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.

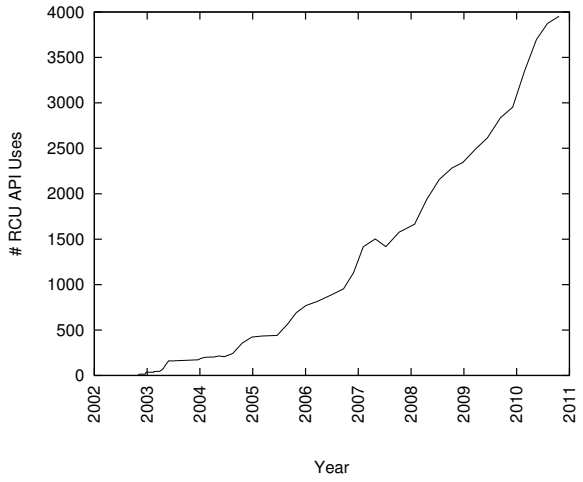


Fig. 1. Linux-Kernel Usage of RCU

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, with a definition of RCU semantics in Appendix C in the *Supplementary Material*. Then, Section III describes user-level scenarios that could benefit from

Manuscript received August 17, 2009; revised December 16, 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.

RCU. This is followed by the presentation of three classes of RCU implementation in Appendix D in the *Supplementary Material*. 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. Appendix A in the *Supplementary Material* presents a list of informal RCU desiderata, which details the goals pursued in this work. Appendix B in the *Supplementary Material* walks through an example real-time use of RCU. Finally, Appendix C in the *Supplementary Material* gives a semi-formal description of RCU semantics, including guarantees that allow RCU to operate correctly on systems that do not provide sequential consistency.

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, Appendix D2 in the *Supplementary Material* shows 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 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.

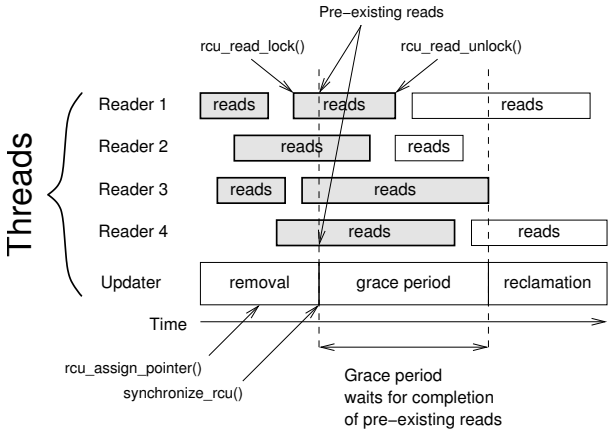


Fig. 2. Schematic of RCU Grace Period and Read-Side Critical Sections

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.

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-

[1] Interestingly 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].

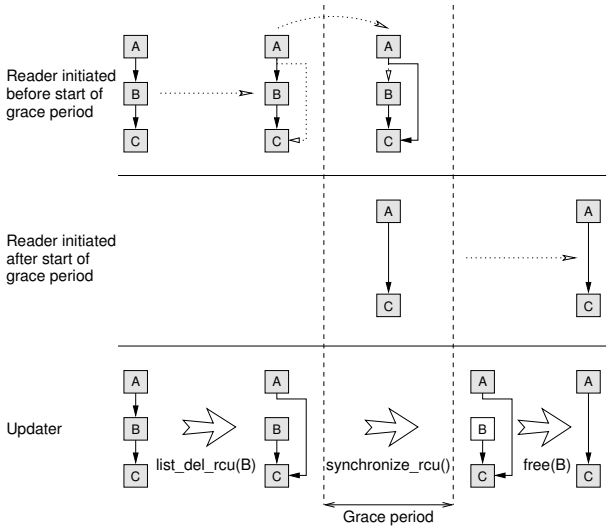


Fig. 3. RCU Linked-List Deletion

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].

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 [19, 20]. 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 is presented in Appendix D2 in the *Supplementary Material*. 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 [21]. 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 (Summary)

Appendix D in the *Supplementary Material* describes several classes of RCU implementations. Appendix D1 first describes some primitives that might be unfamiliar to the reader, and then Appendices D2, D3, and D4 present user-space RCU implementations that are optimized for different use cases. The QSBR implementation presented in Appendix D2 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 general-purpose implementation presented in Appendix D3 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. Appendix D4 presents an implementation having low read-side overhead and requiring only that the application give up one POSIX signal to RCU update processing, and is called the signal-based implementation. Finally, Appendix D5 demonstrates how to create non-blocking RCU update primitives.

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 [22]. 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 [23]. 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 9 and 10 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 [24], albeit in some cases by placing the burden of instrumentation and privatization on the developer [25]. HTM (Hardware Transactional Memory) [26, 27, 28] is likely to be more scalable than STM; unfortunately, no system supporting HTM was available for this study.

A. Read-Side Scalability

Figure 4 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. QSBR 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 QSBR 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 4, the traces for pthread mutex and pthread reader-writer locking cannot be easily distinguished from the x axis. Figure 5 therefore displays only these two traces, showing their well-known negative scalability.

B. Read-Side Critical Section Duration

Figure 6 presents the number of reads per second as a function of the duration in nanoseconds of the read-side critical

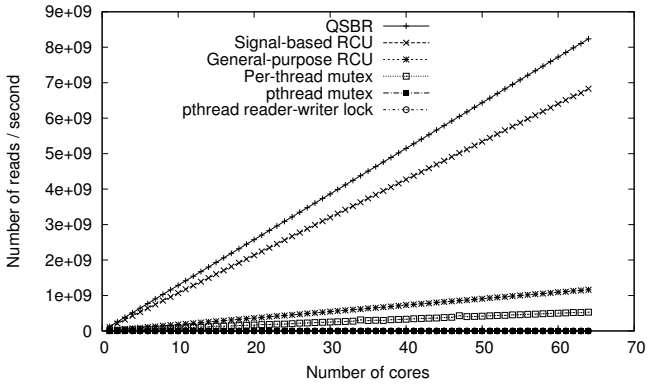


Fig. 4. Read-Side Scalability of Various Synchronization Primitives, 64-core POWER5+

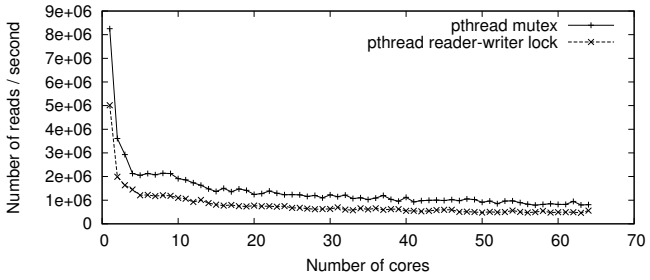


Fig. 5. Read-Side Scalability of Mutex and Reader-Writer Lock, 64-core POWER5+

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.

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 6–8 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 7 and 8. The difference between them is that Figure 8 uses 64 reader threads and 64 cores, whereas Figure 7 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

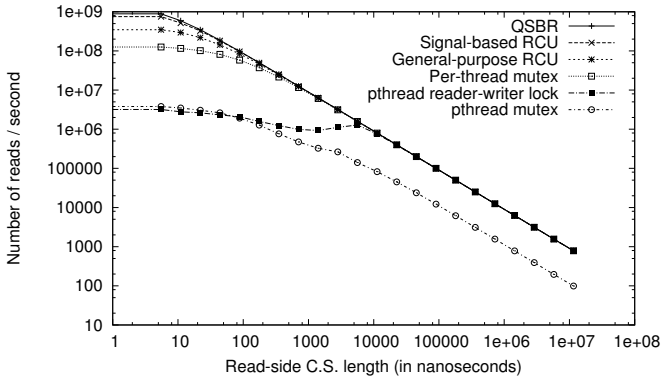


Fig. 6. Impact of Read-Side Critical Section Length on 8-core Xeon, Logarithmic Scale

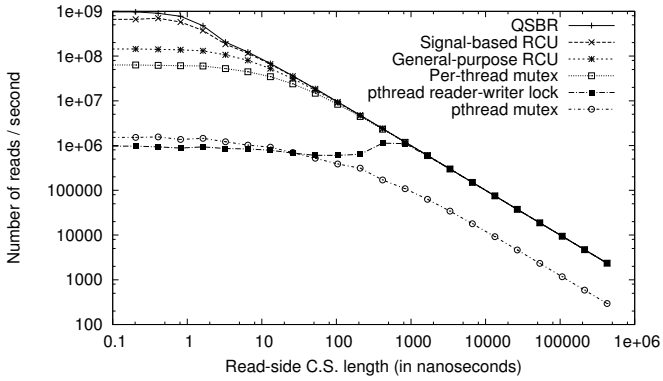


Fig. 7. Impact of Read-Side Critical Section Length, 8 Reader Threads on 64-core POWER5+, Logarithmic Scale

mutex to be slightly faster at lower stride values (not shown). This has no significant effect on our results.

Comparing Figures 7 and 8 shows that the read-side overheads of both the reader-writer lock and the pthread mutex 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

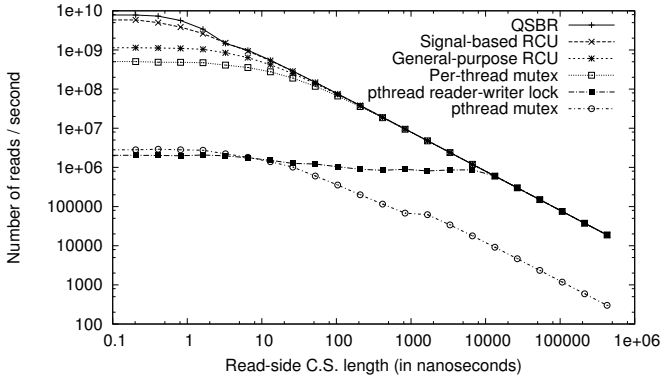


Fig. 8. Impact of Read-Side Critical Section Length, 64 Reader Threads on 64-core POWER5+, Logarithmic Scale

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 6, 7, and 8 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.

Figure 9 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 10 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 eventually suffers from reader starvation.

The RCU read-side performance shown in Figures 9 and 10 trails off at high update rates, the causes of which are presented in Appendix E of the *Supplementary Material*.

Figures 11 and 12 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

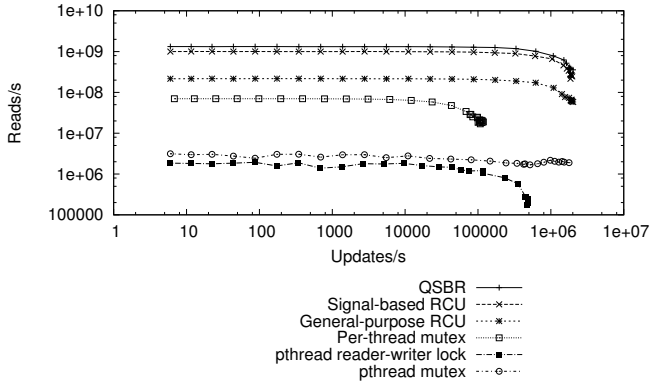


Fig. 9. Update Overhead, 8-core Intel Xeon, Logarithmic Scale

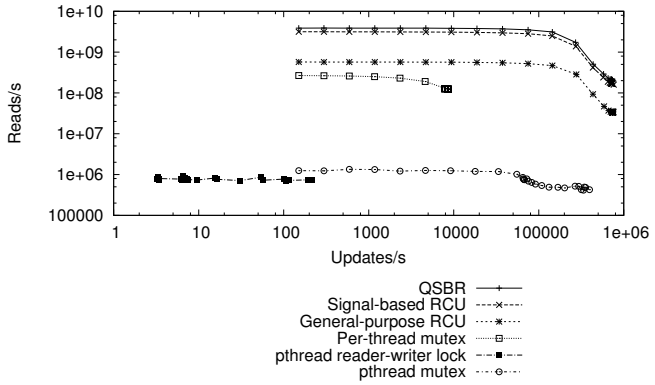


Fig. 10. Update Overhead, 64-core POWER5+, Logarithmic Scale

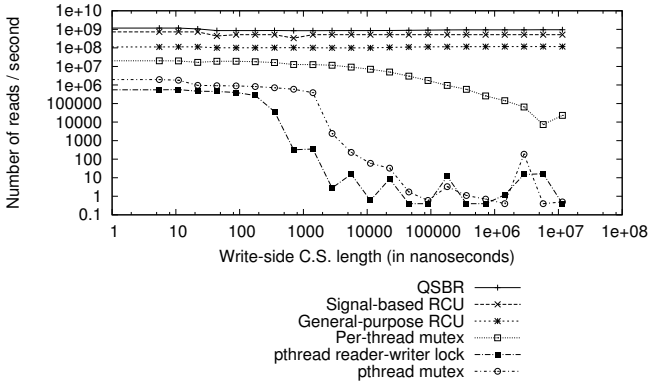


Fig. 11. Impact of Update-Side Critical Section Length on Read-Side, 8-core Intel Xeon, Logarithmic Scale

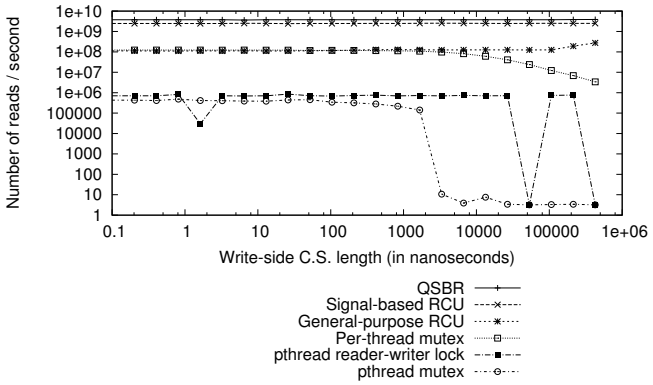


Fig. 12. Impact of Update-Side Critical Section Length on Read-Side, 64-core POWER5+, Logarithmic Scale

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 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 11 and 12, 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 12) 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 9 and 10 by selecting the rightmost point of a given trace. For example, Figure 9 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 10, 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 9 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 10, 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 [29].

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. Signal-based RCU does not have this requirement, and 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 all the RCU implementations and of per-thread locking. However, they also demonstrate that the performance of the RCU implementations can exceed that of per-thread locking (and thus that of uncontended locking) by up to an order of magnitude, independent of the number of threads. The benchmarks 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. In addition, performing grace-period detection in batch allows RCU to attain 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 so as to provide good cache locality for updaters.

Acknowledgments

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

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 multi-

processor 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] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: streamlining STM by abolishing ownership records," in *PPOPP*, 2010, pp. 67–78.

[25] 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.

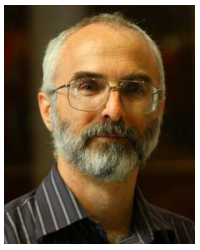
[26] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A scalable, non-blocking approach to transactional memory," in *HPCA Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 97–108.

[27] 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.

[28] 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.

[29] 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).