User-Level Implementations of Read-Copy Update

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

***Abstract*—Read-copy update (RCU) is a synchronization primitive that is often used as a replacement for reader-writer locking, due to the fact that it provides extremely lightweight read-side primitives with sharply bounded execution times. RCU updates are typically much heavier weight than are RCU readers, especially when used in conjunction with locking.**

Although RCU is heavily used in a number of kernel-level environments, these implementations make use of interrupt- and preemption-disabling facilities that are often unavailable to user-level applications. The few RCU implementations that are available to user applications either provide inefficient read-side primitives or restrict application architecture.

This paper describes several classes of efficient RCU implementations that are based on primitives commonly available to user-level applications.

Finally, performance comparison of these RCU primitives with each other and to standard locking leads to discuss appropriate locking for various workloads. This opens the door to use of RCU outside of kernels.

***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 mechanism that was added to the Linux kernel in October of 2002. RCU achieves scalability improvements by allowing reads to occur concurrently with updates. In contrast with conventional locking primitives that ensure mutual exclusion among concurrent threads regardless of whether they be readers or updaters, or with reader-writer locks that allow concurrent reads but not in the presence of updates, RCU supports concurrency between a single updater and multiple readers. RCU ensures that reads are coherent by maintaining multiple versions of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete. RCU defines and uses efficient and scalable mechanisms for publishing and reading new versions of an object, and also for deferring reclamation of old versions. These mechanisms distribute the work among read and update paths in such a way as to make read paths extremely fast. In some cases, as will be presented in Section IV-B, RCU's read-side primitives have zero overhead.

Manuscript received July X, 2009; revised Month Y, 2009

M. Desnoyers (mathieu.desnoyers@polymtl.ca) and M. R. Dagenais (michel.dagenais@polymtl.ca) are with the Computer and Software Engineering Department, Ecole Polytechnique de Montreal.

Paul E. McKenney (paulmck@linux.vnet.ibm.com) is with the IBM Linux Technology Center.

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

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

Although mechanisms similar to RCU have been used in a number of operating-system kernels (1; 2; 3; 4; 5), and, as shown in Figure 1, is heavily used in the Linux kernel, we are not aware of significant application usage. This lack of application-level use is in part due to the fact that prior user-level RCU implementations imposed global constraints on the application's structure and operation (6), and in some cases heavy read-side overhead as well (7). The popularity of RCU in operating-system kernels has been in part due to the fact that these can accommodate the required global constraints imposed by earlier RCU implementations. Kernels therefore permits use of the high-performance quiescent-state based reclamation (QSBR) class of RCU implementations. In fact, in server-class (CONFIG_PREEMPT=n) Linux-kernel builds, RCU incurs zero read-side overhead (8).

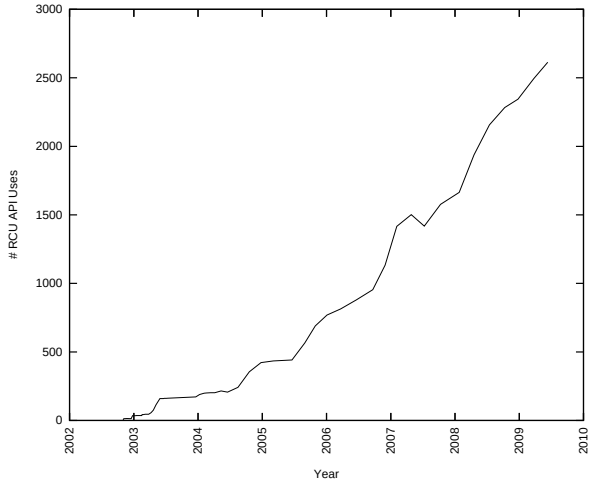

Fig. 1. Linux-Kernel Usage of RCU

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

First, Section II 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. Finally, Section V presents experimental results, comparing RCU solutions to each other and to standard locks. This leads to recommendations on locking use for various workloads presented in Section VI.

II. Brief Overview of RCU

This section introduces a conceptual view covering most RCU-based algorithms in Section II-A to familiarise the reader

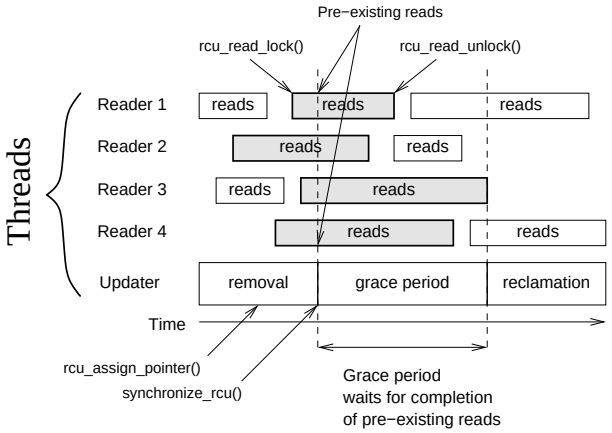

Fig. 2. Schematic of RCU Grace Period and Read-Side Critical Sections

with RCU concepts and vocabulary. It then presents an informal RCU desiderata in Section II-B, which details the goals pursued in this work. Then, Section II-C shows how RCU is used to delete an element from a linked list in the face of concurrent readers. Finally, Section II-D gives an overview of RCU semantics, presenting the synchronization guarantees provided by RCU.

A. Conceptual View of RCU Algorithms

A schematic for the high-level structure of an RCU-based algorithm is shown in Figure 2, which can be thought of as a pictorial view of (1) presented in Section II-D1. The grace period concept, explained thoroughly in section II-D1, can be defined informally for the needs of this section as a period 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.

Here, each box labeled "Reads" is an RCU read-side critical section that begins with rcu_read_lock() and ends with rcu_read_unlock(). Each row of RCU read-side critical sections denotes a separate thread, for a total of four read-side threads. The two boxes at the bottom left and right of the figure denote a fifth thread, this one performing an RCU update.

This RCU update is split into two phases, a removal phase denoted by the lower left-hand box and a reclamation phase denoted by the lower right-hand box. These two phases must be separated by a grace period, which is determined by the duration of the synchronize_rcu() execution. During the removal phase, the RCU update removes elements from the data structure (possibly inserting some as well) by issuing an rcu_assign_pointer() or equivalent pointer-replacement primitive. These removed data elements will not be accessible to RCU read-side critical sections starting after the removal phase ends, but might still be accessed by RCU read-side critical sections initiated during the removal phase. However, by the end of the RCU grace period, all of the RCU read-side critical sections that might be accessing the newly removed data elements are guaranteed to have completed, courtesy of the definition of "grace period". Therefore, the reclamation phase beginning after the grace period ends can safely free the data elements removed previously.

B. User-Space RCU Desiderata

Extensive use of RCU applications has lead to the following user-space RCU desiderata:

1) Read-side primitives (such as rcu_read_lock() and rcu_read_unlock()) bounding RCU read-side critical sections and grace-period primitives (such as synchronize_rcu() and call_rcu()) must have the property that any RCU read-side critical section in existence at the start of a grace period completes by the end of the grace period.
2) RCU read-side primitives should avoid expensive operations such as cache misses, atomic instructions, memory barriers, and conditional branches.
3) RCU read-side primitives should have $O(1)$ computational complexity to enable real-time use. This property guarantees freedom from deadlock.
4) RCU read-side primitives should be usable in all contexts, including nested within other RCU read-side critical sections. Another important special context is library functions having incomplete knowledge of the user application.
5) RCU read-side primitives should be unconditional, thus eliminating the failure checking that would otherwise complicate testing and validation. This property has the nice side-effect of avoiding livelocks.
6) RCU read-side should not cause write-side starvation: grace periods should always complete, even given a steady flow of time-bounded read-side critical sections.
7) Any operation other than a quiescent state (and thus a grace period) should be permitted within an RCU read-side critical section. In particular, non-idempotent operations such as I/O and lock acquisition/release should be permitted.
8) It is permissible to mutate an RCU-protected data structure while executing within an RCU read-side critical section. Of course, any grace periods following this mutation must occur after the RCU read-side critical section completes.
9) RCU primitives should be independent of memory allocator design and implementation, so that RCU data structures may be protected regardless of how their data elements are allocated and freed.
10) RCU grace periods should not be blocked by threads that halt outside of RCU read-side critical sections. (But note that most quiescent-state-based implementations violate this desideratum.)

The RCU implementations described in Section IV are designed to meet the above list of desiderata.

C. 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 column of the figure presents the data structure view of the updater thread. The second column

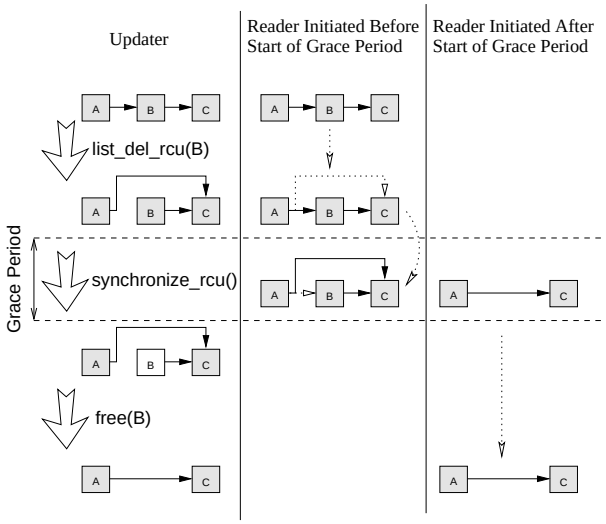

Fig. 3. RCU Linked-List Deletion

presents the data structure view of a reader thread starting before the grace period begins. The third column presents a reader thread starting after the beginning of the grace period.

The first row of the figure shows a list with elements A, B, and C, to each of which every RCU readers initiated before the beginning of the grace period might both acquire and hold references.

The list_del_rcu() primitive unlinks element B from the list, but leaves the link from B to C intact, as shown on the second row of the figure. This permits any RCU readers currently referencing B to advance to C, as shown on the second and third rows of the figure. The transition between the second and third rows shows the reader thread data structure view gradually seeing element B disappear. During this transition, some readers will see element B and others will not. Although there might be RCU readers still referencing Element B, new RCU readers can no longer acquire a reference to it.

The synchronize_rcu() primitive waits for one grace period, after which all pre-existing RCU read-side critical sections will have completed, resulting in the state shown in the fourth row of the figure. This state is the same as the second and third rows, except for the fact that there can no longer be any RCU readers holding references to Element B. This change of state of B from *globally visible* to *private* is depicted by using a white background for the B box. At this point, it is safe to invoke free(), reclaiming the memory consumed by element B, as shown on the last row of the figure.

Of course, the deletion process must be protected by some mutual-exclusion mechanism, most commonly, by locking.

Although RCU is used in a wide variety of ways, this list-deletion process is the most common usage.

D. Overview of RCU Semantics

RCU semantics comprise the grace-period guarantee covered in Section II-D1 and the publication guarantee discussed in Section II-D2. Synchronization guarantees among concurrent modifications of the RCU-protected data structure must be provided by some other mechanism. In the Linux kernel, this other mechanism is typically locking, but any other suitable mechanism may be used, including atomic operations, non-blocking synchronization, transactional memory, or a single designated updater thread.

1) Grace-Period Guarantee: RCU operates by defining *RCU read-side critical sections*, delimited by rcu_read_lock() and rcu_read_unlock(), and by defining *grace periods*, which 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. The RCU primitive synchronize_rcu() starts a grace period and then waits for it to complete. Most RCU implementations allow RCU read-side critical sections to be nested.

Somewhat more formally, suppose we have a group of C-language statements S_i within an RCU read-side critical section as follows:

rcu_read_lock(); S_0; S_1; S_2; ...; rcu_read_unlock();

Suppose further that we have a group of C-language mutation statements M_i and a group of C-language destruction statements D_i separated by an RCU grace period:

M_0; M_1; M_2; ...; synchronize_rcu(); D_0; D_1; D_2; ...;

Then the following holds, where "$\rightarrow$" indicates that the statement on the left executes prior to that on the right, and where "$\Longrightarrow$" denotes logical implication:

$$\exists S_a, M_b (S_a \rightarrow M_b) \Longrightarrow \forall S_i, D_j (S_i \rightarrow D_j) \qquad (1)$$

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

This guarantee permits RCU-based algorithms to trivially avoid a number of difficult race conditions that can otherwise result in poor performance, limited scalability, and great complexity. However this guarantee is insufficient, as it does not show that readers can operate consistently while an update is in progress. This case is covered by the guarantee presented in the next section.

2) Publication Guarantee: It is important to note that the statements S_a and M_b may execute concurrently, even in the case where S_a is referencing the same data element that M_b is concurrently modifying. The publication guarantee associated with the rcu_assign_pointer() and rcu_dereference() primitives allow this concurrency to be handled both correctly and easily: any dereference of a pointer returned by rcu_dereference() is guaranteed to see any changes prior to the corresponding rcu_assign_pointer(), including any changes prior to any earlier rcu_assign_pointer() involving that same pointer.

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

I_0; I_1; I_2; ...; rcu_assign_pointer(g,p);

where each I_i is a C-language statement that initializes a field in the structure referenced by the local pointer p, and where the global pointer g is visible to reading threads.

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

q = rcu_dereference(g); R_0; R_1; R_2; ...;

where this RCU read-side critical section is enclosed in rcu_read_lock() and rcu_read_unlock(), q is a local pointer, g is the same global pointer updated by the earlier rcu_assign_pointer() (and possibly updated again by some later invocations of rcu_assign_pointer()), and each R_i dereferences q to access one of the fields initialized by one of the statements I_i.

Then we have the following, where A is the rcu_assign_pointer() and D is the rcu_dereference():

$$A \rightarrow D \Longrightarrow \forall I_i, R_j (I_i \rightarrow R_j) \tag{2}$$

In other words, if a given rcu_dereference() statement accesses the value stored 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 the rcu_assign_pointer().

This guarantee allows new data to be initialized and added to an RCU-protected data structure in face of concurrent RCU readers.

Given both the grace-period and publication guarantees, these five primitives enable a wide variety of algorithms and data structures providing extremely low read-side overheads for read-mostly data structures (8; 6; 9; 10). Again, note that concurrent updates must be handled by some synchronization mechanism, be it locking, atomic operations, non-blocking synchronization, transactional memory, or a single updater thread.

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 past year has seen increased interest in applying RCU to user-space applications.

User-level RCU was needed for a user-level infrastructure that provides low-overhead tracing for user-mode applications. RCU is used for tracer control data synchronization in the LTTng tracer implementation (11), which is being ported to a user-space library. This usage scenario poses important constraints on the RCU requirements. This tracing library cannot be too intrusive in terms of program modification, which makes the QSBR approach presented in Section IV-B inappropriate for such usage scenario. It also needs to support extensible instrumentation of user-selected execution sites, including signal handlers, which therefore requires supporting nested RCU critical sections and RCU reader critical sections in signal handlers. This usage scenario is also very performance demanding on workloads involving instrumentation of frequent execution sites. Therefore, having a low-overhead and scalable read-side is very important. Therefore, an ideal locking primitive for a tracing library would require no knowledge of the application and could be used to protect data structures used in a library.

User-level RCU has also been proposed for an elliptics-network distributed cloud-based storage project (12). BIND, a major domain name server at the root of Internet domain name resolution, is facing multi-threading scalability issues which are currently addressed with reader-writer locks (13). Given the domain names are read often but rarely updated, these could benefit from major performance improvement by using user-level RCU. Others have mentioned possibilities in financial applications. One can also argue that RCU has seen long use at user level in the guise of user-mode Linux.

In general, the area of applicability of RCU to user-mode applications 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, with Sections IV-B, IV-C, and IV-D presenting user-space RCU implementations that are optimized for different usage by user-space applications, but first Section IV-A describes some primitives which might be unfamiliar. The implementation presented in Section IV-B offers the best possible read-side performance, but requires that each of the application's threads periodically pass through a quiescent state, thus strongly constraining the application's design. The implementation presented in Section IV-C places almost no constraints on the application's design, thus being appropriate for use within a general-purpose library, but having higher read-side overhead. Section IV-D presents an implementation having low read-side overhead, and requiring only that the application give up one signal to RCU processing. Finally, Section IV-E demonstrates how to create wait-free RCU update primitives.

A. Notation

The examples in this section use a number of primitives that may be unfamiliar, and are thus listed in this section.

Per-thread variables are defined via DEFINE_PER_THREAD(). A thread may access its own instance of a per-thread variable using __get_thread_var(), or some other thread's instance via per_thread(). The for_each_thread() primitive sequences through all threads, one at a time.

The pthread_mutex is a type defined by the pthread library for mutual exclusion variables. The mutex_lock() primitive acquires a pthread_mutex instance, and mutex_unlock() releases it. The mb keyword stands for "memory barrier". The smp_mb() primitive emits a full memory barrier, for example, the sync instruction on the PowerPC architecture. The smp_wmb() and smp_rmb() primitives are, respectively, store and load memory barriers, corresponding, for example, to the sfence and lfence instructions on the x86 architecture. The ACCESS_ONCE() primitive prohibits any compiler optimization that might otherwise turn a single fetch or store into multiple fetches, as might happen under heavy register pressure. The barrier() primitive prohibits any compiler code-motion optimization that might otherwise move fetches or stores across the barrier() primitive.

```
 1 long rcu_gp_ctr = 0;
 2 DEFINE_PER_THREAD(long, rcu_reader_qs_gp);
 3
 4 static inline void rcu_read_lock(void)
 5 {
 6 }
 7
 8 static inline void rcu_read_unlock(void)
 9 {
10 }
11
12 static inline void rcu_quiescent_state(void)
13 {
14   smp_mb();
15   __get_thread_var(rcu_reader_qs_gp) =
16     ACCESS_ONCE(rcu_gp_ctr) + 1;
17   smp_mb();
18 }
19
20 static inline void rcu_thread_offline(void)
21 {
22   smp_mb();
23   __get_thread_var(rcu_reader_qs_gp) =
24     ACCESS_ONCE(rcu_gp_ctr);
25 }
26
27 static inline void rcu_thread_online(void)
28 {
29   __get_thread_var(rcu_reader_qs_gp) =
30     ACCESS_ONCE(rcu_gp_ctr) + 1;
31   smp_mb();
32 }
```

Fig. 4. RCU Read Side Using Quiescent States

B. Quiescent-State-Based Reclamation RCU

The QSBR RCU implementation provides near zero-overhead read-side, but requires to modify the application, as this section explains.

Figure 4 shows the read-side primitives used to construct a user-level quiescent-state-based reclamation (QSBR) implementation of RCU based on quiescent states. As can be seen from lines 4–10 in the figure, the rcu_read_lock() and rcu_read_unlock() primitives do nothing, and can in fact be expected to be inlined and optimized away, as they are in server builds of the Linux kernel. This is due to the fact that quiescent-state-based RCU implementations *approximate* the extents of RCU read-side critical sections using the aforementioned quiescent states, which contain calls to rcu_quiescent_state(), shown from lines 12–18 in the figure. Threads entering extended quiescent states (for example, when blocking) may instead use the thread_offline() and thread_online() APIs to mark the beginning and the end, respectively, of such an extended quiescent state. As such, thread_online() is analogous to rcu_read_lock() and thread_offline() is analogous to rcu_read_unlock(). These two functions are shown on lines 20–32 in the figure. In either case, it is invalid for a quiescent state to appear within an RCU read-side critical section.

In rcu_quiescent_state(), line 14 executes a memory barrier to prevent any code prior to the quiescent state from being reordered into the quiescent state. Lines 15–16 pick up a copy of the global rcu_gp_ctr (RCU grace-period counter), using ACCESS_ONCE() to ensure that the compiler does not employ any optimizations that would result in rcu_gp_ctr being fetched more than once, and then adds one to the value fetched and stores it into the per-thread rcu_reader_qs_gp variable, so that any concurrent instance of synchronize_rcu()

```
 1 static inline int rcu_gp_ongoing(int thread)
 2 {
 3   return per_thread(rcu_reader_qs_gp, thread) & 1;
 4 }
 5
 6 void synchronize_rcu(void)
 7 {
 8   int t;
 9
10   smp_mb();
11   mutex_lock(&rcu_gp_lock);
12   rcu_gp_ctr += 2;
13   for_each_thread(t) {
14     while (rcu_gp_ongoing(t) &&
15            ((per_thread(rcu_reader_qs_gp, t) -
16              rcu_gp_ctr) < 0)) {
17       poll(NULL, 0, 10);
18       barrier();
19     }
20   }
21   mutex_unlock(&rcu_gp_lock);
22   smp_mb();
23 }
```

Fig. 5. RCU Update Side Using Quiescent States

will see an odd-numbered value, thus becoming aware that a new RCU read-side critical section has started. Instances of synchronize_rcu() that are waiting on older RCU read-side critical sections will know to ignore this new one. Finally, line 17 executes a memory barrier to ensure that the update to rcu_reader_qs_gp is seen by all threads to happen before any subsequent RCU read-side critical sections.

Some applications might use RCU only occasionally, but use it very heavily when they do use it. Such applications might choose to use rcu_thread_online() when starting to use RCU and rcu_thread_offline() when no longer using RCU. The time between a call to rcu_thread_offline() and a subsequent call to rcu_thread_online() is an extended quiescent state, so that RCU will not expect explicit quiescent states to be registered during this time.

The rcu_thread_offline() function simply sets the per-thread rcu_reader_qs_gp variable to the current value of rcu_gp_ctr, which has an even-numbered value. Any instance of synchronize_rcu() will thus know to ignore this thread. A memory barrier is needed at the beginning of the function to ensure all RCU read side-effects are globally visible before making the thread appear offline. No memory barrier is needed in the innermost part of rcu_thread_offline() because it is invalid to perform RCU accesses on this side of the function. There is therefore no need to prevent reordering.

The rcu_thread_online() function is the counterpart of rcu_thread_offline(). It marks the end of the extended quiescent state. It is similar to rcu_quiescent_state(), except that the only memory barrier required is at the end of the function.

Figure 5 shows the implementation of synchronize_rcu(). It implicitly refers to the variables declared in Lines 1–2 of Figure 4. Lines 1–4 show the rcu_gp_ongoing() helper function, which returns true if the specified thread's rcu_reader_qs_gp variable has an odd-numbered value. Lines 6–22 show the implementation of synchronize_rcu() itself. Line 10 is a memory barrier that ensures that the caller's mutation of the RCU-protected data structure is seen by all CPUs to happen before the grace period identified by this invocation of synchronize_rcu(). Line 11 acquires a pthread_mutex

named rcu_gp_lock in order to serialize concurrent calls to synchronize_rcu(), and line 21 releases it. Line 12 adds the value "2" to the global variable rcu_gp_ctr to indicate the beginning of a new grace period. Line 13 sequences through all threads, and lines 14–16 check to see if the current thread is still in an RCU read-side critical section that began before the counter was incremented back on line 12: if so, we must wait for it on line 17. Line 18 ensures that the compiler refetches the rcu_reader_qs_gp variable. Line 22 executes one last memory barrier to ensure that all other CPUs have fully completed their RCU read-side critical sections before the caller of synchronize_rcu() performs any destructive actions (such as freeing up memory).

This implementation has low-cost read-side primitives, as can be seen in Figure 4. Read-side overhead depends on how often rcu_quiescent_state() is called. These read-side primitives qualify as wait-free under the most severe conceivable definition (14). The synchronize_rcu() overhead ranges from about 600 nanoseconds on a single-CPU Power5 system up to more than 100 microseconds on a 64-CPU system with one thread per CPU.

Because it waits for readers to complete, synchronize_rcu() does not qualify as non-blocking. Section IV-E describes how RCU updates can support wait-free algorithms in the same sense as wait-free algorithms are supported by garbage collectors.

However, this implementation requires that each thread either invoke rcu_quiescent_state() periodically or invoke rcu_thread_offline() for extended quiescent states. The need to invoke these functions periodically can make this implementation difficult to use in some situations, such as for certain types of library functions.

In addition, this implementation does not permit concurrent calls to synchronize_rcu() to share overlapping grace periods. That said, one could easily imagine a production-quality RCU implementation based on this version of RCU.

Finally, on systems where the rcu_gp_ctr is implemented using 32-bit counters, this algorithm can fail if a reader is preempted in line 3 of rcu_read_lock() in Figure 4 for enough time to allow the rcu_gp_ctr to advance through more than half (but not all) of its possible values. Although one solution is to avoid 32-bit systems, 32-bit systems can be handled by adapting rcu_read_lock() and rcu_read_unlock() from Figure 6 for use in rcu_quiescent_state() and rcu_offline_thread(), respectively. This would of course also require adopting the synchronize_rcu() implementation from Figure 7.

Another point worth discussing is that if read-side critical sections are expected to execute in a signal handler, the rcu_quiescent_state() primitive must run with signals disabled, and signals must be kept disabled while threads are kept offline. Effectively, if a signal handler nests over rcu_quiescent_state() between the memory barriers, the read-side could be interleaved with the rcu_reader_qs_gp update and therefore spawn across two grace periods, which could cause synchronize_rcu() to return before the quiescent state is reached and lead to data corruption.

The next section discusses an RCU implementation that is safe for use in libraries, where the library code cannot guarantee that all threads of a yet-as-unwritten application will traverse quiescent states in a timely fashion.

```
 1 #define RCU_GP_CTR_BOTTOM_BIT 0x80000000
 2 #define RCU_GP_CTR_NEST_MASK (RCU_GP_CTR_BOTTOM_BIT - 1)
 3 long rcu_gp_ctr = 1;
 4 DEFINE_PER_THREAD(long, rcu_reader_gp);
 5
 6 static inline void rcu_read_lock(void)
 7 {
 8   long tmp;
 9   long *rrgp;
10
11   rrgp = &__get_thread_var(rcu_reader_gp);
12   tmp = *rrgp;
13   if ((tmp & RCU_GP_CTR_NEST_MASK) == 0) {
14     *rrgp = ACCESS_ONCE(rcu_gp_ctr);
15     smp_mb();
16   } else {
17     *rrgp = tmp + 1;
18   }
19 }
20
21 static inline void rcu_read_unlock(void)
22 {
23   long tmp;
24
25   smp_mb();
26   __get_thread_var(rcu_reader_gp)--;
27 }
```

Fig. 6. RCU Read Side Using Memory Barriers

C. General-Purpose RCU

The general-purpose RCU implementation can in theory be used in any software environment, including even in library functions that are not aware of the design of the enclosing application. However, the price paid for this generality is relatively high read-side overhead, though this overhead is still significantly less than a single compare-and-swap operation on most hardware.

A global variable rcu_gp_ctr is initialized to 1 and a per-thread variable rcu_reader_gp is initialized to zero. The low-order bits of rcu_reader_gp is a count of the rcu_read_lock() nesting depth, while the upper bit indicates the grace-period phase at the time of the invocation of the outermost rcu_read_lock() (15). The upper bit of global variable rcu_gp_ctr is the current grace-period phase, while the low-order field is set to the value 1 for reasons that will become apparent shortly.

The read-side primitives are shown in Figure 6. Lines 1–4 are declarations, lines 6–19 are rcu_read_lock(), and lines 21–27 are rcu_read_unlock().

In rcu_read_lock(), line 11 obtains a reference to the current thread's instance of rcu_reader_gp, and line 12 fetches the contents into the local variable tmp. Line 13 then checks to see if this is the outermost rcu_read_lock(), and, if so, line 14 copies the current value of the global rcu_gp_ctr to this thread's rcu_reader_gp variable, thereby snapshotting the current grace-period phase and setting the nesting count to 1 in a single operation. Otherwise, line 17 increments the nesting count in this thread's rcu_reader_gp variable.

Line 26 decrements the thread's rcu_reader_gp, which has the effect of decrementing the nesting count.

For outermost read-side rcu_read_lock(), the memory barrier on line 15 ensures that the rcu_reader_gp value is globally observable before any of the outermost read-side critical

```
 1 static inline int rcu_old_gp_ongoing(int t)
 2 {
 3   int v = ACCESS_ONCE(per_thread(rcu_reader_gp, t));
 4
 5   return (v & RCU_GP_CTR_NEST_MASK) &&
 6          ((v ^ rcu_gp_ctr) & ~RCU_GP_CTR_NEST_MASK);
 7 }
 8
 9 static void flip_counter_and_wait(void)
10 {
11   int t;
12
13   rcu_gp_ctr ^= RCU_GP_CTR_BOTTOM_BIT;
14   for_each_thread(t) {
15     while (rcu_old_gp_ongoing(t)) {
16       poll(NULL, 0, 10);
17       barrier();
18     }
19   }
20 }
21
22 void synchronize_rcu(void)
23 {
24   smp_mb();
25   mutex_lock(&rcu_gp_lock);
26   flip_counter_and_wait();
27   flip_counter_and_wait();
28   mutex_unlock(&rcu_gp_lock);
29   smp_mb();
30 }
```

Fig. 7. RCU Update Side Using Memory Barriers

section memory accesses. It ensures that neither the compiler nor the CPU will reorder memory accesses across this barrier by adding a compiler barrier and issuing a memory barrier instruction. Only the outermost rcu_read_lock() needs to have such memory barrier because only this outermost lock can change the reader's current grace period.

In rcu_read_unlock(), line 25 executes a memory barrier to ensure that all globally observable effects of the RCU read-side critical section reach memory before rcu_reader_gp is decremented. The memory barrier on line 25 is needed only for the outermost rcu_read_unlock(), but given the outermost and innermost nesting level behave in the exact same way, a branch in the rcu_read_unlock() code is unneeded, and given the common case is to perform single-level nesting, the memory barrier is executed unconditionally for innermost and outermost nesting levels.

Section IV-D shows one way of getting rid of both memory barriers; however, even with the memory barriers, both rcu_read_lock() and rcu_read_unlock() are wait-free.

The effect of this implementation of rcu_read_lock() and rcu_read_unlock() is that a given thread may be ignored by the current grace-period phase in either of the following cases:

1) The lower-order bits of the thread's rcu_reader_gp variable are all zero, in which case the thread is not currently in an RCU read-side critical section.
2) The upper bit of the thread's rcu_reader_gp variable matches that of the global rcu_gp_ctr, in which case this thread's RCU read-side critical section started after the beginning of the current grace-period phase.

These checks are implemented by the function rcu_old_gp_ongoing(), which is shown on lines 1–7 of Figure 7. This figure implicitly refers to the declarations and variables in Lines 1–4 of Figure 6. Given a thread t, line 3 fetches t's rcu_reader_gp variable, with the ACCESS_ONCE() primitive ensuring the variable is read with a single memory access. This prevents the compiler from refetching the variable or fetching it in pieces. Line 5 then checks to see if the low-order field is non-zero, and line 6 checks to see if the upper bit differs from that of the rcu_gp_ctr global variable. Only if both these conditions hold does rcu_old_gp_ongoing() report that the current grace-period phase must wait on this thread.

Lines 9–20 of Figure 7 show flip_counter_and_wait(), which initiates a grace-period phase and waits for it to elapse. Line 13 complements the upper bit of global variable rcu_gp_ctr, which initiates a new grace-period phase. Line 14 cycles through all threads. The "while" loop at line 15 repeatedly executes lines 16–17 until rcu_old_gp_ongoing() reports that the thread no longer resides in an RCU read-side critical section that affects the current grace-period phase. Line 16, which is optional, blocks for a short period of time, and line 17 ensures that the compiler refetches variables when executing rcu_old_gp_ongoing().

Lines 22–30 of Figure 7 shows synchronize_rcu(), which waits for a full two-phase grace period to elapse. Line 24 executes a memory barrier to ensure that any prior data-structure modification is seen by all threads to precede the grace period. Line 25 acquires rcu_gp_lock to serialize any concurrent invocations of synchronize_rcu(). Lines 26–27 wait for two grace-period phases, line 28 releases the lock, and line 29 executes a memory barrier to ensure that all threads see the grace period happening before any subsequent destructive operations (such as free()).

Memory ordering between the rcu_gp_ctl complement and testing the reader's current grace period with rcu_old_gp_ongoing() is not strictly needed. The only requirement is that each and every reader thread that was executing in a read-side critical section before memory barrier on line 24 has finished its critical section after the memory barrier on line 29. This two-phase grace period scheme is used to ensure updater progress through a grace period even if a steady flow of readers comes. The only requirement is that, when the updater busy-loops waiting for readers, it eventually reaches a point where all new readers are in the new grace period parity.

Grace period identification, by either a bit (in the two-phase scheme) or by a counter, ensures that readers starting during the grace period will not prevent the grace period from completing. In fact, if a simplistic scheme where the updater waits for *all* readers to complete would be used, the grace period would be considered as complete when the updater reaches a point where no reader is active in the system. However, this would allow new readers starting after the beginning of the grace period to impede reaching quiescent state. This would prevent grace-period progress in the presence of reader threads releasing the read-side critical section for very short periods. Faster cached local data access would therefore provide an unfair advantage to the reader over the updater.

Now that the grace period identification question is settled, this raises the question "why isn't a single grace-period phase sufficient?" To see why, consider the following sequence of events which involves one read-side critical section and two consecutive grace periods:

1) Thread A invokes rcu_read_lock(), executing lines 11–13 of Figure 6, and finding that this instance of rcu_read_lock() is not nested, fetching the value of rcu_gp_ctr on line 14, but not yet storing it.
2) Thread B invokes synchronize_rcu(), executing lines 24 and 25 of Figure 7, then invoking flip_counter_and_wait() on line 26, where it complements the grace-period phase bit on line 13, so that the new value of this bit is now 1.
3) Because no thread is in an RCU read-side critical section (recall that thread A has not yet executed the store operation on line 14), Thread B proceeds through lines 14–19 of Figure 7, returns to synchronize_rcu(), executing lines 28–30 (recall that line 27 is omitted in this scenario), and returning to the caller.
4) Thread A now performs the store in line 14 of Figure 6. Recall that it is using the old value of rcu_gp_ctr where the value of the grace-period phase bit is 0.
5) Thread A then executes the memory barrier on line 15, and returns to the caller, which proceeds in to the RCU read-side critical section.
6) Thread B invokes synchronize_rcu() once more, again complementing the grace-period phase bit on line 13 of Figure 7, so that the value is again zero.
7) When Thread B examines Thread A's rcu_reader_gp variable on line 6 of Figure 7, it finds that the grace-period phase bit matches that of the global variable rcu_gp_ctr. Thread A is therefore ignored, and Thread B therefore exits from synchronize_rcu().
8) But Thread A is still in its RCU read-side critical section in violation of RCU semantics.

Invoking flip_counter_and_wait() twice avoids this problem by making sure the grace period waits for reader critical sections for each of the possible two phases.

A single-phase approach is possible if the current grace period is identified by a free-running counter, as shown in Section IV-B. However, the counter size is important because this counter is subject overflow. The single-flip problem shown above, which involves two consecutive grace periods, is actually a case where a single-bit overflow occurs. A similar scenario is therefore possible given a number of grace periods sufficient to overflow the grace period counter passing during a read lock section. This could realistically happen on 32-bit architectures if read-side critical sections are preempted.

The following section shows one way to eliminate the read-side memory barriers.

D. Low-Overhead RCU Via Signal Handling

The largest sources of overhead for the QSBR and general-purpose RCU read-side primitives shown in Figures 4 and 6 are the memory barriers. One way to eliminate this overhead is to use POSIX signals. The readers' signal handlers contain memory-barrier instructions, which allows an updater to force readers to execute a memory-barrier instruction only when needed, rather than suffering the extra overhead during every call to a read-side primitive.

```
 1 #define RCU_GP_COUNT         (1UL << 0)
 2 #define RCU_GP_CTR_BIT       (1UL << (sizeof(long) * 4))
 3 #define RCU_GP_CTR_NEST_MASK (RCU_GP_CTR_BIT - 1)
 4
 5 long urcu_gp_ctr = RCU_GP_COUNT;
 6 long __thread urcu_active_readers = 0L;
 7
 8 static inline void rcu_read_lock(void)
 9 {
10   long tmp;
11
12   tmp = urcu_active_readers;
13   if (!(tmp & RCU_GP_CTR_NEST_MASK))
14     urcu_active_readers = ACCESS_ONCE(urcu_gp_ctr);
15   else
16     urcu_active_readers = tmp + RCU_GP_COUNT;
17   barrier();
18 }
19
20 static inline void rcu_read_unlock(void)
21 {
22   barrier();
23   urcu_active_readers = urcu_active_readers - RCU_GP_COUNT;
24 }
```

Fig. 8. RCU Read Side Using Signals

One unexpected but quite pleasant surprise is that this approach results in relatively simple read-side primitives. In contrast, those of preemptable RCU are notoriously complex.

The read-side primitives are shown in Figure 8, along with the data definitions and state variables. The urcu_ prefix used for variables stands for "user-space RCU" Lines 1–3 show the definitions controlling both the urcu_gp_ctr global variable (line 5) and the urcu_active_readers per-thread variable (line 6). The low-order bits (those corresponding to 1-bit in RCU_GP_CTR_NEST_MASK) are used to count the rcu_read_lock() nesting level, while the bit selected by RCU_GP_CTR_BIT is used to detect grace periods. All other bits are unused. The global urcu_gp_ctr may be accessed at any time by any thread, but may be updated only by the thread holding the lock that guards grace-period detection. The per-thread urcu_active_readers variable may be modified only by the corresponding thread, and is otherwise read only by the thread holding the lock that guards grace-period detection.

The rcu_read_lock() implementation is shown on lines 9–18. Line 12 picks up the current value of this thread's urcu_active_readers variable and places it in the local variable tmp. Line 13 checks to see if the nesting-level portion of urcu_active_readers is zero (indicating that this is the outermost rcu_read_lock()), and, if so, line 14 copies the global variable urcu_gp_ctr to this thread's urcu_active_readers variable. Note that urcu_gp_ctr has been initialized with its low-order bit set, so that the nesting level is automatically set correctly. Otherwise, line 16 increments the nesting level in this thread's urcu_active_readers variable. In either case, line 17 executes a barrier directive in order to prevent the compiler from undertaking any code-motion optimization that might otherwise cause the contents of the subsequent RCU read-side critical section to be reordered to precede the rcu_read_lock().

The implementation of rcu_read_unlock() is shown on lines 20–24. Line 22 executes a barrier directive, again, in order to prevent the compiler from undertaking any code-motion optimization that might otherwise cause the contents of the prior RCU read-side critical section to be reordered to

```
 1 struct reader_registry {
 2   pthread_t tid;
 3   long *urcu_active_readers;
 4   char *need_mb;
 5 } *registry;
 6 static char __thread need_mb;
 7 static int num_readers;
 8
 9 static void force_mb_all_threads(void)
10 {
11   struct reader_registry *index;
12
13   if (!registry)
14     return;
15   index = registry;
16   for (; index < registry + num_readers; index++) {
17     *index->need_mb = 1;
18     pthread_kill(index->tid, SIGURCU);
19   }
20   index = registry;
21   for (; index < registry + num_readers; index++) {
22     while (*index->need_mb) {
23       pthread_kill(index->tid, SIGURCU);
24       poll(NULL, 0, 1);
25     }
26   }
27   smp_mb();
28 }
29
30 static void sigurcu_handler(int signo, siginfo_t *siginfo,
31                             void *context)
32 {
33   smp_mb();
34   need_mb = 0;
35   smp_mb();
36 }
```

Fig. 9. RCU Signal Handling

follow the rcu_read_unlock(). Line 23 decrements the value of this thread's urcu_active_readers variable, so that if this is the outermost rcu_read_unlock(), the low-order bits indicating the nesting level will now be zero.

Both rcu_read_lock() and rcu_read_unlock() execute a sharply bounded number of instructions, hence both are wait-free.

The signal-handling primitives are shown in Figure 9, including variable declarations on lines 1–7, force_mb_all_threads() on lines 9–28 and sigurcu_handler() on lines 30–36.

The structures on lines 1–5 represents a thread, with its thread ID in tid, a pointer to its urcu_active_readers per-thread variable, and a pointer to its need_mb per-thread variable. Line 6 declares the per-thread need_mb variable, and line 7 defines the global variable num_readers, which contains the number of threads that are represented in the registry array defined on line 5.

The force_mb_all_threads() function ensures a memory barrier is executed on each running threads by sending a POSIX signal to all threads, waiting for each to respond. As we will see, this has the effect of promoting compiler-ordering directives such as barrier() to full memory barriers, while avoiding the need to incur the cost of expensive barriers in read-side primitives in the common case. Lines 13–14 return if there are no readers, and lines 16-19 set each thread's need_mb per-thread variable to the value one, then send that thread a POSIX signal. Note that the system call executed for pthread_kill() implies a full memory barrier before the system call execution at the operating system level. This memory barrier ensures that all memory accesses done prior to the call to pthread_kill() are not reordered after the start of the system call. Lines 20–26 then rescan the threads, waiting until one each has responded by setting its need_mb per-thread variable to zero. Because some versions of some operating systems can lose signals, line 23 will resend the signal if a response is not received in a timely fashion. Finally, line 27 executes a memory barrier to ensure that the signals have been received and acknowledged before later operations that might otherwise destructively interfere with readers.

Lines 30–36 show the signal handler that runs in response to a given thread receiving the POSIX signal sent by force_mb_all_threads(). This sigurcu_handler() function executes a pair of memory barriers separated by setting its need_mb per-thread variable to zero. This has the effect of placing a full memory barrier at whatever point in the thread's code that was executing at the time that the signal was received, preventing the CPU from reordering across that point.

The sender thread has two memory barriers around whole sequence consisting of sending the signal and waiting for the remote thread to acknowledge its reception. The remote thread executes a memory barrier before acknowledging the signal reception. These two conditions ensure that the remote thread's program order and memory accesses passed by a point where they were executing in order between the two memory barriers on the sender thread. Therefore, execution in program order and with ordered memory accesses is ensured on the remote processor at that point. This promotes all compiler barriers on the receiver side to memory barriers, but only when the matching memory barrier is executed on the sender side.

The update-side grace-period primitives are shown in Figure 10, including switch_next_urcu_qparity() on lines 1–4, rcu_old_gp_ongoing() on lines 6–15, wait_for_quiescent_state() on lines 17–29, and synchronize_rcu() on lines 31–41.

The switch_next_urcu_qparity() function starts a new grace-period phase, where a pair of such phases make up a grace period. A single phase is insufficient for the same reasons discussed in Section IV-C. This function simply complements the designated bit in the urcu_gp_ctr global variable.

The rcu_old_gp_ongoing() determines whether or not the thread with the referenced per-thread urcu_active_readers variable is still executing within an RCU read-side critical section that started before this grace-period phase. Lines 10–11 check to see if there is no thread, and returns zero if there is not, given that a non-existent thread cannot be executing at all, let alone within an RCU read-side critical section. This will hold for the whole grace-period because thread registration needs to hold the internal_rcu_lock. Otherwise, line 12 fetches the value, using the ACCESS_ONCE() primitive to defeat compiler optimizations that might otherwise cause the value to be fetched more than once. Line 13 then checks to see if the corresponding thread is in an RCU read-side critical section, and, if so, line 14 checks to see if that RCU read-side critical section predates the beginning of the current grace-period phase.

The wait_for_quiescent_state() waits for each thread to pass through a quiescent state, thereby completing one phase of the grace period. Lines 21–22 return immediately if there are no threads. Otherwise, the loop spanning lines 23–28 waits

```
 1 static void switch_next_urcu_qparity(void)
 2 {
 3   urcu_gp_ctr = urcu_gp_ctr ^ RCU_GP_CTR_BIT;
 4 }
 5
 6 static inline int rcu_old_gp_ongoing(long *value)
 7 {
 8   long v;
 9
10   if (value == NULL)
11     return 0;
12   v = ACCESS_ONCE(*value);
13   return (v & RCU_GP_CTR_NEST_MASK) &&
14      ((v ^ urcu_gp_ctr) & RCU_GP_CTR_BIT);
15 }
16
17 static void wait_for_quiescent_state(void)
18 {
19   struct reader_registry *i;
20
21   if (!registry)
22     return;
23   i = registry;
24   for (; i < registry + num_readers; i++) {
25     while (rcu_old_gp_ongoing(i->urcu_active_readers))
26       cpu_relax();
27     }
28   }
29 }
30
31 void synchronize_rcu(void)
32 {
33   internal_urcu_lock();
34   force_mb_all_threads();
35   switch_next_urcu_qparity();
36   wait_for_quiescent_state();
37   switch_next_urcu_qparity();
38   wait_for_quiescent_state();
39   force_mb_all_threads();
40   internal_urcu_unlock();
41 }
```

Fig. 10. RCU Update Side Using Signals

for each thread to exit any pre-existing RCU read-side critical section.

The synchronize_rcu() primitive waits for a full grace period to elapse. Line 33 acquires a pthread_mutex that prevents concurrent synchronize_rcu() invocations from interfering with each other and reader thread registration. Line 40 releases this same pthread_mutex. Line 34 ensures that any thread that sees the start of the new grace period (line 35) will also see any changes made by the caller prior to the synchronize_rcu() invocation. Line 35 starts a new grace-period phase, and line 36 waits for it to complete. Lines 37 and 38 similarly start and end a second grace-period phase. Line 39 forces each thread to execute a memory barrier, ensuring that each thread will see any destructive actions subsequent to the call to synchronize_rcu() as happening after any RCU read-side critical section that started before the grace period began.

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 wait-freedom to RCU updates as well as to RCU readers.

E. Wait-Free RCU Updates

Although some algorithms use RCU as a first-class technique, in most situations RCU is instead simply used as an approximation to a garbage collector. In these situations, given sufficient memory, the delays built into synchronize_rcu() need not block the algorithm itself, just as delays built into an automatic garbage collector need not block a wait-free algorithm.

```
 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(head);
16     synchronize_rcu();
17     while (wait) {
18       next = wait->next;
19       wait->func(wait);
20       wait = next;
21     }
22     poll(NULL, 0, 1);
23   }
24 }
```

Fig. 11. Avoiding Update-Side Blocking by RCU

One way of accomplishing this is shown in Figure 11, which implements the asynchronous call_rcu() primitive found in the Linux kernel. Lines 4 and 5 initialize an RCU callback, and line 6 uses a wait-free enqueue algorithm (16) to enqueue the callback on the rcu_data list. This call_rcu() function is then clearly wait-free.

A separate thread would remove and invoke these callbacks after a grace period has elapsed, using synchronize_rcu() for this purpose, as shown on lines 9–24 of Figure 11, with each pass of the loop spanning lines 14–23 waiting for one grace period. Line 15 uses a (possibly blocking) dequeue algorithm to remove all elements from the rcu_data list en masse, and line 16 waits for a grace period to elapse. Lines 17–21 invoke all the RCU callbacks from the list dequeued by line 15. Finally, line 22 blocks for a short period to allow additional RCU callbacks to be enqueued. Note that the longer line 22 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. However, as long as the callback function func that was passed to call_rcu() does nothing other than free memory, as long as the synchronization mechanism used to coordinate RCU updates is wait-free, and as long as there is sufficient memory for allocations to succeed without blocking, RCU-based algorithms that use call_rcu() will themselves be wait-free.

V. Experimental Results

This section presents benchmarks of each RCU mechanism presented in this paper with respect to each other, compared

to mutexes, to reader-writer locks and to per-thread locks[1]. It first demonstrates read-side scalability, discusses the impact of read-side critical section length on the respective locking primitive behavior and finally presents update operation rate impact on read-side performance. The goal of this section is to clearly demonstrate in which situation RCU outperforms classic locking solutions to help identifying for which workloads RCU can bring performance improvements compared to classic locks 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 PowerPC POWER5+ clocked at 1.9 GHz. Each core of the PowerPC machine has 2 hardware threads. To eliminate thread-level contention for processor resources, benchmarks are performed with affinity to the 64 even-numbered CPUs of the 128 logical CPUs presented by the system.

The mutex and reader-writer lock implementations used for comparison are the standard pthreads 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 it is already known to incur high overhead and to scale poorly (17). HTM (Hardware Transactional Memory) (18; 19; 20) is likely to be more scalable than STM. However, HTM hardware is not available to us due to the fact that it is expensive and not very common, preventing us from including it in our performance results.

A. Scalability

Figure 12 presents the read-side scalability comparison of each RCU mechanism with standard locking primitives for the PowerPC. The goal of this test is to determine how each synchronization primitive performs in heavy read-side scenarios when the number of CPU increases. This is done by executing from 1 to 64 reader threads for 10 seconds, each taking a read-lock, reading a data unit and releasing the lock in a tight loop. No updater thread is present in this test. As a result, we observe that linear scalability is achieved for RCU and per-thread mutex approaches. This is expected, given readers does not need to exchange cache-lines. The QSBR approach is the fastest, followed by the signal-based RCU, general-purpose RCU and per-thread mutex, each adding a constant per-CPU overhead. The Intel Xeon behaves similarly.

However, Figure 12 does not show the scalability trend of the pthread mutex and pthread reader-writer lock primitives. This is the purpose of Figure 13, which presents scalability of those two primitives. As we can see, with more than 8 cores, overall performance actually decreases when the number of core increases.

B. Read-Side Critical Section Length

Due to the large performance difference between RCU and other approaches, we notice that linear-scaled graphs are not appropriate for the following comparisons.

[1]The per-thread lock approach consists in using one mutex per reader thread. The updater threads must take all the mutexes, always in the same order, to exclude all readers. This approach ensures reader cache locality at the expense of a slower write-side locking.

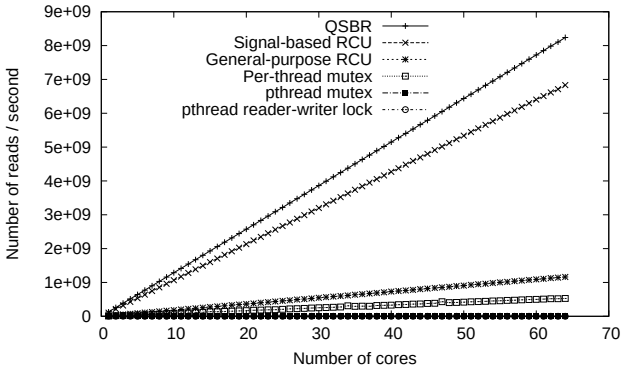

Fig. 12. Read-Side Scalability of Various Synchronization Primitives, 64-core POWER5+

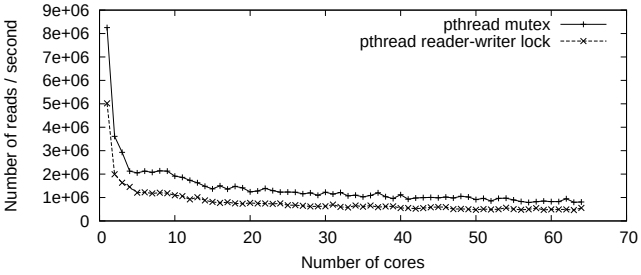

Fig. 13. Read-Side Scalability of Mutex and Reader-Writer Lock, 64-core POWER5+

Therefore, Figure 14 presents the read-side critical section length impact using logarithmic x and y axis. This benchmark is performed with 8 reader threads taking the read lock, reading the data structure, waiting for a variable delay and releasing the lock, without any active updater. Interestingly, on this 8-core machine, we notice that starting at about 1000 cycles per critical section, the difference between RCU and per-thread locks becomes insignificant. At 20000 cycles per critical section, the reader-writer locks are almost as fast as the other solutions. Only pthread mutex performance always has significantly worse performance for all critical section lengths.

To appropriately present the 64-core read-side critical section length impact on the read-side speed, we must first introduce the effects that alter the reader-writer lock and mutex

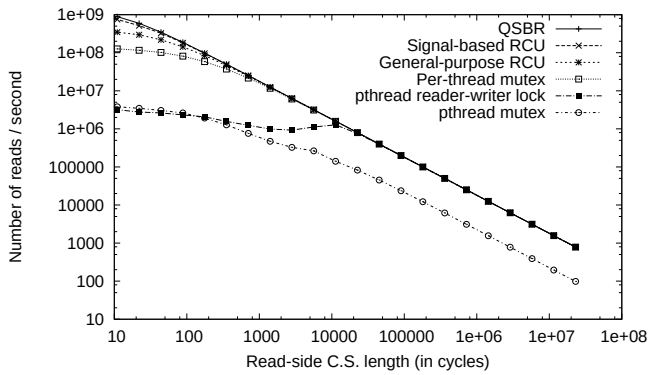

Fig. 14. Impact of Read-Side Critical Section Length, 8-core Intel Xeon, Logarithmic Scale

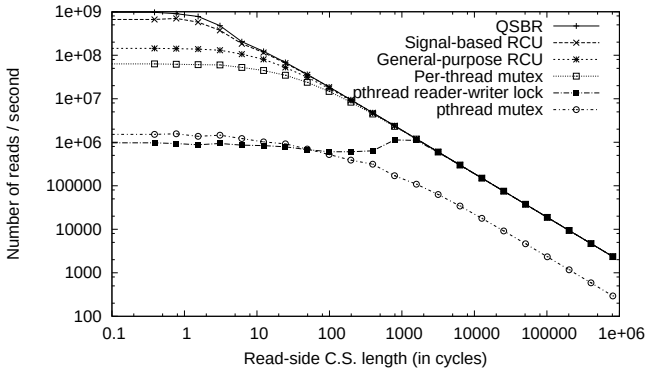

Fig. 15. Impact of Read-Side Critical Section Length, 8 Reader Threads on POWER5+, Logarithmic Scale

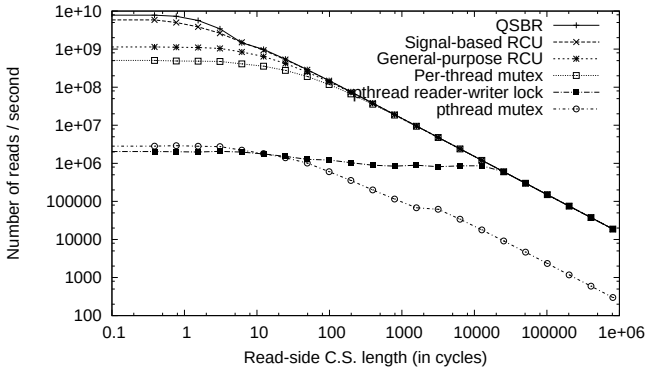

Fig. 16. Impact of Read-Side Critical Section Length, 64 Reader Threads on POWER5+, Logarithmic Scale

behavior. First, the interprocessor cache-line exchange time affects the lock access time. Second, the number of cores needing to access the lock also affects the number of lock-access per second.

Therefore, we first present, in Figure 15, the equivalent POWER5+ graph with only 8 cores used to specifically show the effect of architecture and cache-line access time change. The cores are spaced by a striding of 8. Changing stride to 1, 2 or 4 (not presented here for brevity) only very slightly affects read speed for reader-writer lock and mutex. Cores close to each other share a common L2 and L3 cache on the POWER5+, which causes reader-writer lock and mutex to be slightly faster at lower striding values. Given it has no significant effect on the update rate at which the various locking primitives are equivalent, this factor can be left out of the rest of this study.

The same workload executed with 64 reader threads is presented in Figure 16. These threads are concurrently reading the data structure with an added variable delay. We notice, when comparing to the 8-core graph in Figure 15, that we need a critical section about 10 times larger (20000 instead of 2000 cycles) before the reader-writer lock performance reaches the RCU or per-thread lock performances. Therefore, as we increase the number of cores, reader-writer lock protected critical sections must be larger to behave similarly to RCU and per-thread locks.

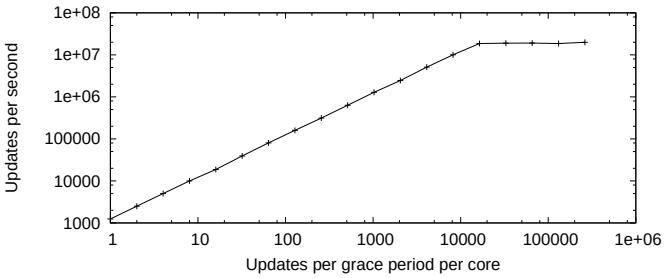

Fig. 17. Impact of Grace-Period Batch-Size on Number of Update Operations, 8-core Intel Xeon, Logarithmic Scale

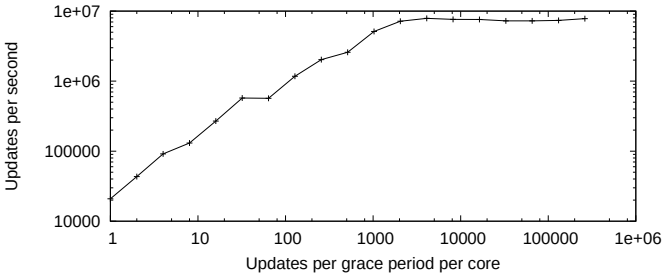

Fig. 18. Impact of Grace-Period Batch-Size on Number of Update Operations, 64-core POWER5+, Logarithmic Scale

C. RCU Grace-Period Batch Calibration

After looking at read-side only performance, it is appropriate to see how concurrent updates influence the read-side behavior. To appropriately represent the RCU update-side performance impact, we must first calibrate the reclamation batch size to ensure we amortize the grace-period overhead over multiple updates. Such calibration is presented for Intel and POWER5+ in Figures 17 and 18, respectively for 8 cores and 64 cores. For update operation benchmark, we use half the number of cores for readers and the other half for updaters.

We calibrate with the signal-based RCU approach, likely to provide the highest grace-period overhead due to signal-handler execution. The ideal batch size for both architectures with 8 cores used is determined to be 32768 per updater thread. Given the test duration is 10 sec, we have to eliminate batch sizes large enough to be a significant portion of updates performed during the test because non-reclaimed batches are not accounted for. This is why the largest batch sizes are ignored even if they seem slightly better. Figure 18 shows that with 64 cores used, the ideal batch size is slightly lower (4096) due to the fact that per-update pointer exchange overhead increases exponentially with the number of threads while the grace-period overhead increases linearly. Therefore, smaller batch sizes are required to amortize the grace-period overhead and perform slightly better due to increased cache locality. However, given the performance difference is not very large, we use a 32768 batch size for both 8-core and 64-core tests.

D. Update Overhead

Once batch-size calibration is performed, we can proceed to update rate impact comparison. Figure 19 presents the impact of update frequency on read-side performance for the various

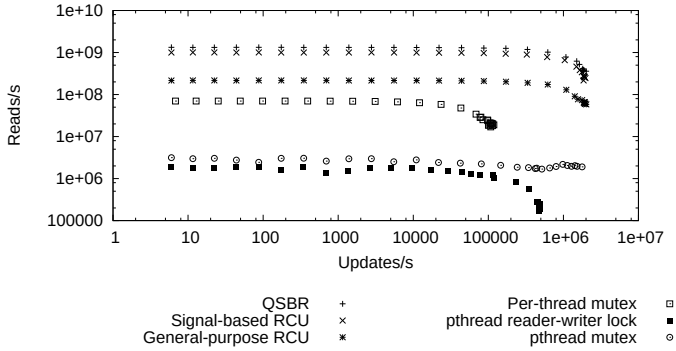

Fig. 19. Update Overhead, 8-core Intel Xeon, Logarithmic Scale

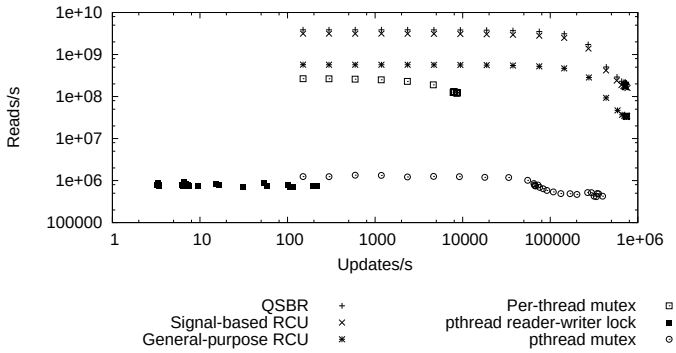

Fig. 21. Update Overhead, 64-core POWER5+, Logarithmic Scale

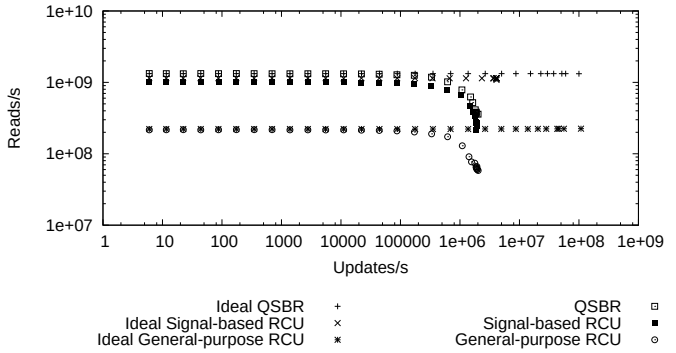

Fig. 20. Impact of Pointer Exchange on Update Overhead, 8-core Intel Xeon, Logarithmic Scale

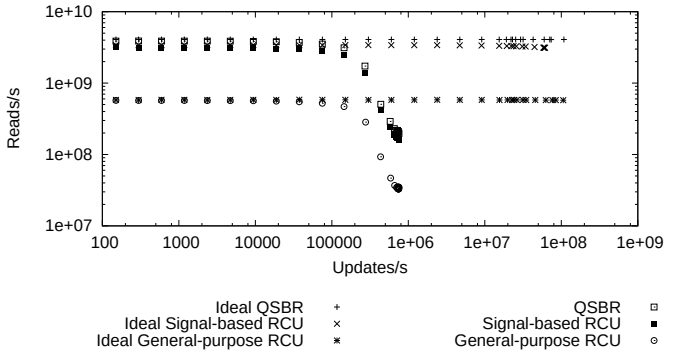

Fig. 22. Impact of Pointer Exchange on Update Overhead, 64-core POWER5+, Logarithmic Scale

locking primitives. It is performed by running 4 reader and 4 updater threads and varying the delay between updates. We notice that RCU approches outperforms the per-thread lock approach especially in terms of maximum updates per second. The former can reach 2 million updates per second while per-thread locks can only perform 0.1 million updates per second. Interestingly, on such workload with 4 tight loop readers, mutexes outperforms the reader-writer lock primitive in all aspects. Furthermore, reader-writer locks seems to show a case of reader starvation with high updates per second rates.

But while Figure 19 presents a fair comparison between the locking primitives, it presents a non-ideal scenario for RCU. In our attempt to present a comparison where all locking primitives perform equivalent work, this figure includes the RCU pointer exchange overhead. To factor out this overhead, Figure 20 shows both ideal RCU grace-period performance and the equivalent with added pointer exchange. This shows that it is the pointer exchange that becomes the update-rate bottleneck, not the grace period. Therefore, in an ideal scenario where pointers updates would be local to each thread, RCU could be expected to preserve its read-side scalability characteristics even under frequent updates. Such local updates could be ensured by appropriately designed list or hash table data structures.

Figure 21 shows update overhead on a 64-core POWER5+, with 32 reader and 32 updater threads. We can conclude that RCU QSBR and general purpose approaches reach the highest update rates, even compared to mutexes. 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 mutex. Mutex-based benchmark performance seems to drop starting at 30000 updates per second with 32 updater threads. A similar effect is present with only 4 updater threads (graph not presented for brevity). Figure 19 seemed to show that update overhead stayed 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. Two approaches seems to be very affected by increasing the number of updaters. The reader-writer lock, where updaters clearly seem to be starved by readers, has a maximum update rate of 175 updates per second. Per-thread locks are limited to a maximum update rate of 10000 updates per second with 32 reader threads.

Finally, Figure 22 presents, as previously done for Xeon, how grace-period detection (ideal RCU) compares to RCU grace period with pointer exchange. Therefore, with appropriately designed data structures, better update locality would ideally lead to constant updater overhead as the update frequency increases.

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 each reader thread to periodically pass through a quiescent state. Signal-based RCU performs almost as well as does QSBR, but requires reserving a 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 invoke an initialization function before entering its first RCU read-side critical section.

Benchmarks demonstrate read-side linear scalability of the RCU and per-thread lock approaches. It also shows that the smallest read-side critical section duration for which reader-writer locks, RCU and per-thread lock approaches are nearly equivalent in terms of read-side performance impact grows larger as the number of cores increases. These benchmarks also show that, by performing memory reclamation in batch, RCU approaches reach update rates much higher than reader-writer locks, per-thread locks and mutexes on similar workloads where updates are performed on a shared data structure. Furthermore, given ideal data structures preserving update cache locality, RCU approaches are shown to have a constant update overhead as update frequency increases. Therefore, the upper-bound for RCU update overhead is demonstrated to be far below lock-based overhead. Furthermore, it is still possible to decrease RCU update-side overhead even more by designing data structures providing good update cache-locality.

ACKNOWLEDGEMENTS

We owe thanks to Maged Michael for many illuminating discussions, to Kathy Bennett for her support of this effort, to Etienne Bergeron and Alexandre Desnoyers for reviewing this paper.

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

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.

[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] 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.

[7] K. A. Fraser, "Practical lock-freedom," Ph.D. dissertation, King's College, University of Cambridge, 2003.

[8] 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.

[9] P. E. McKenney, "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.

[10] ——, "What is RCU? part 2: Usage," January 2008, available: http://lwn.net/Articles/263130/.

[11] M. Desnoyers and M. R. Dagenais, "Synchronization for fast and reentrant operating system kernel tracing," To appear.

[12] E. Polyakov, "The elliptics network," April 2009, available: http://www.ioremap.net/projects/elliptics.

[13] 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.

[14] M. Herlihy, "Wait-free synchronization," *ACM TOPLAS*, vol. 13, no. 1, pp. 124–149, January 1991.

[15] P. E. McKenney, "Using a malicious user-level RCU to torture RCU-based algorithms," in *linux.conf.au 2009*, Hobart, Australia, January 2009.

[16] 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.

[17] 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.

[18] *A Scalable, Non-blocking Approach to Transactional Memory*, 2007.

[19] *Scalable and reliable communication for hardware transactional memory*, 2008.

[20] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commericial hardware transactional memory implementation," in *Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, Washington, DC, USA, March 2009, p. 12.