A Non-Intrusive Read-Copy-Update for UTS

Andrej Podzimek, Martin Děcký, Lubomír Bulej, Petr Tůma
Department of Distributed an Dependable Systems
Faculty of Mathematics and Physics
Charles University in Prague
{podzimek, decky, bulej, tuma}@d3s.mff.cuni.cz

***Abstract*—Read-Copy-Update (RCU) is a mechanism designed to increase the level of concurrency in readers-writer synchronization scenarios, vastly improving scalability of software running on multiprocessor machines. Most existing RCU variants have been developed for and studied within the Linux kernel. Due to strong dependency on the Linux internals, they cannot be easily transferred to other operating system kernels. This paper presents a novel non-intrusive variant of the RCU mechanism (AP-RCU), which depends only on basic kernel-level concepts while maintaining the scalability benefits. We have implemented AP-RCU in the Solaris kernel (UTS) and experimentally confirmed the expected benefits over traditional forms of synchronization, comparable with previous RCU implementations.**

***Keywords*-RCU; read-copy-update; UTS; Solaris; SMP; synchronization; scalability**

I. Introduction

In contemporary SMP systems, traditional synchronization mechanisms (e.g. locking) do not scale well with the increasing number of processors and parallel activities, given the latency involved in accessing shared memory. Reasons for the scaling issues include the use of memory barriers and atomic instructions (which induce heavy cache coherence traffic) [6], and the scheduling artifacts of mutual exclusion (serialization of non-conflicting operations, convoying, etc.) [3].

The *Read-Copy-Update* (RCU) is a data access synchronization mechanism designed to improve SMP and massive MP scalability under asymmetric (readers-writers) data sharing. Unlike readers-writer (R/W) locks that isolate one writer from (possibly concurrent) readers, RCU allows writers to run in parallel with readers. Readers never need to block when entering their critical sections, thus read-side critical sections can occur even in contexts where blocking is not permitted. Moreover, readers can mostly avoid expensive memory operations (barriers and atomic instructions), which may improve performance even on uniprocessor systems [6].

To achieve both correctness and efficiency, RCU algorithms need to consider many technical properties of the target platform, such as the memory model and the scheduler behavior. This is far from trivial, as illustrated by Linux – since the introduction of RCU into the Linux kernel in 2002, at least four distinct RCU algorithms were used. The drawback of these algorithms is that they rely on properties particular to the Linux kernel.

The contribution of this paper is twofold. First, we present our novel non-intrusive variant of RCU (designated AP-RCU), which depends only on basic kernel-level concepts and explicitly avoids dependency on clock tick handling and scheduler code modification. Second, we provide a proof-of-concept implementation of AP-RCU in the Solaris kernel (UTS) and present measurements showing robust performance, improving concurrency in RCU-friendly scenarios, and keeping up with conventional solutions in RCU-unfriendly scenarios.

The rest of the paper is organized as follows. To provide more context, Sections II and III characterize the two most common classes of RCU. The design of AP-RCU is outlined in Section IV. Section V describes the benchmarking environment, with benchmark results presented in Section VI. Section VII provides an overview of closely related work and future directions, and Section VIII concludes the paper.

II. RCU Overview

RCU enables parallel execution of readers and writers by providing the readers with an illusion of data integrity. The illusion is achieved by using *atomic assignments* and *deferred destruction* to keep multiple copies ("versions") of the protected data "alive" for as long as the readers access them. This reduces the readers' overhead at the expense of writers, which is suitable for read-mostly data structures [14]. Migrating parallel algorithms from mutual exclusion to RCU mostly requires adjustments to handle stale data gracefully [6].

RCU can be perceived as a means of synchronization among three types of entities: *readers*, *writers* and *reclaimers*. The roles of readers and writers can be assumed by threads, processors or other scheduling entities, depending on the approach. Reclaimers are an internal mechanism of some RCU implementations and will be described later.

Readers access RCU-protected data in such a way that multiple readers can operate in parallel. Readers are guaranteed not to observe any changes to the shared data made by writers during their read-side critical sections, as long as both readers and writers adhere to certain data access rules. For instance, readers are not allowed to change the direction of doubly linked list traversal within a read-side section (more generally, they are not allowed to load certain pointers from main memory more than once per critical section).

Writers modify the protected data and, in contrast with standard R/W locks, operate in parallel with readers. However, they must cooperate with the RCU mechanism to provide the readers with the illusion of data integrity. Modifying protected data thus comprises loading a pointer referencing

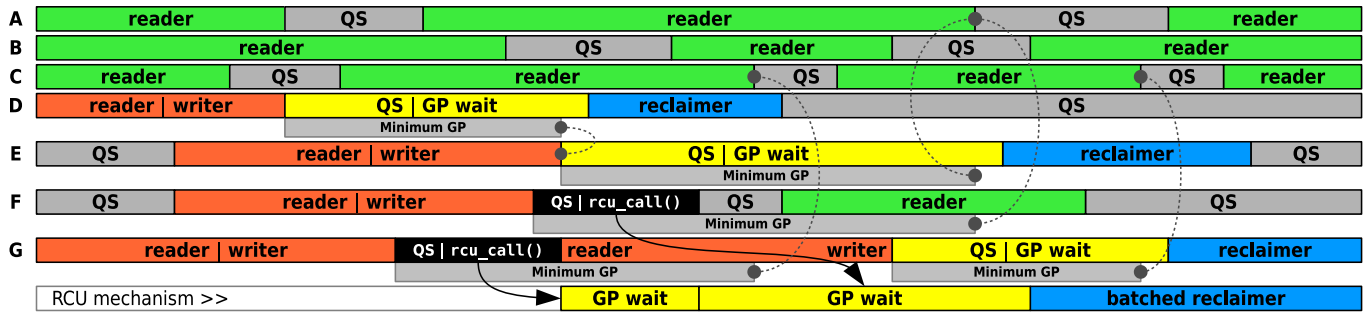

Fig. 1. Example RCU Scenario (QS = Quiescent State, GP = Grace Period)

data ("Read"), copying the data to a new location ("Copy"), modifying the copy and finally assigning the address of the new data to the original pointer ("Update"). Using this method, writers can modify data of arbitrary size atomically (as observed by readers). Since word-aligned pointer assignments are atomic on virtually all current hardware [8], readers will always see consistent data — they will either read the old pointer value and work with the old "*stale*" data item, or obtain the new pointer value, accessing the new data. In either case, they will not observe any changes during their operation.

Multiple writers running in parallel must synchronize outside the RCU mechanism, using either standard synchronization primitives, or non-blocking algorithms.

Reclaimers handle the *deferred destruction* part of RCU by reclaiming the resources (mostly allocated memory) made obsolete by writers. The reclamation has to be postponed until no more readers operate on the *stale* data (the data no longer accessible from the RCU-protected data structures). To release resources after disconnecting a portion of a linked data structure, the writers can either a) enqueue a callback handler to be invoked by the RCU reclamation mechanism at a suitable (later) time and continue immediately, or b) use the RCU mechanism to block for a sufficient period of time and perform the reclamation afterwards.

The "sufficient period of time" to guarantee that all readers operating on a stale data item have finished is called a *Grace Period* (GP). The key responsibility of RCU is to determine when stale data can be safely reclaimed, which requires estimating the upper bound of the grace period duration.

Grace period detection is the key part of all RCU implementations and is closely related to the basic idea behind RCU — "waiting for things to finish" [15]. The GP detection mechanism must be both correct (never ending a grace period with readers still accessing stale data) and efficient (providing an upper bound on grace period duration as long as the duration of all possible read-side critical sections is bounded).

To illustrate GP detection, Figure 1 shows a scenario with eight scheduling entities (either threads or processors, details will follow in Sections III-A and III-B) using the RCU mechanism. The entities A, B, and C run RCU readers in parallel (with both writers and other readers) and are guaranteed to never block. The periods between read-side critical sections are called *Quiescent States* (QS).

A *Quiescent State* of a reader is a moment when the reader does not keep local copies (in CPU registers or local variables) of RCU-protected data — more precisely, all possible local copies are stored in *dead* variables, i.e. variables reloaded from memory before subsequent access. In this context, an interval during which all potential readers pass through a quiescent state at least once is certainly a grace period. As the order of quiescent state occurrence or observation does not matter, grace period intervals may overlap.

Since writers always assign pointers to new versions of protected data before waiting for a grace period and releasing the old versions, new readers that start during a grace period need not be waited for (they can only reference the new version of the protected data).

For example, the writer in the RCU scheduling entity D in Figure 1 modifies shared data, waits for a grace period and then enters the reclamation phase, releasing old versions of the data. There is no need to wait for any reader working in entity A, since neither of them was active at the beginning of the grace period. Similarly, a quiescent state also occurred in the scheduling entity C, which guarantees that all of its subsequent readers will only see the new data. As for entity B, it is running a read-side critical section that must end before the end of entity D's grace period. Similarly, readers running in entities E, F and G (as part of readers-writers) must be waited for. The last read-side section to finish is in entity E. Its end is the lower bound on grace period termination time for the observed writer-to-reclaimer transition in entity D.

The entity at the bottom of Figure 1 represents the RCU reclamation mechanism. Writers in the entities D and E block for (at least) the duration of one grace period and then perform reclamation on their own (within the same scheduling context). In contrast, writers in the entities F and G use a non-blocking approach — they provide RCU with a reclamation callback and proceed immediately, relying on the RCU reclamation mechanism to invoke the callback when appropriate.

In Figure 1, the shortest possible grace period is indicated for each writer, under the corresponding scheduling entity time-line. For each minimum grace period, grey dotted lines identify the last reader that had to be waited for. Most RCU implementations exploit the fact that any extension of a grace period is also a grace period. Reclamations therefore usually occur later than after the shortest grace period. This makes grace period detection more efficient at the expense of higher latency, because precise detection of all possible grace periods

may be extremely costly in practice.

Writers that do not block and let the RCU mechanism handle deferred destruction requests illustrate two important facts (see entities F and G in Figure 1): First, writers using the non-blocking RCU facilities can be followed by other writers and readers in the same scheduling entity (execution context) immediately, with no delay related to reclamation. Second, reclamation requests can be postponed and handled in batches by the RCU implementation.

III. RCU Algorithm Classification

There are two main classes of RCU algorithms, depending on the scheduler entities they track as potential readers. Most system-wide algorithms are based on the *Processors as Readers* (PaR) approach. Some algorithms track individual threads, leading to the *Threads as Readers* (TaR) approach. A more detailed classification of the RCU algorithms and a summary of their characteristics can be found in [22].

A. Processors as Readers

When processors (not individual threads) are considered as potential readers, there is only one global instance of the RCU mechanism, handling grace period detection requests for the whole system. Since there is no notion of processors in user space and no unified programming interface to control processors as scheduling entities, all RCU implementations tracking processors strongly depend on the kernel environment and cannot be easily deployed in user space. PaR-class algorithms are preferred when the highest achievable read-side efficiency and an efficient callback batching mechanism are required.

To guarantee an upper bound on grace period durations, the duration of all read-side critical sections must be bounded. This requires that they work correctly, run with disabled preemption (to eliminate thread scheduling interference), and neither block nor sleep.

Asynchronous grace period detection is easier to support (and more commonly found) with the PaR approach. Instead of blocking and waiting for a grace period, writers enqueue a callback that reclaims resources associated with stale data and continue execution. The callback invocation will be deferred by RCU until all potentially related readers have finished. Since all callbacks are handled by one central RCU mechanism, they can be efficiently batched or executed in parallel.

Figure 2 shows the blocking and non-blocking API of our AP-RCU implementation. This interface is typical for PaR class algorithms — since there is only one system-wide RCU facility, most of the functions do not take any arguments.

The functions *rcu_read_lock()* and *rcu_read_unlock()* delimit a read-side critical section. Callbacks that perform deferred reclamation and do not block or sleep can be enqueued using *rcu_call()*, while *rcu_call_excl()* allows enqueing callbacks that may need to block. The choice between performance and flexibility is up to the user and will be discussed in Section IV. Callbacks are invoked after at least one grace period.

```
rcu_read_lock();
rcu_read_unlock();
rcu_call(callback, argument, weight);
rcu_call_excl(callback, argument, weight);

rcu_synchronize();
rcu_synchronize_shared(max_wait);
rcu_call_synchronize();
```

Fig. 2. Non-blocking (upper) and blocking (lower) "PaR" API functions

B. Threads as Readers

When threads are considered as potential readers, the read-side overhead is slightly higher (due to explicit bookkeeping) and readers are required to actively cooperate with writers using expensive memory operations. As threads can be managed without kernel environment dependencies, some of the TaR algorithms are easy to port and can be deployed in user space.

Readers are allowed to block and sleep for a virtually unlimited time in TaR algorithms. Therefore an upper bound on grace period duration cannot be easily estimated. Consequently, callbacks cannot be batched, for it is necessary to eliminate the possibility of accumulating an excessive number of waiting callbacks. Most TaR algorithms therefore do not support asynchronous grace period detection.

The key advantage of the TaR approach is the ability to create multiple instances of the RCU mechanism, similar to standard synchronization primitives. This can partially compensate for virtually unbounded grace periods sleeping readers may cause. Misbehaving readers (sleeping for a long time) bound to an RCU instance will not block other RCU instances.

```
int qrcu_read_lock(qrcu_t *qrcu);
void qrcu_read_unlock(qrcu_t *qrcu, int index);
void qrcu_synchronize(qrcu_t *qrcu);
```

Fig. 3. "TaR" API functions

To illustrate the difference between PaR and TaR algorithms, Figure 3 shows the API of our port of QRCU, a TaR algorithm originally designed for Linux [11].

In contrast to the AP-RCU API in Figure 2, the QRCU API is designed to work with multiple instances, but lacks support for asynchronous grace period detection and reclamation — there is no equivalent of the *rcu_call()* and similar (non-blocking) functions. The writers can only wait for a grace period using the blocking function *qrcu_synchronize()*.

IV. AP-RCU: Designing a Non-Intrusive RCU

Most current operating system kernels (including Linux and UTS) are heading towards a fully "tickless" kernel design using on-demand and possibly irregular timer interrupts. It is therefore preferable not to base future RCU algorithms around clock tick handlers. Also, there are substantial differences between the systems as far as clock tick handling is concerned.

For example, the Linux kernel handles clock interrupts on each running processor, while the UTS kernel only had

a single processor handling clock interrupts in Solaris 10 [9]. Multiprocessor clock interrupt handling has been introduced by OpenSolaris. However, only a limited subset of available processors (one per NUMA hardware node, for instance) takes part in clock interrupt handling.

The most frequently used RCU algorithms on Linux depend on clock tick handlers executing on every processor. When RCU was added to Linux, clock tick handler code was instrumented to perform some tasks critical to RCU operation, such as grace period detection. Since UTS does not handle clock interrupts on all processors, it is virtually impossible to port the core Linux RCU algorithms to UTS. Making the Linux RCU implementations compatible with the "tickless" processor idle states required much more code (and effort) than the original RCU algorithm implementation [18].

Moreover, the most frequently used RCU algorithms implemented in the Linux kernel require some data processing, such as quiescent state announcements, to be handled by the scheduler code on every context switch. However, the UTS kernel can (and in certain situations does) assign time quanta in the order of seconds. For this reason, we believe that instrumenting scheduler code to perform RCU-related computations would do more harm than good — adding overhead when rescheduling is frequent and causing latency issues when long time quanta occur.

Since porting of the core Linux RCU algorithms to other systems does not seem to be feasible, we have decided to design a new RCU variant so as to explicitly avoid dependency on clock tick handling and scheduler code modification.

To provide a complete set of RCU facilities, a kernel should also support a TaR class RCU algorithm. In general, TaR algorithms are more portable than PaR algorithms. To complement AP-RCU (a PaR class algorithm), we have also ported QRCU[1] (a TaR class algorithm) from a (rejected) patch against Linux source code to the UTS kernel. Details concerning our implementation of QRCU can be found in [22].

All the new RCU functionality in UTS is implemented using only standard kernel threads and synchronization primitives. No modifications to the scheduler or clock interrupt handler code were needed. This guarantees that migrating the whole UTS kernel to tickless scheduling would not require any additional work related to RCU.

The following sections describe the key points of our implementation of AP-RCU for UTS. It relies on many technical subtleties to properly support complex real-life situations (e.g. dynamic CPU on-lining and off-lining). Readers interested in the details are strongly encouraged to read [22] and the commented C source code. The complete source code patch against recent Illumos source tree[2] (including the kernel benchmarks described later in this paper) can be downloaded from [23]. The QRCU implementation will not be described here for the sake of brevity — we refer the interested reader to McKenney's original description [11], [12].

[1]Note that our AP-RCU (a PaR class algorithm) is not based on QRCU (a TaR class algorithm) or any other existing RCU implementation.

[2]`hg clone ssh://anonhg@hg.illumos.org/illumos-gate`

A. Grace Period Detection in AP-RCU

A grace period boundary can be announced as soon as all processors (potential readers) are known to have passed through a quiescent state since the preceding grace period boundary. Because PaR class RCU algorithms do not allow preemption to occur inside read-side critical sections, context switches represent naturally occurring quiescent states.

Comparing the context switch counter (maintained by the UTS kernel for every processor) with a copy of the counter sampled in the past is one of the means of quiescent state detection. Context switches are not the only naturally occurring quiescent states; processors running the *idle thread* or user space code are also in a quiescent state.

In AP-RCU, a system-wide *detector* thread checks the state of active processors. Once it observes a quiescent state on all of them, it announces a grace period boundary by waking up threads that wait for it.

All non-blocking RCU API functions assist the AP-RCU mechanism in quiescent state detection when used frequently. AP-RCU maintains a global counter of grace periods, incremented by the detector thread on each grace period boundary. Non-blocking RCU functions compare the value of the global grace period counter with a copy of this counter local to the processor they run on at the moment. If the two values differ, the local copy is updated to the value of the global counter and a memory barrier instruction is executed on the current processor. The memory barrier prevents RCU readers from accessing stale data before updating their counter copy.

Without the memory barrier, a stale data access could occur after making the local counter copy update visible to other processors, including the one on which the detector runs at the moment. This may happen due to reordering of memory accesses on some CPU architectures and may result in premature grace period boundary announcements [13]. In such improbable, but possible cases, the absence of memory barriers might cause the RCU mechanism to fail, possibly compromising the readers' illusion of data integrity. Thanks to the "on-demand" memory barrier, the detector thread can rely on the per-CPU copies of its grace period counter and use them to identify processors that have already passed through a quiescent state since the last grace period boundary.

When there is no grace period detection activity, the detector thread sleeps and the global grace period counter does not change. Under such circumstances, all RCU read-side critical sections proceed without memory barriers. Avoiding expensive memory barriers is vital for performance when designing a parallel algorithm [6].

When the number of pending callbacks surpasses a "high-water mark" or when quiescent states are not observed on some processors within a time interval, the detector thread forcibly migrates itself onto the processors in question. This causes context switches (which rank among quiescent states) to occur on these processors as soon as possible. (The priority of the detector thread is high enough to preempt most other threads instantly.) Natural quiescent state conditions (values

of context switch counters, grace period counter copies, "running" idle threads, etc.) are reevaluated before each forced context switch in order to avoid unnecessary preemption. As a result, observations confirmed that the detector does not need to initiate forced context switches in most cases.

The possibility to induce quiescent states provides an upper bound on grace period duration, as long as all read-side critical sections (and other code fragments that run with disabled preemption) take a bounded amount of time. Not all RCU implementations have this feature.

B. Callback Handling in AP-RCU

Each processor runs a bound *reclaimer* thread that handles callback processing. Pending callbacks can be either processed directly by the reclaimer thread or offloaded to a dynamic thread pool. Callbacks processed by the thread pool (enqueued using the *rcu_call_excl()* API function) can sleep or block ("blocking callbacks"), whereas callbacks processed by the reclaimer threads (enqueued using the *rcu_call()* API function) must run continuously ("non-blocking callbacks").

As long as callbacks are pending, reclaimer threads use standard means of synchronization (e.g. condition variables) to keep the detector thread active and wait for grace period boundaries to occur. On each grace period boundary, the detector thread wakes up the waiting reclaimer threads (potentially many of them at once), announcing the grace period progress. On each such announcement, the reclaimer threads handle callbacks that have been waiting long enough.

Reclaimers maintain two callback queues — callbacks waiting for the next boundary and callbacks waiting for the subsequent one. This allows them to process callbacks on each grace period boundary and still guarantee that each callback has spent at least two grace period boundaries (a full grace period) waiting before invocation.

Non-blocking callbacks are executed in batches by the same processor that created them, assuming that the processor's local cache may still contain related data.

Blocking callbacks are more flexible, at the expense of much higher dispatching overhead. They are not bound to a processor and they run in independent threads that can migrate freely among processors. Blocking callbacks are offloaded to an asynchronous thread pool for execution. If the offloading fails due to thread pool contention, the reclaimer attempts to process all non-blocking callbacks first, and finishes offloading of the blocking callbacks afterwards, this time waiting until the thread pool becomes available.

Threads outside AP-RCU can synchronously wait for grace period boundary announcements from the detector thread, using exactly the same mechanism as reclaimer threads. This is how the blocking RCU API is implemented. Two grace period boundaries have to be waited for to ensure that a full grace period has elapsed.

V. Benchmarking AP-RCU in UTS

Since RCU is not a drop-in replacement for traditional synchronization, evaluating the benefits of RCU in a particular operating system would require converting a selected subsystem to use RCU and compare its performance against the non-RCU version under real-life workload. Unfortunately, modifying a complex kernel subsystem (networking, file systems) is a time-consuming task we currently cannot afford to undertake. However, since the benefits of RCU in Linux have been thoroughly investigated [3], [6], we believe that RCU can be reasonably expected to improve scalability of similar subsystems in the UTS kernel if the RCU implementation can be shown to possess qualities essential to RCU, i.e., to improve scalability over traditional synchronization.

We therefore aim to show that AP-RCU running in the UTS kernel can improve scalability over traditional synchronization in reader-dominated workloads. Our selection of workloads is motivated by the existing (Linux) applications of RCU in the directory name lookup cache and the routing tables. Both are structures that are indexed by keys and mostly read. The directory name lookup cache has to cope with frequent concurrent lookups and writes into colliding items, while the routing table handles mostly lookups in face of occasional updates that should never block readers.

We approximate the implementation of such subsystems by a general shared non-blocking hash table implemented using multiple RCU variants and subject it to workloads with varying ratios of readers to writers. We then compare the performance of the hash table implemented using AP-RCU to the performance of hash tables implemented using two other RCU variants, the QRCU algorithm ported from linux, and a "dummy" implementation of the RCU API (DRCU), which uses an R/W lock internally. While the DRCU "algorithm" lacks the key features expected in a sound RCU implementation (e.g. it does not guarantee that readers never block) [16], it is still useful for comparison with the two "real" RCU implementations on the same workload — readers and writers run in parallel, using only the read-side of the internal lock, while the write-side of the lock is only taken and immediately released for the sake of grace period detection.

A. The Non-Blocking Hash Table

We have implemented the non-blocking hash table from scratch based on McKenney's description [6] of Marc Auslander's algorithm. The structure of the hash table is equivalent to a standard separate chaining hash table. Hash buckets are represented by an array of pointers referencing the beginnings of collision chains.

As long as hash table items are only retrieved and added, RCU is not needed. However, to make removals possible, all the three operations need RCU **read-side** protection. Furthermore, at least one grace period has to elapse between the removal of a hash table item and its re-use or disposal, to make sure there are no more readers accessing it.

When multiple threads add the same new key concurrently, only one of them can succeed. Similarly, when multiple threads remove the same existing key concurrently, only one of them can succeed. Meeting these constraints requires the atomic *Compare and Swap* (CaS) operation. Items representing removed keys must be atomically invalidated prior to the

actual removal. Manipulation with the collision chains also requires atomic operations.

Additionally, item removals must occur atomically with respect to item invalidations. This is achieved by encoding the item validity flag into the lowest order bit of the hash chain pointer. Readers interested in the subtleties of the algorithm are encouraged to read McKenney's description. Implementation details can be found in [22].

The hash table exists in five variants and the one used is determined by the benchmark scenario. The first variant is meant for a PaR-friendly scenario, where a single hash table instance is protected either by the global PaR class RCU, or a single instance of TaR class QRCU/DRCU. Presumably, the QRCU/DRCU instance is exposed to serious contention.

The four other variants, one for each RCU algorithm, are meant for a TaR-friendly scenario, in which the variants for TaR class algorithms use a separate QRCU/DRCU instance for each hash table bucket, while the variants for PaR class algorithms still use the global RCU mechanism. The vast number of TaR-class RCU instances per hash table ensures that there is virtually no contention on the individual instances.

B. The Benchmark Workload

The benchmark spawns a number of kernel threads that perform concurrent operations on a non-blocking hash table, protecting the data with RCU. The number of threads is proportional to the number of available hardware threads. Each benchmarking thread inserts and removes a set of keys (and values), repeating this work cycle multiple times. Some insertions and removals will fail due to non-empty intersections between the sets of keys used by the benchmarking threads. This is meant to mimic real workload and to stress both the hash table and the RCU mechanism in use as much as possible.

The data items referenced by the hash table are small buffers filled with a bitmap pattern. After each insertion, retrieval and removal of a data item, the pattern is verified and overwritten multiple times. Pattern verification checks that readers only "meet" other readers and that writers have exclusive access once they have waited for at least one grace period.

All benchmarking threads interleave insertions and deletions (read-write operations), with retrievals (read-only operations). The workload was run with different ratios between read-only and read-write operations, ranging from 1:1 (one retrieval after each insertion or deletion) to 2047:1.

While the total number of all hash table operations performed by each benchmarking thread is constant across all readers-to-writers (R:W) ratios, this does not guarantee that all operations are equally demanding — read-only operations naturally take less time on average than insertions or deletions. Comparing the performance of one RCU algorithm under various R:W ratios therefore does not make much sense. The goal was to compare different algorithms under the same R:W ratios.

Our benchmarking code provides a simple abstraction over the three RCU implementations (AP-RCU, QRCU and DRCU) and the five hash table implementations, so that the benchmark workload can be executed in both PaR-friendly and TaR-friendly scenarios. The benchmark measures the duration of the multithreaded workload in UTS timer ticks (each corresponding to 10 milliseconds). The number of benchmarking threads was twice the number of available hardware threads. The benchmarking threads ran in the *System Duty Cycle* (SDC) scheduling class, so that involuntary context switches based on time quanta could take place, similar to a real workload.

VI. AP-RCU Benchmark Results

All benchmarking kernels were built using the standard OS/Net compiler suite (Sun Studio 12 from September 2009). All performance measurements presented in this paper ran on a "non-debug" kernel. Five machines were used for benchmarking — four x86-64 *Intel "Nehalem"* machines (one 8-CPU, two 16-CPU and one 24-CPU) with "TurboBoost" capability and one SPARCv9 *Sun UltraSPARC T1 "Niagara"* system with 24 processor threads.

We ran each benchmark on an otherwise idle machine; 17 times in case of the PaR-friendly scenario, and 7 times in the TaR-friendly case. Since most of the results do not fit a normal distribution, we report median values with 95% confidence intervals calculated using order statistics.

Benchmarks results are listed in Table I, where "DRCU" and "QRCU" denote the respective TaR-class algorithms, while "AP-RCU**s**" denotes the blocking API and "AP-RCU**c**" denotes the asynchronous callback API of AP-RCU.

A. PaR-friendly Scenario

The arrangement with a global AP-RCU instance or a single QRCU/DRCU instance per hash table was expected to be disadvantageous for the TaR-class algorithms and the results confirm that. Due to massive contention on the single QRCU/DRCU instance, DRCU is unusable under low R:W ratios on all platforms. This is caused by the DRCU grace period detection mechanism, which needs to lock and unlock the write-side of its internal R/W lock, interrupting all readers.

QRCU performed reasonably well, except for extremely low R:W ratios where write-side contention starts to dominate.

Despite the fact that RCU is designed for high R:W ratios, on the x86-64 platform the performance of the AP-RCUc variant was consistently superior to other mechanisms, even at extremely low R:W ratios, such as 1:1. The AP-RCUs variant appears unsuitable for such low ratios, but its performance is improving with R:W ratios increasing.

8 Hardware Threads, x86-64[3] (Column A). AP-RCU algorithm performs well even under low R:W ratios when its asynchronous API is used. The synchronous API has been outperformed by QRCU at the extreme 1:1 ratio, but performed better than QRCU otherwise.

16 Hardware Threads, x86-64[4] (Column B). The results are similar to those obtained from the machine with 8 hardware threads. Larger differences between some of the results suggest that AP-RCU scales slightly better than the other algorithms.

[3] Intel Core i7 (Q820 at 1.73 GHz) machine, 8 GB of RAM.

[4] Dual Intel Xeon (E5540 at 2.53 GHz) machine, 48 GB of RAM.

	PaR-friendly Scenario (single instance of QRCU/DRCU)						**TaR**-friendly Scenario (per-bucket QRCU/DRCU)					
R:W Type	**(A)** 8-CPU x86-64		**(B)** 16-CPU x86-64		**(C)** 24-CPU SPARCv9		**(D)** 8-CPU x86-64		**(E)** 16-CPU x86-64		**(F)** 24-CPU x86-64	
1:1 DRCU	Too Slow		Too Slow		Too Slow		**3196** ± 12	**100%**	3330 ± 27	105%	2967 ± 39	105%
1:1 QRCU	12319 ± 16	357%	25012 ± 54	771%	*119543* ± 223	*560%*	3668 ± 8	115%	3712 ± 9	117%	3243 ± 4	115%
1:1 AP-RCUs	*14187* ± 26	*411%*	*27567* ± 86	*849%*	43826 ± 102	205%	*13772* ± 72	*431%*	*21795* ± 180	*686%*	*27040* ± 97	*852%*
1:1 AP-RCUc	**3451** ± 13	**100%**	**3246** ± 21	**100%**	**21365** ± 84	**100%**	3316 ± 63	104%	**3175** ± 32	**100%**	**2830** ± 51	**100%**
7:1 DRCU	Too Slow		Too Slow		Too Slow		**2893** ± 2	**100%**	3240 ± 6	119%	2872 ± 7	117%
7:1 QRCU	*8300* ± 26	*273%*	*13466* ± 17	*502%*	*82236* ± 275	*422%*	3510 ± 16	121%	3811 ± 11	140%	3381 ± 7	138%
7:1 AP-RCUs	6037 ± 30	199%	9743 ± 48	363%	21477 ± 105	110%	*5883* ± 27	*203%*	*7648* ± 114	*281%*	*9450* ± 67	*384%*
7:1 AP-RCUc	**3037** ± 7	**100%**	**2682** ± 8	**100%**	**19504** ± 31	**100%**	**2895** ± 19	**100%**	**2724** ± 19	**100%**	**2458** ± 44	**100%**
31:1 DRCU	*24232* ± 533	*844%*	*50537* ± 327	*2016%*	*131062* ± 2208	*853%*	2753 ± 3	101%	3134 ± 8	123%	2775 ± 3	118%
31:1 QRCU	6682 ± 7	233%	11534 ± 22	460%	95583 ± 595	622%	3306 ± 17	121%	3687 ± 11	144%	3324 ± 24	141%
31:1 AP-RCUs	4018 ± 18	140%	5410 ± 27	216%	**15368** ± 49	**100%**	*3879* ± 69	*142%*	*4085* ± 84	*160%*	*5063* ± 107	*215%*
31:1 AP-RCUc	**2872** ± 10	**100%**	**2507** ± 5	**100%**	18947 ± 20	123%	**2724** ± 18	**100%**	**2555** ± 24	**100%**	**2353** ± 19	**100%**
127:1 DRCU	*11400* ± 307	*405%*	*26457* ± 219	*1081%*	62164 ± 257	449%	**2673** ± 12	**100%**	3146 ± 15	126%	2795 ± 5	123%
127:1 QRCU	6318 ± 6	225%	10884 ± 19	445%	*102330* ± 933	*740%*	3094 ± 13	116%	*3558* ± 20	*142%*	*3201* ± 54	*140%*
127:1 AP-RCUs	3320 ± 32	118%	3225 ± 93	132%	**13837** ± 90	**100%**	*3210* ± 104	*121%*	3159 ± 224	126%	*3355* ± 340	*147%*
127:1 AP-RCUc	**2813** ± 9	**100%**	**2447** ± 15	**100%**	18637 ± 20	135%	**2663** ± 15	**100%**	**2501** ± 21	**100%**	**2279** ± 27	**100%**
511:1 DRCU	*7742* ± 32	*286%*	*16124* ± 119	*692%*	42226 ± 159	232%	**2555** ± 19	**100%**	3120 ± 18	130%	2800 ± 16	130%
511:1 QRCU	6303 ± 17	233%	10985 ± 40	472%	*106186* ± 121	*740%*	*2888* ± 28	*113%*	*3343* ± 40	*139%*	*3019* ± 39	*140%*
511:1 AP-RCUs	2785 ± 35	103%	2824 ± 31	121%	**13063** ± 87	**100%**	*2710* ± 170	*106%*	2800 ± 105	117%	2733 ± 89	127%
511:1 AP-RCUc	**2708** ± 9	**100%**	**2329** ± 11	**100%**	17964 ± 27	135%	**2549** ± 32	**100%**	**2403** ± 37	**100%**	**2156** ± 38	**100%**
2047:1 DRCU							**2238** ± 24	**101%**	2831 ± 27	140%	2579 ± 30	142%
2047:1 QRCU	Not						*2490* ± 21	*113%*	*2966* ± 16	*146%*	*2661* ± 35	*147%*
2047:1 AP-RCUs	Tested						2568 ± 19	116%	2443 ± 31	120%	2169 ± 38	119%
2047:1 AP-RCUc							**2212** ± 21	**100%**	**2029** ± 21	**100%**	**1816** ± 35	**100%**

TABLE I
AP-RCU BENCHMARK RESULTS (**bold** = BEST VALUE, *italics* = WORST VALUE)

24 Hardware Threads, SPARCv9[5] (Column C). On the SPARCv9 architecture, the QRCU algorithm seems to have a very high read-side overhead. Surprisingly, the higher the R:W ratio, the worse the performance of QRCU. This might be caused by the emulation of atomic increment and decrement instructions with the CaS instruction on SPARC.

AP-RCU seems to scale well on 24 hardware threads. In this special case, it outperforms QRCU even at the extreme 1:1 ratio. The AP-RCUs variant surprisingly managed to outperform the AP-RCUc variant for higher R:W ratios. This phenomenon is currently being investigated.

B. TaR-friendly Scenario

The arrangement with one QRCU/DRCU instance per hash table bucket leads to virtually zero contention on the QRCU/DRCU instances, resulting in major speedup for TaR-class algorithms (compared to the PaR-friendly scenario). Due to the negligible contention, DRCU consistently outperforms QRCU, because its read-side critical section only uses a single atomic instruction and a memory barrier, compared to (more expensive) atomic increment and decrement in QRCU. While QRCU generally outperforms other RCU algorithms under high write-side contention, the read-side overhead of QRCU dominates in this scenario.

While both variants of AP-RCU perform well in the PaR-friendly scenario, they cannot benefit from the reduced contention, since PaR-class algorithms only use one global instance. Consistent with the PaR-friendly scenario, the results indicate that the AP-RCUs variant is unsuitable for very low R:W ratios. On the other hand, the AP-RCUc variant exhibits competitive performance even with extremely low R:W ratios. In general, the performance of both AP-RCUs and AP-RCUc improves as the number of processors and R:W ratio increase, in some cases significantly outperforming the TaR-class algorithms even in the TaR-friendly scenario.

Hash table implementations slightly differ from the one used in the TaR-friendly scenario and results will differ even for AP-RCUs and AP-RCUc, although both scenarios run the same workload.

8 and 16 Hardware Threads, x86-64 (Columns D & E). Similarly to the PaR-friendly scenario, AP-RCU scaled slightly better under high R:W ratios.

24 Hardware Threads, x86-64 (Column F). In comparison to column E, the performance benefit of AP-RCU over QRCU and DRCU tends to be worse or better under low or high R:W ratios, respectively, which can be expected.

The AP-RCUc variant outperforms QRCU and DRCU on NUMA machines with first-touch policy (Columns E and F) and performs roughly equally well on UMA (Column D), despite the fact that only one global instance of AP-RCU can exist, whereas independent QRCU and DRCU instances are created for each hash table bucket.

VII. FUTURE AND RELATED WORK

To further test our claim than AP-RCU is non-intrusive, we are currently implementing our algorithm in HelenOS [24], a microkernel-based operating system. Our initial observations are promising. Although the architecture of HelenOS is unrelated to UTS, no modifications to the scheduler or clock handler were needed for the implementation of AP-RCU.

Mechanisms similar to RCU date back to late 1980s. Research carried out by Paul McKenney and members of the Linux kernel community (Rusty Russell, Dipankar Sarma) provides a foundation for most existing applications of RCU. Various RCU algorithms and related data structures are described in [6] and further advances in implementations and applications are outlined in a series of Linux Weekly News (LWN) articles [10], [12], [13], [14], [15], [17], [18].

[5]Sun UltraSPARC T1 (at 1.0 GHz) machine, 4 GB of RAM.

Algorithms based on RCU are also used in experimental kernels, such as K42 [2]. Alternative RCU algorithms have been developed and tested by other kernel hackers [6, p. 123, 326]. A detailed discussion of memory ordering issues related to RCU on various hardware architectures can be found in McKenney's Linux Journal articles [8].

Several major Linux kernel subsystems were modified to take advantage of RCU during the 2.5 kernel development, avoiding mutual exclusion and improving scalability [5], [7]. Key design patterns for migrating algorithms based on mutual exclusion to RCU are described in [6, p. 137–178].

The current Linux kernel contains multiple RCU implementations. By far the most commonly used RCU implementation strongly depends on scheduler and clock tick handler instrumentation. Actions related to RCU are taken from within these contexts when appropriate. Conditions related to RCU are checked on each context switch and on each timer interrupt. The algorithm strongly depends on Linux kernel internals and cannot be easily ported to other kernels. All the alternative RCU implementations in the Linux kernel have historically enjoyed only limited and special-purpose uses.

The research on improving scalability when accessing shared data has many branches, and RCU is but one of them. The Scalable Synchronization Research Group (Sun Labs at Oracle) has been experimenting with hardware transactional memory (HTM), evaluating optimistic transactional algorithms accessing a hash table and a red-black tree concurrently [20]. The upcoming Intel CPU microarchitecture (Haswell) will also support HTM instructions [21]. Another branch of research focuses on lock-free data structures [1] and related non-blocking reclamation mechanisms (hazard pointers [19], Pass the Buck [4], or lock-free reference counting [25]).

While RCU and the above mentioned techniques are but two sides of one coin and there is no globally optimal scheme [3], we have focused on RCU, because it is the approach taken to improve scalability in Linux, a widely-used operating system. Experimental evaluations of the Linux RCU have clearly shown its benefits, and we believe that RCU would also prove beneficial in other operating systems. Unlike HTM, RCU can be implemented on virtually all contemporary hardware architectures, including commodity hardware.

VIII. Conclusion

With the growing degree of parallelism in hardware, the benefits of RCU and similar techniques are becoming increasingly important. To simplify adoption of RCU, we have designed a novel non-intrusive variant of PaR-class RCU algorithm (AP-RCU) that does not require modifications to the scheduler or clock interrupt handler.

AP-RCU depends only on basic kernel-level concepts, such as atomic operations, memory barriers, mutexes, threads and CPU affinity API. As most modern operating system kernels (both monolithic kernels and microkernels) readily provide these features, we consider AP-RCU non-intrusive.

In contrast to TaR-class algorithms, AP-RCU can guarantee both readers and writers to never block, which makes it suitable for kernel use — especially in contexts where blocking is not allowed.

As a proof-of-concept, we have implemented AP-RCU within the Solaris kernel (UTS) and tested it on both supported hardware architectures. Even though the implementation was done with focus on clarity of code over throughput, we have shown that it exhibits performance comparable to the best non-RCU solutions in RCU-unfriendly situations, and is superior to non-RCU solutions in reader-heavy scenarios, for which RCU was designed in the first place.

The implementation of AP-RCU for UTS kernel, along with the implementation of QRCU and a non-blocking hash-table [6], can be obtained from [23].

Acknowledgements. The work was partially supported by Charles University institutional funding SVV-2012-265312 and Charles University research funds PRVOUK.

References

[1] Fraser K.: *Practical lock-freedom*, Ph.D. Thesis, University of Cambridge, 2004
[2] Gamsa B., Krieger O., Appavoo J., Stumm M.: *Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System*, University of Toronto, IBM Watson Research Center, 1997
[3] Hart T., McKenney P., Brown A.: *Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation*, IPDPS, 2006
[4] Herlihy M., Luchangco V., Moir M.: *The Repeat Offender Problem: A Mechanism for Supporting Dynamic-sized Lock-free Data Structures*, Tech. Report, Sun Microsystems, 2002
[5] McKenney P., Appavoo J., Kleen A., Krieger O., Russell R., Sarma D., Soni M.: *Read-Copy Update*, Ottawa Linux Symposium, http://lse.sourceforge.net/locking/rcu/rclock_OLS.2001.05.01c.sc.pdf, 2001
[6] McKenney P.: *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*, Ph.D. Thesis, Oregon State University, 2004
[7] McKenney P.: *Kernel Korner - Using RCU in the Linux 2.5 Kernel*, Linux Journal, Art. 6993, 2003
[8] McKenney P.: *Memory Ordering in Modern Microprocessors*, Linux Journal, Art. 8211 & Art. 8212, 2005
[9] McDougall R., Mauro J.: *Solaris Internals*, Prentice Hall, Westford, 2006
[10] McKenney P.: *Priority-Boosting RCU Read-Side Critical Sections*, Linux Technology Center, IBM Beaverton, http://www.rdrop.com/users/paulmck/RCU/RCUbooststate.2007.04.16a.pdf, 2007
[11] McKenney P.: *QRCU with Lockless Fastpath*, LWN, Art. 223752, 2007
[12] McKenney P.: *Using Promela and Spin to verify parallel algorithms*, LWN, Art. 243851, 2007
[13] McKenney P.: *The design of preemptible read-copy-update*, LWN, Art. 253651, 2007
[14] McKenney P., Walpole J.: *What is RCU, Fundamentally?*, LWN, Art. 262464, 2007
[15] McKenney P.: *What Is RCU? Part 2: Usage*, LWN, Art. 263130, 2007
[16] McKenney P.: *What Is RCU, Really?*, LWN, http://www.rdrop.com/users/paulmck/RCU/whatisRCU.html, 2007
[17] McKenney P.: *RCU part 3: the RCU API*, LWN, Art. 264090, 2008
[18] McKenney P.: *Hierarchical RCU*, LWN, Art. 305782, 2008
[19] Michael M.: *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*, IEEE Trans. on Parallel and Distributed Systems, Vol. 15, No. 6, 2004
[20] Dice D., Yossi L., Moir M., Nussbaum D.: *Early Experience with a Commercial Hardware Transactional Memory Implementation*, ASPLOS, 2009
[21] *Transactional Synchronization in Haswell*, http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/
[22] Podzimek A.: *Read-Copy-Update for OpenSolaris*, Diploma Thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2010
[23] http://d3s.mff.cuni.cz/software/rcu/rcu.patch
[24] HelenOS microkernel multiserver operating system, http://www.helenos.org/, Charles University in Prague
[25] Valois J.: *Lock-free linked lists using compare-and-swap*, PODC, 1995