) class of RCU implementations. QSBR implementations provide unmatched performance and scalability for read-mostly data structures on cache-coherent shared-memory multiprocessors [7], even with weakly ordered hardware and compilers.

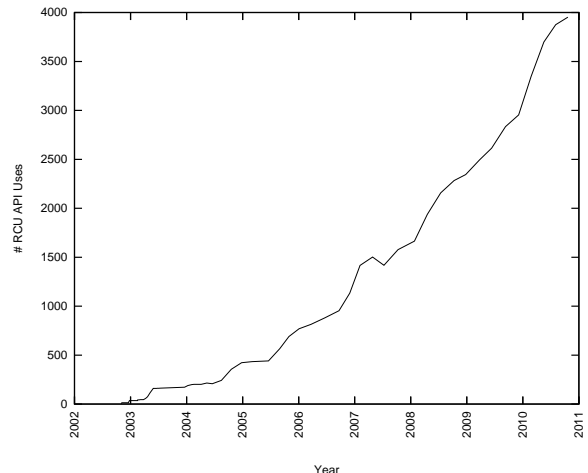


Fig. 1. Linux-Kernel Usage of RCU

Manuscript received August 17, 2009; revised November 12, 2010

Mathieu Desnoyers (mathieu.desnoyers@efficios.com) is with EfficiOS, work done while at the Computer and Software Engineering Department, Ecole Polytechnique de Montreal.

Paul E. McKenney (paulmck@linux.vnet.ibm.com) works at the IBM Linux Technology Center on the Linaro project.

Alan S. Stern (stern@rowland.harvard.edu) is with the Rowland Institute, Harvard University.

Michel R. Dagenais (michel.dagenais@polymtl.ca) is with the Computer and Software Engineering Department, Ecole Polytechnique de Montreal.

Jonathan Walpole (walpole@cs.pdx.edu) is with the Computer Science Department, Portland State University.

This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

Whereas we cannot yet put forward a single user-level RCU implementation that is ideal for all environments, the three classes of RCU implementations described in this paper should suffice for most user-level uses of RCU.

This article is organized as follows: Section II first provides a brief overview of RCU, including RCU semantics. Then, Section III describes user-level scenarios that could benefit from RCU. This is followed by the presentation of three classes of RCU implementation in Section IV. Section V presents experimental results, comparing RCU implementations to each other and to locking, and finally Section VI presents conclusions and recommendations.

II. BRIEF OVERVIEW OF RCU

This overview begins with an introduction to RCU concepts in Section II-A. Section II-B shows how to delete an element from an RCU-protected linked list in spite of concurrent readers. Section II-C presents a list of informal RCU desiderata, which details the goals pursued in this work. Finally, Section II-D gives a more detailed description of RCU semantics and guarantees.

A. Conceptual View of RCU Algorithms

RCU readers execute within *RCU read-side critical sections*. Each such critical section begins with `rcu_read_lock()`, ends with `rcu_read_unlock()`, and may contain `rcu_dereference()` or equivalent functions that access pointers to RCU-protected data structures. These pointer-access functions implement the notion of a dependency-ordered load, also known as a *memory_order_consume* load [12], which suppresses aggressive code-motion compiler optimizations and generates a simple load on any system other than DEC Alpha, where it generates a load followed by a memory-barrier instruction. The performance benefits of RCU are due to the fact that `rcu_read_lock()` and `rcu_read_unlock()` are exceedingly fast. In fact, Section IV-C will show how these two primitives can incur exactly zero overhead, as they do in server-class Linux-kernel builds [13].

When a thread is not in an RCU read-side critical section, it is in a *quiescent state*. A quiescent state that persists for a significant time period is an *extended quiescent state*. Any time period during which every thread has been in at least one quiescent state is a *grace period*; this implies that every RCU read-side critical section that starts before a grace period must end before that grace period does. Distinct grace periods may overlap, either partially or completely. Any time period that includes a grace period is by definition itself a grace period [13, 14]. Each grace period is guaranteed to complete as long as all read-side critical sections are finite in duration; thus even a constant flow of such critical sections is unable to extend an RCU grace period indefinitely.

Suppose that readers enclose each of their data-structure traversals in an RCU read-side critical section. If an updater first removes an element from such a data structure and then waits for a grace period, there can be no more readers accessing that element. The updater can then carry out destructive operations, for example freeing the element, without disturbing

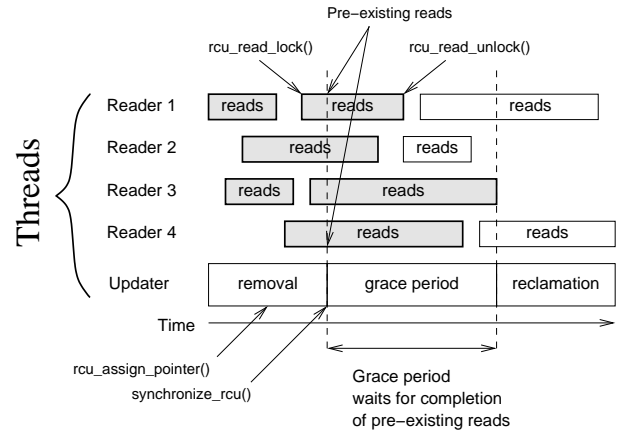


Fig. 2. Schematic of RCU Grace Period and Read-Side Critical Sections

any readers. A high-level schematic of such an RCU-based algorithm is shown in Figure 2. Here, each box labeled “reads” is an RCU read-side critical section.

Each row of read-side critical sections denotes a separate thread, for a total of four read-side threads. The bottom row of the figure denotes a fifth thread performing an RCU update. This RCU update is split into two phases, a removal phase on the lower left of the figure and a reclamation phase on the lower right. These two phases must be separated by a grace period, for example via the `synchronize_rcu()` primitive, which initiates a grace period and waits for it to finish. During the removal phase, the RCU update removes elements from a shared data structure (possibly inserting some as well) by calling `rcu_assign_pointer()` or an equivalent pointer-replacement function. The `rcu_assign_pointer()` primitive implements the notion of store release [12], which on sequentially consistent and total-store-ordered systems compiles to a simple assignment. Pointers stored by `rcu_assign_pointer()` can be fetched from within read-side critical sections by `rcu_dereference()`. The removed data elements will only be accessible to read-side critical sections that ran concurrently with the removal phase (shown in gray), which are guaranteed to complete before the grace period ends. Therefore the reclamation phase can safely free the data elements removed by the removal phase.¹

A single grace period can serve multiple removal phases, even those carried out by multiple updaters. Furthermore, the overhead of tracking RCU grace periods can be piggybacked on existing process-scheduling operations, to which RCU adds a small constant overhead. For some common workloads, the grace-period-tracking overhead of RCU during a given time interval may be amortized over an arbitrarily large number of RCU updates in that same interval [17], resulting in average per-RCU-update overheads arbitrarily close to zero.

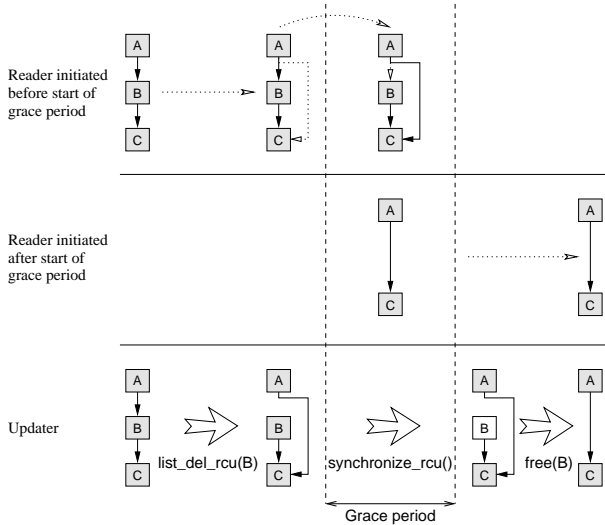


Fig. 3. RCU Linked-List Deletion

B. RCU Deletion From a Linked List

RCU-protected data structures in the Linux kernel include linked lists, hash tables, radix trees, and a number of custom-built data structures. Figure 3 shows how RCU may be used to delete an element from a linked list that is concurrently being traversed by RCU readers, as long as each reader conducts its traversal within the confines of a single RCU read-side critical section. The first and second rows present the data structure from the viewpoint of a reader thread that started before (first row) or after (second row) the grace period began. The last row of the figure shows the updater's view of the data structure.

The first column of the figure shows a singly-linked list with elements A, B, and C. Any reader initiated before the grace period might hold references to any of these elements.

The `list_del_rcu()` routine unlinks element B from the list, but leaves the link from B to C intact, as shown on the second column of the figure. This permits readers already referencing B to advance to C, as shown on the second and third columns of the figure. The transition from the second to the third column shows element B disappearing from the reader-thread viewpoint. During this transition, element B moves from *globally visible*, where any reader may obtain a new reference, to *locally visible*, where only readers already having a reference can see element B.

The `synchronize_rcu()` primitive waits for a grace period, after which all pre-existing read-side critical sections will have completed, resulting in the state shown in the fourth column of the figure, where readers no longer hold references to element B. Element B's transition from *locally visible* to *private* is denoted by the white background for the B box. It is then safe to invoke `free()`, reclaiming element B's memory, as shown in the last column of the figure.

Although RCU has many uses, this list-deletion process is frequently used to replace reader-writer locking [18].

¹Interestingly enough, placing non-blocking-synchronization (NBS) [15] updates in RCU read-side critical sections admits the same simplifications to NBS algorithms that are commonly provided by automatic garbage collectors. In particular, this approach avoids the ABA problem [16].

C. User-Space RCU Desiderata

Extensive use of RCU within the practitioner community has lead to the following user-space RCU desiderata:

- 1) RCU read-side primitives must have $O(1)$ computational complexity with a small constant, thus enabling real-time use and avoiding lock-based deadlocks. "Small constant" means avoiding expensive operations such as cache misses, atomic instructions, memory barriers, and, where feasible, conditional branches [19].
- 2) RCU read-side primitives should be usable in all contexts, including nested within other RCU read-side critical sections and inside user-level signal handlers [13].
- 3) RCU read-side primitives must be unconditional, with neither failure nor retry, thus avoiding livelocks [20].
- 4) RCU readers must not starve writers, even given arbitrarily high rates of time-bounded read-side critical sections.
- 5) RCU read-side critical sections may not contain operations that wait for a grace period, such as `synchronize_rcu()` (it would self-deadlock), nor may they acquire locks that are held across calls to `synchronize_rcu()`. However, non-interfering lock acquisition/release and other non-idempotent operations such as I/O should be permitted [21].
- 6) Mutating RCU-protected data structures must be permitted within RCU read-side critical sections, for example by acquiring the lock used to protect updates [21]. Such lock acquisitions can be thought of as unconditional read-to-write upgrades. However, any lock acquired within a read-side critical section cannot be held while waiting for a grace period.
- 7) RCU primitives should be independent of memory allocator design and implementation [20].

Although in-kernel RCU implementations are common, making them available to user applications is not practical. Firstly, many kernel-level RCU implementations assume that RCU read-side critical sections cannot be preempted, which is not the case at user level. Secondly, a user application invoking kernel-level RCU primitives could hang the system by remaining in an RCU read-side critical section indefinitely. Finally, invoking any in-kernel RCU implementation from user-level code introduces system-call overhead, violating the first desideratum above.

In contrast, the RCU implementations described in Section IV are designed to meet the above list of desiderata with acceptably low overheads.

D. Overview of RCU Semantics

RCU semantics comprise the *grace-period guarantee* and the *publication guarantee*. As noted earlier, concurrent modifications of an RCU-protected data structure must be coordinated by some other mechanism, for example, locking.

1) *Grace-Period Guarantee*: As noted in Section II-A, RCU read-side critical sections are delimited by `rcu_read_lock()` and `rcu_read_unlock()`, and RCU grace periods are periods of time such that all RCU read-side critical sections in existence at the beginning of a given grace period have completed before its end.

Somewhat more formally, consider a group of statements R_i within a single RCU read-side critical section:

```
rcu_read_lock();
R0; R1; R2; ...;
rcu_read_unlock();
```

Consider also groups of statements M_m (some of which may mutate shared data structures) and D_n (some of which may destroy shared data structures) separated by `synchronize_rcu()`:

```
M0; M1; M2; ...; synchronize_rcu(); D0; D1; D2; ...;
```

Then the following holds, where “ \rightarrow ” indicates that the statement on the left executes prior to that on the right:²

$$\forall m, i(M_m \rightarrow R_i) \vee \forall i, n(R_i \rightarrow D_n). \quad (1)$$

In other words, a given read-side critical section cannot extend beyond both sides of a grace period. (Figure 2 above provides a striking illustration of this idea.) Formulas 2 and 3 follow straightforwardly and are often used to validate uses of RCU (“ \Rightarrow ” denotes logical implication):

$$\exists i, m(R_i \rightarrow M_m) \Rightarrow \forall j, n(R_j \rightarrow D_n), \quad (2)$$

$$\exists n, i(D_n \rightarrow R_i) \Rightarrow \forall m, j(M_m \rightarrow R_j). \quad (3)$$

In other words, if any statement in a given read-side critical section executes prior to any statement preceding a given grace period, then all statements in that critical section must execute prior to any statement following this same grace period. Conversely, if any statement in a given read-side critical section executes after any statement following a given grace period, then all statements in that critical section must execute after any statement preceding this same grace period.³

This guarantee permits RCU-based algorithms to trivially avoid a number of difficult race conditions whose resolution can otherwise result in poor performance, limited scalability, and great complexity. However, on weakly ordered systems this guarantee is insufficient. We also need some way to guarantee that if a reader sees a pointer to a new data structure, it will also see the values stored during initialization of that structure. This guarantee is presented in the next section.

2) *Publication Guarantee*: Note well that the statements R_i and M_m may execute concurrently, even in the case where R_i is referencing the same data element that M_m is concurrently modifying. The publication guarantee provided by `rcu_assign_pointer()` and `rcu_dereference()` handles

this concurrency correctly and easily: even on weakly ordered systems, any dereference of a pointer returned by `rcu_dereference()` is guaranteed to see any change prior to the corresponding `rcu_assign_pointer()`, including any change prior to any earlier `rcu_assign_pointer()` involving that same pointer.

Somewhat more formally, suppose that `rcu_assign_pointer()` is used as follows:

```
I0; I1; I2; ...; rcu_assign_pointer(g, p);
```

where each I_i is a statement (including those initializing fields in the structure referenced by local pointer p), and where global pointer g is visible to reading threads. This initialization sequence is part of the M_m sequence of statements discussed earlier.

The body of a canonical RCU read-side critical section would appear as follows:

```
...; q = rcu_dereference(g); A0; A1; A2; ...;
```

where q is a local pointer, g is the same global pointer updated by the earlier `rcu_assign_pointer()` (and possibly updated again by later invocations of `rcu_assign_pointer()`), and some of the A_i statements dereference q to access fields initialized by some of the I_i statements. This sequence of `rcu_dereference()` followed by A_i statements is part of the R_i statements discussed earlier.

Then we have the following, where M is the `rcu_assign_pointer()` and R is the `rcu_dereference()`:⁴

$$M \rightarrow R \Rightarrow \forall i, j(I_i \rightarrow A_j). \quad (4)$$

In other words, if a given `rcu_dereference()` statement accesses the value stored to g by a given `rcu_assign_pointer()`, then all statements dereferencing the pointer returned by that `rcu_dereference()` must see the effects of any initialization statements preceding that `rcu_assign_pointer()` or any earlier `rcu_assign_pointer()` storing to g .

This guarantee provides readers a consistent view of newly added data.

3) *Uses of RCU Guarantees*: These grace-period and publication guarantees are extremely useful, but in ways that are not always immediately obvious. This section therefore describes a few of the most common uses of these guarantees.

First, they can provide *existence guarantees* [1], so that any RCU-provided data element accessed anywhere within a given RCU read-side critical section is guaranteed to remain intact throughout that RCU read-side critical section. Existence guarantees are provided by ensuring that an RCU grace period elapses between the moment a given data element is rendered inaccessible to readers and the moment this element’s memory is reclaimed and/or reused.

Second, the RCU guarantees can provide *type-safe memory* [22] by integrating RCU grace periods into the memory allocator—for example, the Linux kernel’s slab allocator provides type-safe memory when the `SLAB_DESTROY_BY_RCU`

²This description is deliberately vague. More-precise definitions of “ $A \rightarrow B$ ” [12, Section 1.10] consider the individual memory locations accessed by both A and B , and order the two statements with respect to each of those accesses. For our purposes, what matters is that $A \rightarrow B$ and $B \rightarrow A$ can’t both hold. If A and B execute concurrently then both relations may fail. However as a special case, if A is a store to a variable and B is a load from that same variable, then either $A \rightarrow B$ (B reads the value stored by A or a later value) or $B \rightarrow A$ (B reads a value prior to that stored by A).

³Some RCU implementations may choose to weaken this guarantee so as to exclude special-purpose operations such as MMIO accesses, I/O-port instructions, and self-modifying code. Such weakening is appropriate on systems where ordering these operations is expensive and where the users of that RCU implementation either (1) are not using these operations or (2) insert the appropriate ordering into their own code, as many system calls do.

⁴Formula 4 is not strictly correct. On some architectures, $I_i \rightarrow A_j$ is guaranteed only if A_j carries a data dependency from the local pointer q ; otherwise the CPU may reorder or speculatively execute A_j before the `rcu_dereference()` call. In practice this restriction does not lead to problems.

flag is specified. This integration is accomplished by permitting a freed data element to be immediately reused, but only if its type remains unchanged. The allocator must ensure that an RCU grace period elapses before that element's type is permitted to change. This approach guarantees that any data element accessed within a given RCU read-side critical section retains its type throughout that RCU read-side critical section.

Finally, as noted earlier, RCU's grace-period and publication guarantees can often be used to replace reader-writer locking.

As a result, the grace-period and publication guarantees enable a wide variety of algorithms and data structures providing extremely low read-side overheads for read-mostly data structures [7, 13, 18, 19]. Again, note that concurrent updates must be handled by some other synchronization mechanism.

With this background on RCU, we are ready to consider how it might be used in user-level applications.

III. USER-SPACE RCU USAGE SCENARIOS

The user-level RCU work described later in this paper was inspired by the need to reduce the overhead and improve the scalability of the LTTng userspace tracer (UST), which carries out performance analysis and monitoring of user-mode applications [23, 24]. UST imposes important constraints on the user-level RCU implementation. Firstly, UST cannot require source-level modifications to the application under test, which rules out the QSBR approach that will be presented in Section IV-C. Secondly, UST must support instrumentation of execution sites selected by the user at runtime. Because the user is permitted to instrument signal handlers and library functions, RCU read-side critical sections must be nestable.

BIND, a major domain-name server used for Internet domain-name resolution, is facing scalability issues [25]. Since domain names are read often but rarely updated, using user-level RCU might be beneficial. Others have mentioned possibilities in financial applications. Finally, one can also argue that RCU has seen long use at user level in the guise of user-mode Linux.

In general, user-level RCU's area of applicability appears similar to that in the Linux kernel: to read-mostly data structures, especially in cases where stale data can be accommodated.

IV. CLASSES OF RCU IMPLEMENTATIONS

This section describes several classes of RCU implementations. Section IV-A first describes some primitives that might be unfamiliar to the reader, Section IV-B presents an example use of RCU, and then Sections IV-C, IV-D, and IV-E present user-space RCU implementations that are optimized for different use cases. The QSBR implementation presented in Section IV-C offers the best possible read-side performance, but requires that each thread periodically calls a function to announce that it is in a quiescent state, thus strongly constraining the application's design. The implementation presented in Section IV-D places almost no constraints on the application's design, thus being appropriate for use within a general-purpose library, but it has higher read-side overhead. Section IV-E presents an implementation having low read-side overhead and

```

1 #define ACCESS_ONCE(x)  (*(volatile typeof(x) *)&(x))
2
3 #define LOAD_SHARED(p)   ACCESS_ONCE(p)
4 #define STORE_SHARED(x, v)  ({ ACCESS_ONCE(x) = (v); })
5
6 #define barrier()        asm volatile("" : : : "memory")

```

Fig. 4. Shared-Memory Compiler Primitives

requiring only that the application give up one POSIX signal to RCU update processing. Finally, Section IV-F demonstrates how to create non-blocking RCU update primitives.

We start with a rough overview of some elements common to all three implementations. A global variable, `rcu_gp_ctr`, tracks grace periods. Each thread has a local variable indicating whether or not it is currently in a read-side critical section, together with a snapshot of `rcu_gp_ctr`'s value at the time the read-side critical section began. The `synchronize_rcu()` routine iterates over all threads, using these snapshots to wait so long as any thread is in a read-side critical section that started before the current grace period.

Grace periods can be tracked in two different ways. The simplest method, used in Section IV-C, is for `rcu_gp_ctr` simply to count grace periods. Because of the possibility of counter overflow, this method is suitable only for 64-bit architectures. The other method divides each grace period up into two phases and makes `rcu_gp_ctr` track the current phase. As explained in Section IV-D below, this approach avoids the problem of counter overflow at the cost of prolonging grace periods; hence it can be used on all architectures.

A. Common Primitives

This section describes a number of primitives that are used by examples in later sections.

Figure 4 introduces primitives dealing with shared memory at the compiler level. The `ACCESS_ONCE()` primitive applies `volatile` semantics to its argument. The `LOAD_SHARED()` primitive prohibits any compiler optimization that might otherwise turn a single load into multiple loads (as might happen under heavy register pressure), and vice versa. The `STORE_SHARED()` primitive acts as an assignment statement, but prohibits any compiler optimization that might otherwise turn a single store into multiple stores and vice versa. The `barrier()` primitive prohibits any compiler code-motion optimization that might otherwise move loads or stores across the `barrier()`. Among other things, we use it to force the compiler to reload values on each pass through a wait loop. It is strictly a compiler directive; it emits no code.

The `smp_mb()` primitive (not shown in Figure 4 because its implementation is architecture-specific) emits a full memory barrier, for example, the `sync` instruction on the PowerPC architecture ("mb" stands for "memory barrier"). The fundamental ordering property of memory barriers can be expressed as follows: Suppose one thread executes the statements

$A_0; A_1; A_2; \dots; \text{smp_mb}(); B_0; B_1; B_2; \dots;$

and another thread executes the statements

$C_0; C_1; C_2; \dots; \text{smp_mb}(); D_0; D_1; D_2; \dots;$

```

1 pthread_mutex_t rcu_gp_lock =
2   PTHREAD_MUTEX_INITIALIZER;
3 LIST_HEAD(registry);
4
5 struct rcu_reader {
6   unsigned long ctr;
7   char need_mb;
8   struct list_head node;
9   pthread_t tid;
10 };
11 struct rcu_reader __thread rcu_reader;
12
13 void rcu_register_thread(void)
14 {
15   rcu_reader.tid = pthread_self();
16   mutex_lock(&rcu_gp_lock);
17   list_add(&rcu_reader.node, &registry);
18   mutex_unlock(&rcu_gp_lock);
19   rcu_thread_online();
20 }
21
22 void rcu_unregister_thread(void)
23 {
24   rcu_thread_offline();
25   mutex_lock(&rcu_gp_lock);
26   list_del(&rcu_reader.node);
27   mutex_unlock(&rcu_gp_lock);
28 }

```

Fig. 5. RCU Reader-Thread Registry

Then $\exists m, n (B_m \rightarrow C_n)$ implies $\forall i, j (A_i \rightarrow D_j)$.

These primitives can be expressed directly in terms of the upcoming C++0x standard [12]. For the `smp_mb()` primitive this correspondence is not exact; our memory barriers are somewhat stronger than the standard's `atomic_thread_fence(memory_order_seq_cst)`. The `LOAD_SHARED()` primitive maps to `x.load(memory_order_relaxed)` and `STORE_SHARED()` to `x.store(memory_order_relaxed)`. The `barrier()` primitive maps to `atomic_signal_fence(memory_order_seq_cst)`. In addition, `rcu_dereference()` maps to `x.load(memory_order_consume)` and `rcu_assign_pointer()` maps to `x.store(v, memory_order_release)`.

Figure 5 introduces declarations and data structures used by all implementations, along with the process-wide registry tracking all threads containing RCU read-side critical sections. The pthread mutex `rcu_gp_lock` (lines 1–2) serializes addition (line 17), removal (line 26) and iteration (which will be presented in Figures 8, 10, and 13) on the reader thread list (list head is at line 3 and nodes at line 8 of Figure 5). This `rcu_gp_lock` also serializes grace-period detection and updates of the global grace-period counter. The `pthread_mutex_t` type is defined by the pthread library for mutual exclusion variables; the `mutex_lock()` primitive acquires a `pthread_mutex_t` instance and `mutex_unlock()` releases it. Line 11 introduces the `rcu_reader` per-thread variable, through which each access to per-thread registry information is performed. These per-thread variables are declared via the `__thread` storage-class specifier, as specified by C99 [26], and as extended by gcc to permit cross-thread access to per-thread variables.⁵ The `tid` field of `struct`

```

1 struct lin_coefs {
2   double a, b, c;
3 };
4
5 struct lin_coefs *lin_approx_p;
6
7 void control_loop(void)
8 {
9   struct lin_coefs *p;
10  struct lin_coefs lc;
11  double x, y;
12
13  rcu_register_thread();
14  for (;;) {
15    x = measure();
16    rcu_read_lock();
17    p = rcu_dereference(lin_approx_p);
18    lc = *p;
19    rcu_read_unlock();
20    y = lin_approx(x, lc.a, lc.b, lc.c);
21    do_control(y);
22    sleep_us(50);
23  }
24 }
25
26 void lin_approx_loop(void)
27 {
28   struct lin_coefs lc[2];
29   int cur_idx = 0;
30   struct lin_coefs *p;
31
32   rcu_register_thread();
33   for (;;) {
34     cur_idx = !cur_idx;
35     p = &lc[cur_idx];
36     calc_lin_approx(p);
37     rcu_assign_pointer(lin_approx_p, p);
38     synchronize_rcu();
39     sleep(5);
40   }
41 }

```

Fig. 6. RCU Use Case: Real-Time Control

`rcu_reader` entry contains the thread identifier returned by `pthread_self()`. This identifier is used to send signals to specific threads by signal-based RCU, presented in Section IV-E. The `need_mb` field is also used by the signal-based RCU implementation to keep track of threads which have executed their signal handlers. The `rcu_thread_online()` and `rcu_thread_offline()` primitives mark the online status of reader threads and are specific to the QSBR RCU implementation shown in Section IV-C.

B. Example RCU Use Case

Consider a real-time closed-loop control application governed by a complex mathematical control law, where the control loop must execute at least every 100 microseconds. Suppose that this control law is too computationally expensive to be computed each time through the control loop, so a simpler linear approximation is used instead. As environmental parameters (such as temperature) slowly change, a new linear approximation must be computed from the full mathematical control law. Therefore, a lower-priority task computes a set of three coefficients for the linear approximation periodically, for example, every five seconds. The control-loop task then makes use of the most recently produced set of coefficients.

Of course, it is critically important that each control-loop computation use a consistent set of coefficients. It is therefore necessary to use proper synchronization between the control

⁵This extension is quite common. One reason C99 does not mandate this extension is to avoid prohibiting implementations that map any given per-thread variable to a single address for all threads [27].

loop and the production of a new set of coefficients. In contrast, use of a slightly outdated set of coefficients is acceptable. We can therefore use RCU to carry out the synchronization, as shown in the (fanciful) implementation in Figure 6. The overall approach is to periodically publish a new set of coefficients using `rcu_assign_pointer()`, which are subscribed to using `rcu_dereference()`. The `synchronize_rcu()` primitive is used to prevent overwriting a set of coefficients that is still in use. Because only one thread will be updating the coefficients, update-side synchronization is not required.

The `control_loop()` function is invoked from the thread performing closed-loop control. It first invokes `rcu_register_thread()` to make itself known to RCU, and then enters an infinite loop performing the real-time control actions. Each pass through this loop first measures the control input value, then enters an RCU read-side critical section to obtain the current set of coefficients, uses `lin_approx()` to compute a new control value, uses `do_control()` to output this value, and finally does a 50-microsecond delay.⁶ The use of `rcu_dereference()` ensures that the coefficients will be fully initialized, even on weakly ordered systems, and the use of `rcu_read_lock()` and `rcu_read_unlock()` ensure that subsequent grace periods cannot complete until the coefficients are completely copied.

The `lin_approx_loop()` function is invoked from the thread that is to periodically compute a new set of coefficients for use by `control_loop()`. As with `control_loop()`, it first invokes `rcu_register_thread()` to make itself known to RCU, and then enters an infinite loop performing the coefficient calculations. To accomplish this, it defines an array of two sets of coefficients along with an index that selects which of the two sets is currently in effect. Each pass through the loop computes a new set of coefficients into the element of the array that is not currently being used by readers, uses `rcu_assign_pointer()` to publish this new set, uses `synchronize_rcu()` to wait for `control_loop()` to finish using the old set, and finally waits for five seconds before repeating this process.

Because `rcu_dereference()` is wait-free with small overhead, this approach is well-suited to real-time systems running on multi-core systems. In contrast, approaches based on locking would require `control_loop()` to wait on `lin_approx_loop()` when the latter was installing a new set of coefficients, meaning that they might be subject to priority inversion.

C. Quiescent-State-Based Reclamation RCU

The QSBR RCU implementation provides near-zero read-side overhead, as has been presented earlier [7]. This section expands on that work by describing a similar QSBR implementation for 64-bit systems. The price of minimal overhead is that each thread in an application is required to periodically invoke `rcu_quiescent_state()` to announce that it resides in a quiescent state. This requirement can entail extensive application modifications, limiting QSBR's applicability.

⁶We are arbitrarily choosing a delay half that of the real-time deadline. An actual real-time application might compute the delay based on measuring the overhead of the code in the loop, or it might use timers.

```

1 #define RCU_GP_ONLINE 0x1
2 #define RCU_GP_CTR    0x2
3
4 unsigned long rcu_gp_ctr = RCU_GP_ONLINE;
5
6 static inline void rcu_read_lock(void)
7 {
8 }
9
10 static inline void rcu_read_unlock(void)
11 {
12 }
13
14 static inline void rcu_quiescent_state(void)
15 {
16     smp_mb();
17     STORE_SHARED(rcu_reader.ctr,
18                 LOAD_SHARED(rcu_gp_ctr));
19     smp_mb();
20 }
21
22 static inline void rcu_thread_offline(void)
23 {
24     smp_mb();
25     STORE_SHARED(rcu_reader.ctr, 0);
26 }
27
28 static inline void rcu_thread_online(void)
29 {
30     STORE_SHARED(rcu_reader.ctr,
31                 LOAD_SHARED(rcu_gp_ctr));
32     smp_mb();
33 }

```

Fig. 7. RCU Read Side Using Quiescent States

QSBR uses these quiescent-state announcements to *approximate* the extent of read-side critical sections, treating the interval between two successive announcements as a single, large critical section. As a consequence, the `rcu_read_lock()` and `rcu_read_unlock()` primitives need do nothing and will often be inlined and optimized away, as in fact they are in server builds of the Linux kernel. QSBR thus provides unsurpassed read-side performance, albeit at the cost of longer grace periods. When QSBR is being used to reclaim memory, these longer grace periods result in more memory being consumed by data structures waiting for grace periods, in turn resulting in the classic CPU-memory tradeoff.

The 64-bit global counter `rcu_gp_ctr` shown in Figure 7 contains 1 in its low-order bit and contains the current grace-period number in its remaining bits.⁷ It may be accessed at any time by any thread but may be updated only by the thread holding `rcu_gp_lock`. The `rcu_quiescent_state()` function simply copies a snapshot of the global counter to the per-thread `rcu_reader.ctr` variable (which may be modified only by the corresponding thread). The 1 in the low-order bit serves to indicate that the reader thread is not in an extended quiescent state. The two memory barriers enforce ordering of preceding and subsequent accesses.

As an alternative to periodically invoking `rcu_quiescent_state()`, threads may use the `rcu_thread_offline()` and `rcu_thread_online()` APIs to mark the beginnings and ends of extended quiescent states. These three

⁷If quiescent states are counted, logged, or otherwise recorded, then this information may be used in place of the global `rcu_gp_ctr` counter [5]. For example, context switches are counted in the Linux kernel's QSBR implementation, and some classes of user applications are expected to have similar operations [5, Section 3.4]. However, the separate global `rcu_gp_ctr` counter permits discussion independent of any particular application.

```

1 void synchronize_rcu(void)
2 {
3     unsigned long was_online;
4
5     was_online = rcu_reader.ctr;
6     smp_mb();
7     if (was_online)
8         STORE_SHARED(rcu_reader.ctr, 0);
9     mutex_lock(&rcu_gp_lock);
10    update_counter_and_wait();
11    mutex_unlock(&rcu_gp_lock);
12    if (was_online)
13        STORE_SHARED(rcu_reader.ctr,
14                      LOAD_SHARED(rcu_gp_ctr));
15    smp_mb();
16 }
17
18 static void update_counter_and_wait(void)
19 {
20     struct rcu_reader *index;
21
22     STORE_SHARED(rcu_gp_ctr, rcu_gp_ctr + RCU_GP_CTR);
23     barrier();
24     list_for_each_entry(index, &registry, node) {
25         while (rcu_gp_ongoing(&index->ctr))
26             msleep(10);
27     }
28 }
29
30 static inline int rcu_gp_ongoing(unsigned long *ctr)
31 {
32     unsigned long v;
33
34     v = LOAD_SHARED(*ctr);
35     return v && (v != rcu_gp_ctr);
36 }

```

Fig. 8. RCU Update Side Using Quiescent States

functions must not be called from within read-side critical sections. The `rcu_thread_offline()` function simply sets the per-thread `rcu_reader.ctr` variable to zero, indicating that this thread is in an extended quiescent state. Memory ordering is needed only at the beginning of the function because the following code cannot be in a read-side critical section. The `rcu_thread_online()` function is similar to `rcu_quiescent_state()`, except that it requires a memory barrier only at the end. Note that all the functions in Figure 7 are wait-free because they each execute a fixed sequence of instructions.

Figure 8 shows `synchronize_rcu()` and its two helper functions, `update_counter_and_wait()` and `rcu_gp_ongoing()`. The `synchronize_rcu()` function puts the current thread into an extended quiescent state if it is not already in one, forces ordering of the caller's accesses, and invokes `update_counter_and_wait()` under the protection of `rcu_gp_lock`. The `update_counter_and_wait()` function increments the global `rcu_gp_ctr` variable by 2 (recall that the lower bit is reserved for readers to indicate whether they are in an extended quiescent state). It then uses `list_for_each_entry()` to scan all of the threads, invoking `rcu_gp_ongoing()` on each, thus waiting until all threads have exited any pre-existing RCU read-side critical sections. The `barrier()` macro on line 23 prevents the compiler from checking the threads before updating `rcu_gp_ctr`, which could result in deadlock. The `msleep()` function on line 26 blocks for the specified number of milliseconds, in this case chosen arbitrarily. Finally, the `rcu_gp_ongoing()` function checks to see if the specified

counter indicates that the corresponding thread might be in a pre-existing RCU read-side critical section. It accomplishes this with the two-part check on line 35: if the counter is zero, the thread is in an extended quiescent state, while if the counter is equal to `rcu_gp_ctr`, the thread is in an RCU read-side critical section that began after beginning of the current RCU grace period, and therefore need not be waited for.

We specify a 64-bit `rcu_gp_ctr` to avoid overflow. The fundamental issue is that there is no way to copy a value from one memory location to another atomically. Suppose reader thread T is preempted just before executing the `STORE_SHARED()` call in `rcu_thread_online()`, after the `LOAD_SHARED()` call has returned. Until the store takes place the thread is still in its extended quiescent state, so there is nothing to prevent other threads from making multiple calls to `synchronize_rcu()` (and thereby incrementing `rcu_gp_ctr`) during the preemption delay. If the counter cycles through all but one of its values, the stale value finally stored in thread T 's `rcu_reader.ctr` will actually be `rcu_gp_ctr`'s *next* value. As a result, if another thread later calls `synchronize_rcu()` after T has entered a read-side critical section, then `update_counter_and_wait()` might return before T has left this critical section, in violation of RCU's semantics. With 64 bits, thousands of years would be required to overflow the counter and hence the possibility may be ignored. However, given a 32-bit `rcu_gp_ctr` this scenario is possible; hence 32-bit implementations should instead adapt the two-phase scheme discussed in Section IV-D [28].

Given that they are empty functions, the `rcu_read_lock()` and `rcu_read_unlock()` primitives are wait-free under the most severe conceivable definition [29]. Because it waits for pre-existing readers, `synchronize_rcu()` is not non-blocking. Section IV-F describes how RCU updates can support non-blocking algorithms in the same sense as they are supported by garbage collectors.

The need for periodic `rcu_quiescent_state()` invocations can make QSBR impossible to use in some situations, such as within libraries. In addition, this QSBR implementation does not allow concurrent `synchronize_rcu()` calls to share grace periods—a straightforward optimization, but beyond the scope of this paper. That said, this code can form the basis for a production-quality RCU implementation [28].

Another limitation of the quiescent-state approach is that applications requiring read-side critical sections in signal handlers must disable signals around invocation of `rcu_quiescent_state()`, and for the duration of extended quiescent states marked by `rcu_thread_offline()` and `rcu_thread_online()`. In addition, applications needing to invoke `synchronize_rcu()` while holding a lock must ensure that all acquisitions of that lock invoke `rcu_thread_offline()`, presumably via a wrapper function encapsulating the lock-acquisition primitive. Applications needing read-side critical sections within signal handlers or that need to invoke `synchronize_rcu()` while holding a lock might therefore be better served by the RCU implementations described in subsequent sections.

A semi-formal verification that this implementation satisfies the grace-period guarantee (in the absence of overflow) is


```

1 #define RCU_GP_CTR_PHASE 0x10000
2 #define RCU_NEST_MASK    0x0ffff
3 #define RCU_NEST_COUNT   0x1
4
5 unsigned long rcu_gp_ctr = RCU_NEST_COUNT;
6
7 static inline void rcu_read_lock(void)
8 {
9     unsigned long tmp;
10
11     tmp = rcu_reader.ctr;
12     if (!(tmp & RCU_NEST_MASK)) {
13         STORE_SHARED(rcu_reader.ctr,
14                     LOAD_SHARED(rcu_gp_ctr));
15         smp_mb();
16     } else {
17         STORE_SHARED(rcu_reader.ctr, tmp + RCU_NEST_COUNT);
18     }
19 }
20
21 static inline void rcu_read_unlock(void)
22 {
23     smp_mb();
24     STORE_SHARED(rcu_reader.ctr,
25                 rcu_reader.ctr - RCU_NEST_COUNT);
26 }

```

Fig. 9. RCU Read Side Using Memory Barriers

presented in Appendix A.

The next section discusses an RCU implementation that is safe for use in libraries, the tradeoff being higher read-side overhead.

D. General-Purpose RCU

The general-purpose RCU implementation can be used in any software environment, including library functions that are not aware of the design of the calling application. Such library functions cannot guarantee that each application's threads will invoke `rcu_quiescent_state()` sufficiently often, nor can they ensure that threads will invoke `rcu_thread_offline()` and `rcu_thread_online()` around each blocking system call. General-purpose RCU therefore does not require that these three functions ever be invoked.

In addition, this general-purpose implementation avoids the counter-overflow problem discussed in Section IV-C by using a different approach to track grace periods. Each grace period is divided into two *grace-period phases*, and instead of a free-running grace-period counter, a single-bit toggle is used to number the phases within a grace period. A given phase completes only after each thread's local snapshot either contains a copy of the phase's number or indicates the thread is in a quiescent state. If RCU read-side critical sections are finite in duration, one of these two cases must eventually hold for each thread.

Because reader threads snapshot the value of `rcu_gp_ctr` whenever they enter an outermost read-side critical section, explicit tracking of critical-section nesting is required. Nevertheless, the extra read-side overhead is significantly less than a single compare-and-swap operation on most hardware, and a beneficial side effect is that all quiescent states are effectively extended quiescent states. Read-side critical-section nesting is tracked in the lower-order bits (`RCU_NEST_MASK`) of the per-thread `rcu_reader.ctr` variable, as shown in Figure 9. The grace-period phase number occupies only a single high-

```

1 void synchronize_rcu(void)
2 {
3     smp_mb();
4     mutex_lock(&rcu_gp_lock);
5     update_counter_and_wait();
6     barrier();
7     update_counter_and_wait();
8     mutex_unlock(&rcu_gp_lock);
9     smp_mb();
10 }
11
12 static void update_counter_and_wait(void)
13 {
14     struct rcu_reader *index;
15
16     STORE_SHARED(rcu_gp_ctr,
17                 rcu_gp_ctr ^ RCU_GP_CTR_PHASE);
18     barrier();
19     list_for_each_entry(index, &registry, node) {
20         while (rcu_gp_ongoing(&index->ctr))
21             msleep(10);
22     }
23 }
24
25 static inline int rcu_gp_ongoing(unsigned long *ctr)
26 {
27     unsigned long v;
28
29     v = LOAD_SHARED(*ctr);
30     return (v & RCU_NEST_MASK) &&
31         ((v ^ rcu_gp_ctr) & RCU_GP_CTR_PHASE);
32 }

```

Fig. 10. RCU Update Side Using Memory Barriers

order bit (`RCU_GP_CTR_PHASE`), so there is ample room to store the nesting level.

The `rcu_read_lock()` function first checks the per-thread nesting level to see if the calling thread was previously in a quiescent state, snapshotting the global `rcu_gp_ctr` grace-period phase number [30] and executing a memory barrier if it was, and otherwise simply incrementing the nesting level without changing the phase number. The low-order bits of `rcu_gp_ctr` are permanently set to show a nesting level of 1, so that the snapshot can store both the current phase number and the initial nesting level in a single atomic operation. The memory barrier ensures that the store to `rcu_reader.ctr` will be ordered before any access in the subsequent RCU read-side critical section. The `rcu_read_unlock()` function executes a memory barrier and decrements the nesting level. The memory barrier ensures that any access in the prior RCU read-side critical section is ordered before the nesting-level decrement.⁸ Even with the memory barriers, both `rcu_read_lock()` and `rcu_read_unlock()` are wait-free with maximum overhead smaller than many other synchronization primitives. Because they may be implemented as empty functions, `rcu_quiescent_state()`, `rcu_thread_offline()`, and `rcu_thread_online()` are omitted.

Figure 10 shows `synchronize_rcu()` and its two helper functions, `update_counter_and_wait()` and `rcu_gp_ongoing()`. The `synchronize_rcu()` function forces ordering of the caller's accesses (lines 3 and 9) and waits for two grace-period phases under the protection of `rcu_gp_lock`, as discussed earlier. The two phases are

⁸In C++0x, the weaker `store(memory_order_release)` barrier would suffice, but it is not supported by gcc.

separated by a `barrier()` to prevent the compiler from interleaving their accesses, which could result in starvation. The `update_counter_and_wait()` function is invoked to handle each grace-period phase. This function is similar to its counterpart in Figure 8, the only difference being that the update to `rcu_gp_ctr` toggles the phase number rather than incrementing a counter. The `rcu_gp_ongoing()` function is likewise similar to its earlier counterpart; it tests whether the specified snapshot indicates that the corresponding thread is in a non-quiescent state (the nesting level is nonzero) with a phase number different from the current value in `rcu_gp_ctr`.

To show why this works, let us verify that this two-phase approach properly obeys RCU's semantics, i.e., that any read-side critical section in progress when `synchronize_rcu()` begins will terminate before it ends. Suppose thread T is in a read-side critical section. Until the critical section terminates, T 's `rcu_reader.ctr` will show a nonzero nesting level, and its snapshot of the phase number will not change (since the phase number in `rcu_reader.ctr` changes only during an outermost `rcu_read_lock()` call). The invocation of `update_counter_and_wait()` during one of `synchronize_rcu()`'s grace-period phases will wait until T 's phase-number snapshot takes on the value 0, whereas the invocation during the other phase will wait until the phase-number snapshot takes on the value 1. Each of the two invocations will also complete if T 's nesting level takes on the value 0. But regardless of how this works out, it is clearly impossible for both phases to end before T 's read-side critical section has terminated. Appendix B presents a semi-formal verification of this reasoning.

A natural question is "Why doesn't a single grace-period phase suffice?" If `synchronize_rcu()` used a single phase then it would be essentially the same as the function in Figure 8, and it would be subject to the same overflow problem, exacerbated by the use of what is effectively a single-bit counter. In more detail, the following could occur:

- 1) Thread T invokes `rcu_read_lock()`, fetching the value of `rcu_gp_ctr`, but not yet storing it.
- 2) Thread U invokes `synchronize_rcu()`, including invoking `update_counter_and_wait()`, where it toggles the grace-period phase number in `rcu_gp_ctr` so that the phase number is now 1.
- 3) Because no thread is yet in an RCU read-side critical section, thread U completes `update_counter_and_wait()` and returns to `synchronize_rcu()`, which returns to its caller since it uses only one phase.
- 4) Thread T now stores the old value of `rcu_gp_ctr`, with its phase-number snapshot of 0, and proceeds into its read-side critical section.
- 5) Thread U invokes `synchronize_rcu()` once more, again toggling the global grace-period phase number, so that the number is again 0.
- 6) When Thread U examines Thread T 's `rcu_reader.ctr` variable, it finds that the phase number in the snapshot matches that of the global variable `rcu_gp_ctr`. Thread U therefore exits from `synchronize_rcu()`.
- 7) But Thread T is still in its read-side critical section, in

```

1 static inline void rcu_read_lock(void)
2 {
3     unsigned long tmp;
4
5     tmp = rcu_reader.ctr;
6     if (!(tmp & RCU_NEST_MASK)) {
7         STORE_SHARED(rcu_reader.ctr,
8                     LOAD_SHARED(rcu_gp_ctr));
9         barrier();
10    } else {
11        STORE_SHARED(rcu_reader.ctr, tmp + RCU_NEST_COUNT);
12    }
13 }
14
15 static inline void rcu_read_unlock(void)
16 {
17     barrier();
18     STORE_SHARED(rcu_reader.ctr,
19                 rcu_reader.ctr - RCU_NEST_COUNT);
20 }

```

Fig. 11. RCU Read Side Using Signals

```

1 void synchronize_rcu(void)
2 {
3     mutex_lock(&rcu_gp_lock);
4     force_mb_all_threads();
5     update_counter_and_wait();
6     barrier();
7     update_counter_and_wait();
8     force_mb_all_threads();
9     mutex_unlock(&rcu_gp_lock);
10 }

```

Fig. 12. RCU Update Side Using Signals

violation of RCU's semantics.

The extra overhead of a second grace-period phase is not regarded as a serious drawback since it affects only updaters, not readers. The overhead of the read-side memory barriers is more worrisome; the next section shows how it can be eliminated.

E. Low-Overhead RCU Via Signal Handling

On common multiprocessor hardware, the largest source of read-side overhead for general-purpose RCU is the memory barriers. One novel way to eliminate these barriers is to send POSIX signals from the update-side primitives. An unexpected but quite pleasant surprise is that this approach results in relatively simple read-side primitives. In contrast, those of older versions of the Linux kernel's preemptible RCU were notoriously complex [31].

The read-side primitives shown in Figure 11 are identical to those in Figure 9, except that lines 9 and 17 replace the memory barriers with compiler directives that suppress code-motion optimizations. The structures, variables, and constants are identical to those in Figures 5 and 9. As with the previous two implementations, both `rcu_read_lock()` and `rcu_read_unlock()` are wait-free.

The `synchronize_rcu()` primitive shown in Figure 12 is similar to that in Figure 10, the only changes being in lines 3–4 and 8–9. Instead of executing a memory barrier local to the current thread, this implementation forces all threads to execute a memory barrier using `force_mb_all_threads()`, and the two calls to this new function are moved inside the locked region because of the need to iterate

```

1 static void force_mb_all_threads(void)
2 {
3     struct rcu_reader *index;
4     list_for_each_entry(index, &registry, node) {
5         STORE_SHARED(index->need_mb, 1);
6         smp_mb();
7         pthread_kill(index->tid, SIGRCU);
8     }
9     list_for_each_entry(index, &registry, node) {
10         while (LOAD_SHARED(index->need_mb))
11             msleep(1);
12     }
13     smp_mb();
14 }
15
16 static void sigurcu_handler(int signo,
17                             siginfo_t *siginfo,
18                             void *context)
19 {
20     smp_mb();
21     STORE_SHARED(rcu_reader.need_mb, 0);
22     smp_mb();
23 }

```

Fig. 13. RCU Signal Handling for Updates

over the thread registry, which is protected by `rcu_gp_lock`. The `update_counter_and_wait()` and `rcu_gp_ongoing()` routines are identical to those in Figure 10 and are therefore omitted.

Figure 13 shows the signal-handling functions `force_mb_all_threads()` and `sigurcu_handler()`.⁹ Of course, these signals must be used carefully to avoid destroying the readers' wait-free properties, hence the serialization of `synchronize_rcu()`. With simple batching techniques, concurrent invocations of `synchronize_rcu()` could share a single RCU grace period.

The `force_mb_all_threads()` function is invoked from `synchronize_rcu()`. It ensures a memory barrier is executed on each running thread by sending a POSIX signal to all threads and then waiting for each to respond. As shown in Appendix C, this has the effect of promoting compiler-ordering directives such as `barrier()` to full memory barriers, while allowing reader threads to avoid the overhead of memory barriers when they are not needed. An initial iteration over all threads sets each `need_mb` per-thread variable to 1, ensures that this assignment will be seen by the signal handler, and sends a POSIX signal. A second iteration then rescans the threads, waiting until each one has responded by setting its `need_mb` per-thread variable back to zero.¹⁰ Because some versions of some operating systems can lose signals, a production-quality implementation will resend the signal if a response is not received in a timely fashion. Finally, there is a memory barrier to ensure that the signals have been received and acknowledged before later operations that might otherwise destructively interfere with readers.

⁹Some operating systems provide a facility to flush CPU write buffers for all running threads in a given process. Such a facility, where available, can replace the signals.

¹⁰The thread-list scans here and in Figures 8 and 10 are protected by `rcu_gp_lock`. Since the thread-registry list is read-mostly (it is updated only by `rcu_register_thread()` and `rcu_unregister_thread()`), it would appear to be a good candidate for RCU protection. Exercise for the reader: Determine what changes to the implementation would be needed to carry this out.

```

1 void call_rcu(struct rcu_head *head,
2               void (*func)(struct rcu_head *head))
3 {
4     head->func = func;
5     head->next = NULL;
6     enqueue(head, &rcu_data);
7 }
8
9 void call_rcu_cleanup(void)
10 {
11     struct rcu_head *next;
12     struct rcu_head *wait;
13
14     for (;;) {
15         wait = dequeue_all(&rcu_data);
16         if (wait) {
17             synchronize_rcu();
18             while (wait) {
19                 next = wait->next;
20                 wait->func(wait);
21                 wait = next;
22             }
23         }
24         msleep(1);
25     }
26 }

```

Fig. 14. Avoiding Update-Side Blocking by RCU

The signal handler runs in the context of a reader thread in response to the signal sent in line 8. This `sigurcu_handler()` function executes a pair of memory barriers enclosing an assignment of its `need_mb` per-thread variable to zero. The effect is to place a full memory barrier at the point in the receiver's code that was interrupted by the signal, preventing the CPU from reordering memory references across that point.

A proof of correctness for this implementation is the subject of another paper [23]. Of course, as with the other two RCU implementations, this implementation's `synchronize_rcu()` primitive is blocking. The next section shows a way to provide non-blocking RCU updates.

F. Non-Blocking RCU Updates

Although some algorithms use RCU as a first-class technique, RCU is often used only to defer memory reclamation. In these situations, given sufficient memory, `synchronize_rcu()` need not block the update itself, just as automatic garbage collectors need not block non-blocking algorithms. The functions detailed here can be used to perform batched RCU callback execution, allowing multiple callbacks to execute after a grace period has passed.

One way of accomplishing this is shown in Figure 14, which implements the asynchronous `call_rcu()` primitive found in the Linux kernel. The function initializes an RCU callback structure and uses a non-blocking enqueue algorithm [32] to add the callback to the `rcu_data` list. Given that the `call_rcu()` function contains but two simple (and therefore wait-free) assignment statements and an invocation of the non-blocking `enqueue()` function, `call_rcu()` is clearly non-blocking. Systems providing an atomic swap instruction can implement a wait-free `call_rcu()` via the wait-free enqueue algorithm used by some queued locks [33].

A separate thread would remove and invoke these callbacks after a grace period has elapsed, by calling the `call_rcu_cleanup()` shown in Figure 14. On each pass through the

main loop, the function uses a (possibly blocking) dequeue algorithm to remove all elements from the `rcu_data` list en masse. If any elements were present, it waits for a grace period to elapse and then invokes all the RCU callbacks dequeued from the list. Finally, line 24 blocks for a short period to allow additional RCU callbacks to be enqueued. The longer line 24 waits, the more RCU callbacks will accumulate on the `rcu_data` list; this is a classic memory/CPU trade-off, with longer waits allowing more memory to be occupied by RCU callbacks but decreasing the per-callback CPU overhead.

Of course, the use of `synchronize_rcu()` causes `call_rcu_cleanup()` to be blocking, so it should be invoked in a separate thread from the updaters. However, if the synchronization mechanism used to coordinate RCU updates is non-blocking then the updater code paths will execute two non-blocking code sequences in succession (the update and `call_rcu()`), and will therefore themselves be non-blocking.

V. EXPERIMENTAL RESULTS

This section presents benchmarks comparing the RCU mechanisms described in this paper to each other, to pthread mutexes, to pthread reader-writer locks, and to per-thread mutexes. The per-thread mutex approach uses one mutex per reader thread so that updater threads take all the mutexes, always in the same order, to exclude all readers. This approach ensures reader cache locality at the expense of slower write-side locking [34]. Section V-A examines read-side scalability, Section V-B discusses the effect on the read-side primitives of varying the critical-section duration, Section V-C presents the impact of updates on read-side performance, and finally Section V-D compares update-side throughput. The goal is to identify clearly the situations in which RCU outperforms the classic locking solutions found in existing applications.

The machines used to run the benchmarks are an 8-core Intel Core2 Xeon E5405 clocked at 2.0 GHz and a 64-core IBM PowerPC POWER5+ clocked at 1.9 GHz. Each core of the PowerPC machine has 2 hardware threads. To eliminate hardware-thread-level contention for per-core resources, we run our benchmarks using only one hardware thread on each of the 64 cores.

The mutex and reader-writer lock implementations used for comparison are the standard pthread implementations from the GNU C Library 2.7 for 64-bit Intel and GNU C Library 2.5 for 64-bit PowerPC.

STM (Software Transactional Memory) is not included in these comparisons because the jury is still out on STM practicality [35]. STM treats concurrent reads and writes to the same variable as conflicts, requiring frequent conflict checks, in turn degrading reader performance and scalability. In contrast, Figures 20, 21, 22, and 23 will show that RCU's non-conflicting concurrent reads and writes minimize read overhead while maintaining extremely high read scalability, even in the presence of heavy write workloads. Researchers have improved STM's read-side performance and scalability [36], albeit in some cases by placing the burden of instrumentation and privatization on the developer [37]. HTM (Hardware

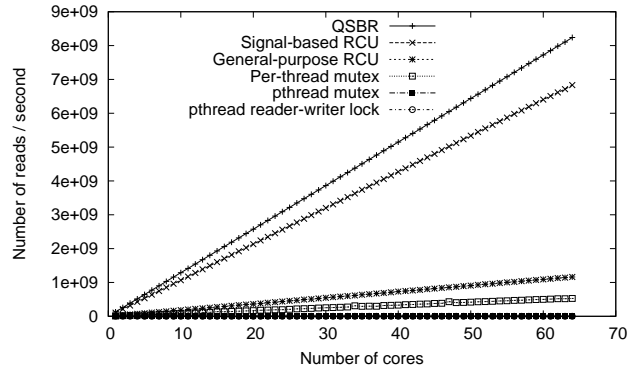


Fig. 15. Read-Side Scalability of Various Synchronization Primitives, 64-core POWER5+

Transactional Memory) [38, 39, 40] is likely to be more scalable than STM; unfortunately, no system supporting HTM was available for this study.

A. Read-Side Scalability

Figure 15 presents a read-side scalability comparison of the RCU mechanisms and the locking primitives on the PowerPC. The goal of this test is to measure each synchronization technique's performance in read-only scenarios, varying the number of CPUs. Each test ran on between 1 and 64 readers for 10 seconds, each taking a read lock, reading one variable, then releasing the lock in a tight loop with no updater. The figure shows that RCU and per-thread mutexes achieve linear scalability, courtesy of the perfect memory locality attained by these approaches. QSRB is fastest, followed by signal-based RCU, general-purpose RCU and per-thread mutex, each adding a constant per-CPU overhead. The Xeon behaves similarly and is not shown here.

Note that the performance of the QSRB and the signal-based-RCU implementations are more than an order of magnitude greater than that of the per-thread mutex. Because the performance of the per-thread mutex corresponds to that of perfect-locality uncontended locking, these two variants of RCU are therefore more than an order of magnitude faster than uncontended locking. Even the slower general-purpose RCU implementation is more than twice as fast as uncontended locking, making use of RCU extremely attractive for read-mostly data structures.

In Figure 15, the traces for pthread mutex and pthread reader-writer locking cannot be easily distinguished from the x axis. Figure 16 therefore displays only these two traces, showing their well-known negative scalability.

B. Read-Side Critical Section Duration

Figure 17 presents the number of reads per second as a function of the duration in nanoseconds of the read-side critical sections. This benchmark is performed with 8 reader threads acquiring the read lock, reading the data structure, busy-waiting for the appropriate delay, and releasing the lock. There is no active updater.

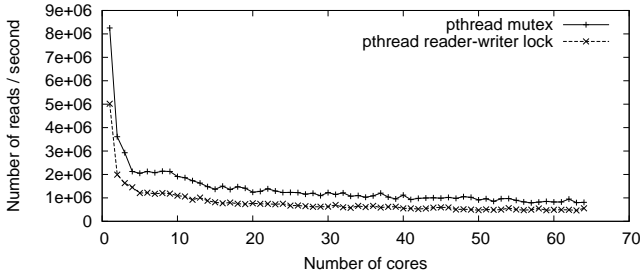


Fig. 16. Read-Side Scalability of Mutex and Reader-Writer Lock, 64-core POWER5+

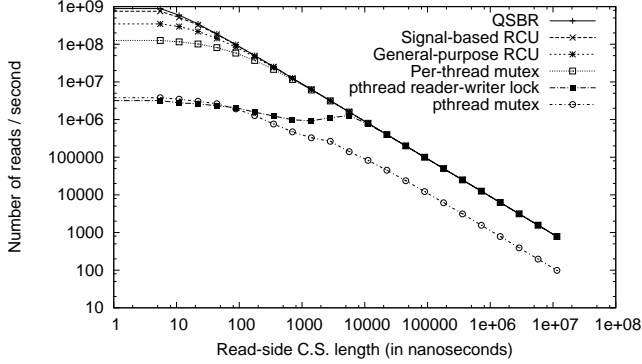


Fig. 17. Impact of Read-Side Critical Section Length on 8-core Xeon, Logarithmic Scale

The number of reads per second is inversely proportional to the sum of the overheads of the read-side primitives and the duration of the read-side critical section. As the critical-section duration increases, the number of reads per second asymptotically approaches the inverse of this duration. The logarithmic axes of Figures 17–19 therefore cause the slopes of the curves to approach -1 . The region where each curve nears its asymptote is closely related to the overhead of the corresponding read-side mechanism.

Thus on the Xeon, QSBR and signal-based RCU have read-side locking overheads at least a factor of 5 better than general-purpose RCU, which in turn is about a factor of 2 better than per-thread mutexes, which in turn is about a factor of 20 better than reader-writer locks (the curves near their asymptotes at 50, 250, 500, and 10,000 nanoseconds respectively). For read-side critical sections longer than 1000 nanoseconds, the difference in overhead between RCU and per-thread mutexes is negligible. The pthread mutex asymptote is lower than the others, because the single mutex can be held by only one reader at a time.

Corresponding curves for the POWER5+ machine appear in Figures 18 and 19. The difference between them is that Figure 19 uses 64 reader threads and 64 cores, whereas Figure 18 uses only 8 threads bound to 8 cores spaced with a stride of 8. Cores close to each other share a common L2 and L3 cache on the POWER5+, which causes reader-writer lock and pthread mutex to be slightly faster at lower stride values (not shown). This has no significant effect on our results.

Comparing Figures 18 and 19 shows that the read-side overheads of both the reader-writer lock and the pthread mutex

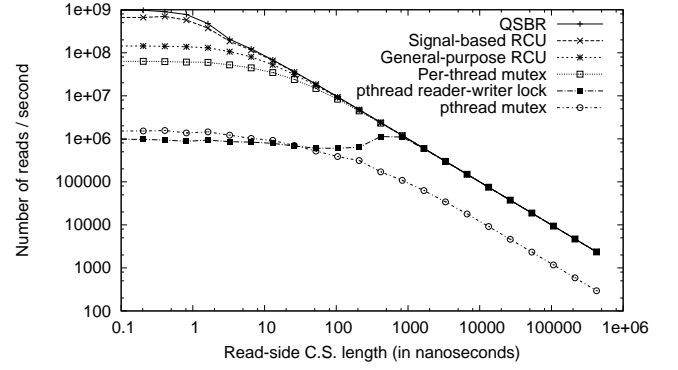


Fig. 18. Impact of Read-Side Critical Section Length, 8 Reader Threads on 64-core POWER5+, Logarithmic Scale

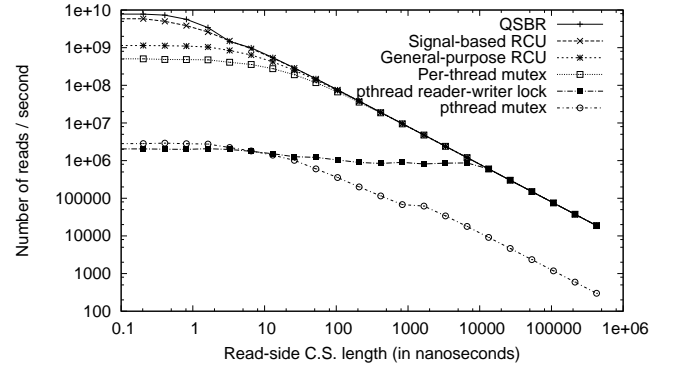


Fig. 19. Impact of Read-Side Critical Section Length, 64 Reader Threads on 64-core POWER5+, Logarithmic Scale

schemes are about 10 times larger when running on 64 cores than on 8 cores (curves near their asymptotes at 10,000 and 2500 nanoseconds instead of 1000 and 250 respectively). This effect is caused by interprocessor cache-line-exchange delays and nonlinear scaling of lock-contention times. By contrast, the read-side overheads of the RCU and per-thread mutex schemes are independent of the number of CPUs, and on this machine, the difference in overhead between these schemes is negligible for critical sections longer than 250 nanoseconds.

Two interesting features of the pthread reader-writer lock trace in Figures 17, 18, and 19 deserve explanation. The first is that the performance of pthread reader-writer locking is inferior to that of pthread mutex for small read-side critical-section lengths, which is due to the slightly higher overhead of reader-writer locking compared to that of pthread mutex's exclusive locking. The second is the slight rise in throughput for reader-writer locking just prior to joining the asymptote, which is due to decreased memory contention on the data structure implementing the reader-writer lock.

C. Effects of Updates on Read-Side Performance

The results in Sections V-A and V-B clearly show RCU's read-side performance advantages. However, RCU updates can incur performance penalties due to the overhead of grace periods and the resulting decreases in locality of reference. This section therefore measures these performance penalties.

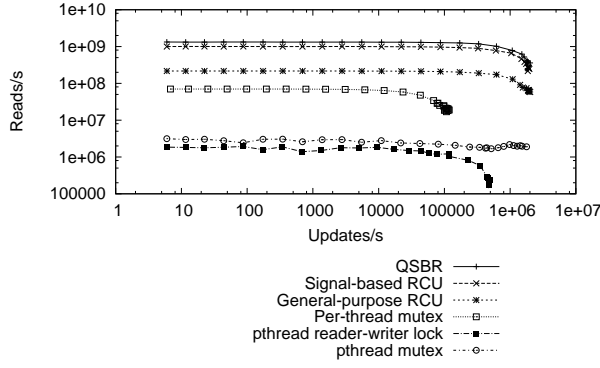


Fig. 20. Update Overhead, 8-core Intel Xeon, Logarithmic Scale

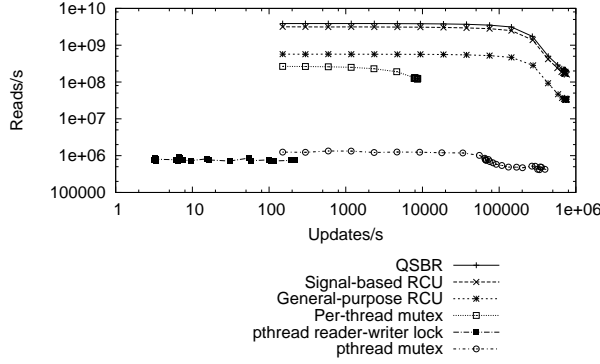


Fig. 21. Update Overhead, 64-core POWER5+, Logarithmic Scale

Figure 20 presents the impact of update frequency on read-side performance for the various locking primitives on the Intel Xeon. It is performed by running 4 reader and 4 updater threads and varying the delay between updates. The updaters for the per-thread mutex, mutex and reader-writer lock experiments store two different integer values successively to the same variable. Readers accessing the variable twice while holding a lock are guaranteed to observe a single, unchanged value. To provide the same effect, the RCU updaters allocate a new structure, store an integer in this newly allocated structure, and then atomically exchange the pointer to the new structure with the old pointer currently being accessed by readers. The RCU experiments store only a single integer value in each structure; we verified that successively storing two values to the same memory location had no significant impact on performance. Memory reclamation is batched using an `rcu_defer()` mechanism; this mechanism uses fixed-size per-thread queues to hold memory reclamation requests so that an updater incurs a grace period no more than once every 4096 updates. A grace period is of course required whenever an updater finds its queue is full. In addition, a separate worker thread empties the queues every 100 milliseconds to provide an upper bound for reclamation delay. Figure 21 shows the result of this same benchmark running on a 64-core POWER5+, with 32 reader and 32 updater threads.

Interestingly, on such a workload with 4 tight-loop readers, mutexes uniformly outperform reader-writer locking. Furthermore, this particular implementation of reader-writer locking

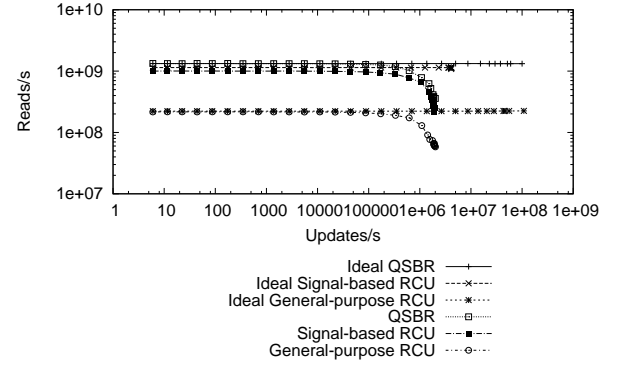


Fig. 22. Comparison of Pointer Exchange and Ideal RCU Update Overhead, 8-core Intel Xeon, Logarithmic Scale

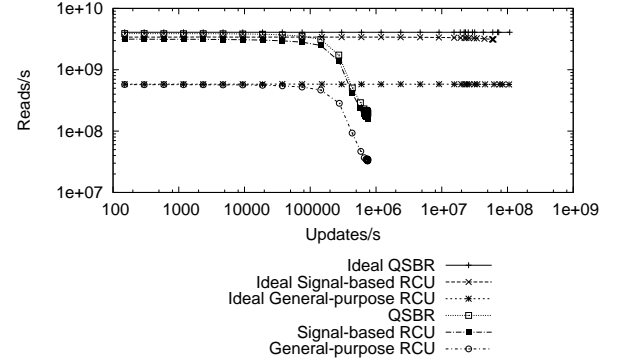


Fig. 23. Comparison of Pointer Exchange and Ideal RCU Update Overhead, 64-core POWER5+, Logarithmic Scale

eventually suffers from reader starvation.

The RCU read-side performance shown in Figures 20 and 21 trails off at high update rates. In principle this could be caused either by the overhead of quiescent-state detection on the write side or by cache interference resulting from the data pointer exchanges. To determine the cause, we defined ideal RCU performance to include only the overheads of the grace periods, and compared this ideal performance to that shown in the earlier figures. We generated the ideal RCU workload by removing the memory allocation and pointer exchanges from the update-side, but we kept the `rcu_defer()` mechanism in order to take into account the overhead of waiting for quiescent states. Figures 22 and 23 present the resulting comparison, clearly showing that the non-linear read-side performance is caused by the pointer exchanges rather than the grace periods.

Figures 24 and 25 present the impact of the update-side critical-section length on read-side performance. These tests are performed with 4 reader and 4 writer threads on the Xeon, and with 32 reader and 32 writer threads on the POWER5+. Readers run as quickly as possible, with no delay between reads. Writer iterations are separated by an arbitrarily-sized delay consisting of 10 iterations of a busy loop, amounting to 55 nanoseconds for the Intel Xeon (due to the “`rep; nop`” instruction recommended for x86 busy-waiting loops) and 2.0 nanoseconds for the POWER5+.

With RCU approaches, the read-side performance is largely unaffected by updates. Slight variations can be seen on a linear

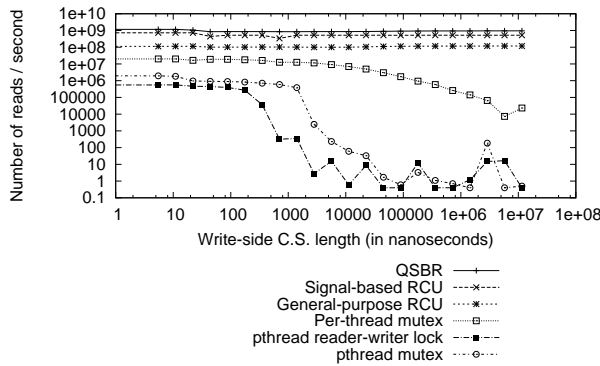


Fig. 24. Impact of Update-Side Critical Section Length on Read-Side, 8-core Intel Xeon, Logarithmic Scale

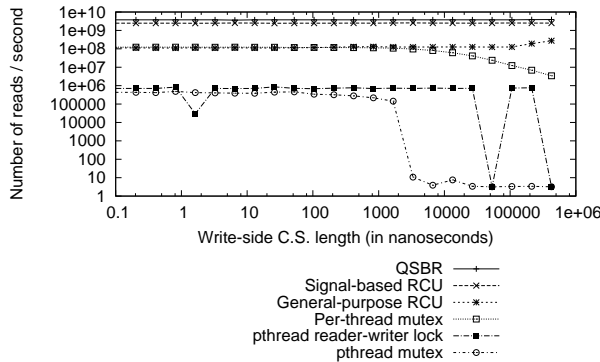


Fig. 25. Impact of Update-Side Critical Section Length on Read-Side, 64-core POWER5+, Logarithmic Scale

scale (not shown here), but these are caused primarily by CPU affinity of readers and writers, which influences the sharing of caches.

Unlike RCU, per-thread mutex readers are significantly impacted by long write-side critical sections. Again referring to Figures 24 and 25, read-side performance degrades significantly beyond a write-side critical-section length of 5,000 nanoseconds on both the Xeon and the POWER5+. On the Xeon, the pthread reader-writer lock and pthread mutex degrade catastrophically starting at 250 to 750 nanoseconds write-side critical-section length. In addition, these schemes show signs of starvation in the presence of long write-side critical sections. We saw instances of both reader starvation (the dips in Figure 25) and writer starvation (not shown); apparently the class which owns the lock first (either readers or writers) tends to keep it for the whole test duration. This is likely caused by the brevity of the delays between reads and updates, which favors the previous lock owner due to unfairness in the pthread implementations.

D. Update Throughput

Maximum update rates can be inferred from the X-axis of Figures 20 and 21 by selecting the rightmost point of a given trace. For example, Figure 20 shows that RCU attains 2 million updates per second, while per-thread locks manages but 0.1 million updates per second. A key reason for this result is that

RCU readers do not block RCU writers. Furthermore, although waiting for an RCU grace period can incur significant latency, it does not necessarily degrade updater bandwidth because in production-quality implementations, RCU grace periods can overlap in time.

In Figure 21, the mutex-based benchmark performance starts degrading at 30,000 updates per second with 32 updater threads, while RCU easily exceeds 100,000 updates per second. These results clearly show the need to partition data in order to attain good performance on larger systems. Benchmarks running only 4 updater threads on the 64-core system show similar effects (data not presented). Figure 20 shows that update overhead remains reasonably constant even at higher update frequency for 4 updater threads on the Xeon. Therefore, as the number of concurrent updaters increases, mutex behavior seems to depend on the architecture and on the specific GNU C Library version.

In Figure 21, the reader-writer lock attains only 175 updates per second, indicating that updaters are starved by readers. Per-thread locks attain only 10,000 updates per second. Thus, locking significantly limits update rate relative to RCU.

These results show that RCU QSBR and general-purpose RCU attain the highest update rates for partitionable read-mostly data structures (where “read mostly” means more than 90% of accesses are reads) even compared to uncontended locking. This is attributed to the lower performance overhead for exchanging a pointer compared to the multiple atomic operations and memory barriers implied by acquiring and releasing a lock. RCU is sometimes used even for update-heavy workloads, due to the wait-free and deadlock-immune properties of its read-side primitives. The performance characteristics of RCU for update-heavy workloads have been presented elsewhere [41].

VI. CONCLUSIONS

We have presented a set of RCU implementations covering a wide spectrum of application architectures. QSBR shows the best performance characteristics, but severely constrains the application architecture by requiring that each reader thread periodically announce that it is in a quiescent state. User-level QSBR imposes two additional requirements: (1) any lock that is held across a grace period must be acquired within an extended quiescent state, and (2) if the handler for a given signal contains RCU read-side critical sections, then that signal must be disabled across all extended quiescent states. Interestingly enough, kernel-level QSBR implementations avoid these two requirements because of QSBR’s integration with the scheduler and interrupt handlers.

Signal-based RCU performs almost as well as QSBR, but requires reserving a POSIX signal. Unlike the other two, general-purpose RCU incurs significant read-side overhead. However it minimizes constraints on application architecture, requiring only that each thread invokes an initialization function before entering its first RCU read-side critical section.

Benchmarks demonstrate linear read-side scalability of RCU and per-thread locking. They also show that there is a read-side critical-section duration beyond which reader-writer locking, RCU, and per-thread locking perform similarly,

and that this duration increases with the number of cores. These benchmarks also show that by performing grace-period detection in batch, RCU attains better update rates than reader-writer locking, per-thread locking, and exclusive locking on read-mostly data structures. It is possible to further decrease RCU update-side overhead by designing data structures providing good update cache locality.

ACKNOWLEDGEMENTS

We owe thanks to Maged Michael, Etienne Bergeron, Alexandre Desnoyers, Michael Stumm, Balaji Rao, Tom Hart, Robert Bauer, Dmitriy V'jukov, and the anonymous reviewers for many helpful suggestions. We are indebted to the Linux community for their use of and contributions to RCU and to Linus Torvalds for sharing his kernel with us all. We are grateful to Kathy Bennett for her support of this effort.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851. This work is funded by Google, Natural Sciences and Engineering Research Council of Canada, Ericsson and Defence Research and Development Canada.

LEGAL STATEMENT

This work represents the views of the authors and does not necessarily represent the view of EfficiOS, Ecole Polytechnique de Montreal, Harvard, IBM, or Portland State University.

Linux is a registered trademark of Linus Torvalds.

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

REFERENCES

- [1] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "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.
- [2] J. P. Hennessy, D. L. Osisek, and J. W. Seigh II, "Passive serialization in a multitasking environment," US Patent and Trademark Office, Washington, DC, Tech. Rep. US Patent 4,809,168 (lapsed), February 1989.
- [3] V. Jacobson, "Avoid read-side locking via delayed free," September 1993, private communication.
- [4] A. John, "Dynamic vnodes – design and implementation," in *USENIX Winter 1995*. New Orleans, LA: USENIX Association, January 1995, pp. 11–23.
- [5] P. E. McKenney and J. D. Slingwine, "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.
- [6] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of Linux scalability to many cores," in *9th USENIX Symposium on Operating System Design and Implementation*. Vancouver, BC, Canada: USENIX, October 2010, pp. 1–16.
- [7] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *J. Parallel Distrib. Comput.*, vol. 67, no. 12, pp. 1270–1285, 2007.
- [8] K. A. Fraser, "Practical lock-freedom," Ph.D. dissertation, King's College, University of Cambridge, 2003.
- [9] K. Fraser and T. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, pp. 1–61, 2007.
- [10] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit, "A lazy concurrent list-based set algorithm," in *Principles of Distributed Systems, 9th International Conference OPODIS 2005*. Springer-Verlag, 2005, pp. 3–16.
- [11] H. T. Kung and Q. Lehman, "Concurrent maintenance of binary search trees," *ACM Transactions on Database Systems*, vol. 5, no. 3, pp. 354–382, September 1980.
- [12] P. Becker, "Working draft, standard for programming language C++," August 2010, [Online]. Available: <http://open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3126.pdf>.
- [13] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole, "The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux," *IBM Systems Journal*, vol. 47, no. 2, pp. 221–236, May 2008.
- [14] P. E. McKenney and J. Walpole. (2007, December) What is RCU, fundamentally? [Online]. Available: Linux Weekly News, <http://lwn.net/Articles/262464/>.
- [15] M. Herlihy, "Implementing highly concurrent data objects," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 5, pp. 745–770, Nov. 1993.
- [16] R. K. Treiber, "Systems programming: Coping with parallelism," April 1986, RJ 5118.
- [17] D. Sarma and P. E. McKenney, "Making RCU safe for deep sub-millisecond response realtime applications," in *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)*. USENIX Association, June 2004, pp. 182–191.
- [18] P. E. McKenney. (2008, January) What is RCU? part 2: Usage. [Online]. Available: Linux Weekly News, <http://lwn.net/Articles/263130/>.
- [19] —, "Exploiting deferred destruction: An analysis of read-copy-update techniques in operating system kernels," Ph.D. dissertation, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004, [Online]. Available: <http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf>.
- [20] P. E. McKenney and D. Sarma, "Towards hard real-time response from the Linux kernel on SMP hardware," in *linux.conf.au 2005*, Canberra, Australia, April 2005, [Online]. Available: <http://www.rdrop.com/users/paulmck/RCU/realtimeRCU.2005.04.23a.pdf>.
- [21] A. Arcangeli, M. Cao, P. E. McKenney, and D. Sarma, "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)*. USENIX Association, June 2003, pp. 297–310.

- [22] M. Greenwald and D. R. Cheriton, "The synergy between non-blocking synchronization and operating system structure," in *Proceedings of the Second Symposium on Operating Systems Design and Implementation*. Seattle, WA: USENIX Association, Oct. 1996, pp. 123–136.
- [23] M. Desnoyers, "Low-impact operating system tracing," Ph.D. dissertation, Ecole Polytechnique de Montréal, December 2009, [Online]. Available: <http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf>.
- [24] P.-M. Fournier, M. Desnoyers, and M. R. Dagenais, "Combined tracing of the kernel and applications with LTTng," in *Proceedings of the 2009 Linux Symposium*, Jul. 2009.
- [25] T. Jinmei and P. Vixie, "Implementation and evaluation of moderate parallelism in the BIND9 DNS server," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, Boston, MA, February 2006, pp. 115–128.
- [26] *Programming languages — C*, ISO WG14 Std., May 2005, [Online]. Available: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf>.
- [27] *Guide to Parallel Programming*, Sequent Computer Systems, Inc., 1988.
- [28] M. Desnoyers, "[RFC git tree] userspace RCU (urcu) for Linux," February 2009, [Online]. Available: <http://lkml.org/lkml/2009/2/5/572>, <http://lttng.org/urcu>.
- [29] M. Herlihy, "Wait-free synchronization," *ACM TOPLAS*, vol. 13, no. 1, pp. 124–149, January 1991.
- [30] P. E. McKenney, "Using a malicious user-level RCU to torture RCU-based algorithms," in *linux.conf.au 2009*, Hobart, Australia, January 2009, [Online]. Available: <http://www.rdrop.com/users/paulmck/RCU/urcutorture.2009.01.22a.pdf>.
- [31] —. (2007, October) The design of preemptible read-copy-update. [Online]. Available: Linux Weekly News, <http://lwn.net/Articles/253651/>.
- [32] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 51, no. 1, pp. 1–26, 1998.
- [33] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *Transactions of Computer Systems*, vol. 9, no. 1, pp. 21–65, February 1991.
- [34] W. C. Hsieh and W. E. Weihl, "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.
- [35] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?" *ACM Queue*, September 2008.
- [36] L. Dalessandro, M. F. Spear, and M. L. Scott, "Norec: streamlining STM by abolishing ownership records," in *PPOPP*, 2010, pp. 67–78.
- [37] A. Dragovejic, P. Felber, V. Gramoli, and R. Guerraoui, "Why STM can be more than a research toy," February 2010, [Online]. Available: <http://infoscience.epfl.ch/record/144052/files/paper.pdf>.
- [38] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A scalable, non-blocking approach to transactional memory," in *HPCA Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 97–108.
- [39] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian, "Scalable and reliable communication for hardware transactional memory," in *PACT Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 144–154.
- [40] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, Washington, DC, USA, March 2009, pp. 157–168.
- [41] P. E. McKenney, "RCU vs. locking performance on different CPUs," in *linux.conf.au*, Adelaide, Australia, January 2004, [Online]. Available: <http://www.rdrop.com/users/paulmck/RCU/lockperf.2004.01.17a.pdf>.

Mathieu Desnoyers is President & Founder of EfficiOS. He maintains the LTTng project and the Userspace RCU library. His research interests are in performance analysis tools, operating systems, scalability and real-time concerns. He holds a Ph.D. degree in Computer Engineering from Ecole Polytechnique de Montreal (2010).



Paul E. McKenney is an Distinguished Engineer at IBM. He maintains the Linux-kernel RCU implementations, and his primary research interest is shared-memory parallel software. He holds a Ph.D. in computer science and engineering from Oregon Health and Sciences University (2004).



Alan S. Stern received a Ph.D. in Mathematical Logic from the University of California at Berkeley in 1984. His current position at the Rowland Institute at Harvard is Staff Computational Scientist. He is actively involved with Linux kernel development, particularly in the USB and Power Management subsystems.



Michel R. Dagenais is professor at Ecole Polytechnique de Montreal, in the Computer and Software Engineering Department. His research interests include several aspects of multi-core distributed systems with emphasis on Linux and open systems. His group has made several original contributions to Linux.



Jonathan Walpole is a Full Professor in the Computer Science Department at Portland State University. His research interests are in operating systems, and scalable concurrent programming. He holds B.Sc. and Ph.D. degrees in Computer Science from Lancaster University, UK (1984 and 1987).



APPENDIX

A. Quiescent-State-Based Reclamation RCU

This appendix presents a semi-formal verification that the QSBR implementation satisfies RCU's grace-period guarantee. Unfortunately it is not possible to prove this outright, owing to the 32-bit wraparound failure discussed in Section IV-C. We will therefore *assume* that such failures don't occur.

Theorem: Assuming that threads are never preempted for too long (see below), in any execution of a program using the QSBR implementation, the grace-period guarantee (Formula 1 in Section II-D1) is satisfied.

Proof: Given QSBR's implementation, we focus on *active segments*: the periods between quiescent states. More precisely, an active segment is a sequence of instructions bounded at the start by `rcu_quiescent_state()`, `rcu_thread_online()`, or lines 12–15 of `synchronize_rcu()` in Figure 8; bounded at the end by `rcu_quiescent_state()`, `rcu_thread_offline()`, or lines 5–8 of `synchronize_rcu()` in Figure 8; and containing no other references to `rcu_reader.ctr`. Every read-side critical section is part of an active segment.

Execution of the k th active segment in thread T (comprising statements $R_{k,i}$ together with some of the bounding statements) can be represented as a sequence of labelled pseudo-code instructions:

```

Ld( $x_k$ ):   $x_k = \text{rcu\_gp\_ctr}$ 
St( $x_k$ ):   $\text{rcu\_reader.ctr}_T = x_k$ 
MB $_k^0$ :    smp_mb()
            $R_{k,0}; R_{k,1}; R_{k,2}; \dots$ 
MB $_k^1$ :    smp_mb()
Ld( $y_k$ ):   $y_k = (\text{either } \text{rcu\_gp\_ctr} \text{ or } 0)$ 
St( $y_k$ ):   $\text{rcu\_reader.ctr}_T = y_k$ 

```

Here x_k and y_k are intermediate values and `rcu_reader.ctrT` refers to T 's instance of the per-thread `rcu_reader` structure. The `Ld(y_k)` line sets y_k to `rcu_gp_ctr` if the active segment ends with `rcu_quiescent_state()` (in which case y_k is x_{k+1} , as the call will also mark the beginning of the next active segment); otherwise it sets y_k to 0.

Execution of `update_counter_and_wait()` is of course serialized by the `rcu_gp_lock` mutex. The n th occurrence of this routine, together with the statements preceding ($M_{n,i}$) and following it ($D_{n,j}$), can be expressed so:

```

            $M_{n,0}; M_{n,1}; M_{n,2}; \dots$ 
MB $_n^2$ :    smp_mb()
           ...
Ld( $z_n$ ):   $z_n = \text{rcu\_gp\_ctr} + 2$ 
St( $z_n$ ):   $\text{rcu\_gp\_ctr} = z_n$ 
           ...
Ld( $v_n$ ):   $v_n = \text{rcu\_reader.ctr}_T$ 
As $_n$ :    assert( $v_n = 0$  or  $v_n = z_n$ )
           ...
MB $_n^3$ :    smp_mb()
            $D_{n,0}; D_{n,1}; D_{n,2}; \dots$ 

```

Several things are omitted from this summary, including the `list_for_each_entry()` loop iterations for threads

other than T and all iterations of the while loop other than the last (the assertion As_n holds precisely because this is the last loop iteration). Both here and above, the use of `barrier()`, `LOAD_SHARED()`, and `STORE_SHARED()` primitives forces the compiler to generate the instructions in the order shown. However the hardware is free to reorder them, within the limits imposed by the memory barriers.

We number the grace periods starting from 1, letting `St(z_0)` stand for a fictitious instruction initializing `rcu_gp_ctr` to 1 before the program starts. In this way each `Ld(x_k)` is preceded by some `St(z_n)`. Furthermore, because there are no other assignments to `rcu_gp_ctr`, it is easy to see that for each n , z_n is equal to $2n + 1$ truncated to the number of bits in an unsigned long.

Our initial assumption regarding overly-long preemption now amounts to the requirement that not too many grace periods occur between each `Ld(x_k)` and the following `St(x_k)`. Grace periods m through $n - 1$ occur during this interval when `Ld(x_k)` \rightarrow `St(z_m)` and `Ld(v_{n-1})` \rightarrow `St(x_k)` (recall that " \rightarrow " indicates that the statement on the left executes prior to that on the right). Under such conditions we therefore require $n - m$ to be sufficiently small that $2(n - m + 1)$ does not overflow an unsigned long and hence $z_{m-1} \neq z_n$.

Let us now verify the grace-period guarantee for a read-side critical section occurring during thread T 's active segment k and for grace period n .

Case 1: `St(z_n)` \rightarrow `Ld(x_k)`. Since these two instructions both access `rcu_gp_ctr`, the memory barrier property for `MB $_n^2$` and `MB $_k^0$` says that $M_{n,i} \rightarrow R_{k,j}$ for each i, j . Thus Formula 1 in Section II-D1 holds because its left disjunct is true.

Case 2: `Ld(v_n)` \rightarrow `St(x_k)`. Since these two instructions both access `rcu_reader.ctrT`, the memory barrier property for `MB $_n^2$` and `MB $_k^0$` says that $M_{n,i} \rightarrow R_{k,j}$ for each i, j . The rest follows as in Case 1.

Case 3: `St(y_k)` \rightarrow `Ld(v_n)`. Since these two instructions both access `rcu_reader.ctrT`, the memory barrier property for `MB $_k^1$` and `MB $_n^3$` says that $R_{k,i} \rightarrow D_{n,j}$ for each i, j . Thus Formula 1 holds because its right disjunct is true.

Case 4: None of Cases 1–3 above. Since cache coherence guarantees that loads and stores from all threads to a given variable are totally ordered, it follows that `Ld(x_k)` \rightarrow `St(z_n)` and `St(x_k)` \rightarrow `Ld(v_n)` \rightarrow `St(y_k)`. We claim that this case cannot occur without violating our assumption about wraparound failures. Indeed, suppose it does occur, and take m to be the least index for which `Ld(x_k)` \rightarrow `St(z_m)`. Clearly $m \leq n$.

Since `St(z_{m-1})` \rightarrow `Ld(x_k)` \rightarrow `St(z_m)`, x_k must be equal to z_{m-1} . Similarly, v_n must be equal to x_k . Since z_{m-1} cannot be 0, As_n implies that $z_{m-1} = z_n$, and since $m \leq n$, this is possible only if $2(n - m + 1)$ overflows (and hence $n > m$).

Now consider grace period $n - 1$. Since `St(z_m)` precedes `St(z_{n-1})` we have `Ld(x_k)` \rightarrow `St(z_{n-1})`, and since `Ld(v_{n-1})` precedes `Ld(v_n)` we also have `Ld(v_{n-1})` \rightarrow `St(y_k)`. If `St(x_k)` \rightarrow `Ld(v_{n-1})` then active segment k and grace period $n - 1$ would also fall under Case 4, implying that $z_{m-1} = z_{n-1}$, which is impossible because $z_{n-1} \neq z_n$. Hence we must have `Ld(v_{n-1})` \rightarrow `St(x_k)`. But then m and n would violate our requirement on the number of grace periods elapsing between `Ld(x_k)` and `St(x_k)`. QED.

B. General-Purpose RCU

This appendix presents a semi-formal verification that the general-purpose implementation satisfies RCU's grace-period guarantee, assuming that read-side critical sections are not nested too deeply.

Proof: Using the same notation as in Appendix A, execution of the k th outermost read-side critical section in thread T can be represented as follows:

```

       $x_{k,0} = \text{rcu\_gp\_ctr}$ 
St( $x_{k,0}$ ):  $\text{rcu\_reader.ctr}_T = x_{k,0}$ 
MB $_k^0$ :     $\text{smp\_mb}()$ 
{
   $R_{k,0}; R_{k,1}; R_{k,2}; \dots$ 
St( $x_{k,i}$ ):  $\text{rcu\_reader.ctr}_T = x_{k,i} \dots$ 
MB $_k^1$ :     $\text{smp\_mb}()$ 
       $y_k = \text{rcu\_reader.ctr}_T -$ 
           $\text{RCU\_NEST\_COUNT}$ 
St( $y_k$ ):   $\text{rcu\_reader.ctr}_T = y_k$ 

```

Here $x_{k,0}$ is the value read by `LOAD_SHARED()` in the outermost `rcu_read_lock()`, $x_{k,i}$ for $i > 0$ corresponds to the i^{th} nested call by thread T to `rcu_read_lock()` or `rcu_read_unlock()` in time order (the “{...}” notation is intended to express that these calls are interspersed among the $R_{k,i}$ statements), and y_k is the value in the `STORE_SHARED()` call in the `rcu_read_unlock()` that ends the critical section. The memory barriers are present because this is an outermost read-side critical section.

The n th occurrence of `synchronize_rcu()`, together with the statements preceding and following it, can similarly be expressed as:

```

       $M_{n,0}; M_{n,1}; M_{n,2}; \dots$ 
MB $_n^2$ :     $\text{smp\_mb}()$ 
...
Tog $_n^0$ :    $\text{rcu\_gp\_ctr} \hat{=} \text{RCU\_GP\_CTR\_PHASE}$ 
...
Ld( $v_n^0$ ):  $v_n^0 = \text{rcu\_reader.ctr}_T$ 
As $_n^0$ :     $\text{assert}(v_n^0\text{'s nesting level is 0 or its}$ 
           $\text{phase number agrees with rcu\_gp\_ctr})$ 
...
Tog $_n^1$ :    $\text{rcu\_gp\_ctr} \hat{=} \text{RCU\_GP\_CTR\_PHASE}$ 
...
Ld( $v_n^1$ ):  $v_n^1 = \text{rcu\_reader.ctr}_T$ 
As $_n^1$ :     $\text{assert}(v_n^1\text{'s nesting level is 0 or its}$ 
           $\text{phase number agrees with rcu\_gp\_ctr})$ 
...
MB $_n^3$ :     $\text{smp\_mb}()$ 
           $D_{n,0}; D_{n,1}; D_{n,2}; \dots$ 

```

As before, the assertions hold because these statements are from the last iteration of the while loop for thread T . We can now verify the grace-period guarantee for T 's read-side critical section k and for grace period n .

Case 1: $\text{Ld}(v_n^m) \rightarrow \text{St}(x_{k,0})$, $m = 0$ or 1. Since these instructions all access `rcu_reader.ctrT`, the memory barrier property for MB_n^2 and MB_k^0 says that $M_{n,i} \rightarrow R_{k,j}$ for each i, j . Thus Formula 1 in Section II-D1 holds because its left disjunct is true.

Case 2: $\text{St}(y_k) \rightarrow \text{Ld}(v_n^m)$, $m = 0$ or 1. Then the memory barrier property for MB_k^1 and MB_n^3 says that $R_{k,i} \rightarrow D_{n,j}$ for

each i, j . Thus Formula 1 holds because its right disjunct is true.

Case 3: Neither of Cases 1–2 above. For each m we must have $\text{St}(x_{k,0}) \rightarrow \text{Ld}(v_n^m) \rightarrow \text{St}(y_k)$; therefore v_n^0 and v_n^1 must be equal to $x_{k,i}$ for some values of $i \geq 0$. We are assuming that the maximum nesting level of read-side critical sections does not exceed the 16-bit capacity of `RCU_NEST_MASK`; therefore each $x_{k,i}$ has a nonzero nesting level and has the same phase number as $x_{k,0}$, and the same must be true of v_n^0 and v_n^1 . However the phase number of `rcu_gp_ctr` is different in As_n^0 and As_n^1 , thanks to Tog_n^1 . Hence As_n^0 and As_n^1 cannot both hold, implying that this case can never arise. QED.

C. Barrier Promotion Using Signals

This appendix discusses how the signal-based RCU implementation is able to “promote” compiler barriers to full-fledged memory barriers. A more accurate, if less dramatic, statement is that the combination of `barrier()` and `force_mb_all_threads()` obeys a strong form of the fundamental ordering property of memory barriers. Namely, if one thread executes the statements

$A_0; A_1; A_2; \dots; \text{barrier}(); B_0; B_1; B_2; \dots;$

and another thread executes the statements

$C_0; C_1; C_2; \dots; \text{force_mb_all_threads}();$
 $D_0; D_1; D_2; \dots;$

then either $A_i \rightarrow D_j$ (all i, j) or $C_i \rightarrow B_j$ (all i, j). To see why, consider that the `pthread_kill()` call in `force_mb_all_threads()` forces the first thread to invoke `sigrcu_handler()` at some place in its instruction stream, either before all the B_j or after all the A_i (although `barrier()` does not generate any executable code, it does force the compiler to emit all the object code for the A_i instructions before any of the B_j object code). Suppose `sigrcu_handler()` is called after all the A_i . Then the first thread actually executes

$A_0; A_1; A_2; \dots; \text{smp_mb}();$
 $\text{STORE_SHARED}(\text{rcu_reader.need_mb}, 0);$

Since the second thread executes

$\text{LOAD_SHARED}(\text{index} \rightarrow \text{need_mb}); \dots;$
 $\text{smp_mb}(); D_0; D_1; D_2; \dots;$

in its last loop iteration for the first thread and the following statements, and since the `LOAD_SHARED()` sees the value stored by the `STORE_SHARED()`, it follows that $A_i \rightarrow D_j$ for all i, j . The case where `sigrcu_handler()` runs before all the B_j can be analyzed similarly.