Grace Sharing Userspace RCU

Pedro Ramalhete[1] **and Andreia Correia**[2]

1 **Cisco Systems**
`pramalhe@gmail.com`

2 **Concurrency Freaks**
`andreiacraveiroramalhete@gmail.com`

Abstract

RCU is a concurrency construct that has been used for many different purposes, from safe memory reclamation, to relativistic programming. Unfortunately, it requires support in the operating system, which is currently provided only in Linux. An alternative is to use Userspace RCU (URCU) which can be deployed in any operating system. Several algorithms are known for implementing URCU but only a few are implemented using only atomics and the C11/C++ memory model, and are capable of providing wait-free progress for the read-side, and scalability on the update-side by *sharing grace periods*. The grace sharing property is interesting because without it, concurrent calls to `synchronize_rcu()` will be serialized.

We present two novel grace sharing URCU algorithms with wait-free progress for readers and starvation-free progress for updaters. Due to the sharing of the grace period and low synchronization overhead, calls to `synchronize_rcu()` can be up to 25 times faster than state of the art URCU implementations in microbenchmarks, while maintaining high scalability for readers calling `rcu_read_lock()` and `rcu_read_unlock()`.

1998 ACM Subject Classification D.4.1 - Process Management - Synchronization

Keywords and phrases RCU, atomics, memory model

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

When implementing data structures in languages without an automatic Garbage Collector, or just to manage object lifetime, some kind of memory reclamation technique is required. Several such techniques have been discovered [6, 13, 2, 1, 11, 5] with RCU [11, 5] one of the most well known. RCU has two particularly interesting properties for its non-reclaiming methods, i.e. `rcu_read_lock()` and `rcu_read_unlock()`, one being its wait-free progress, and the other its low latency with high scalability. RCU is unfortunately not a generic technique because it requires operating system support.

There are several Userspace RCU (URCU) implementations [5, 17, 9]. The **userspace-rcu** [4] library provides two variants that can be used regardless of operating system support, named *Memory Barrier URCU* and *Bullet Proof URCU*. These two variants of URCU are currently implemented in C99 with CPU specific memory barriers, and although many architectures are supported, the support is not universal, and other languages may have difficulty linking with the library. The **userspace-rcu** library provides two other URCU variants which we will not consider here due to their lack of generality, namely, *QSBR URCU* and *Signal-based URCU*, requiring a periodic call to `rcu_quiescent_state()` or POSIX style signals support by the operating system, respectively.

We present two new URCU algorithms which we named *GraceVersion URCU* and *Two-Phase URCU*. They can be implemented with a sequentially consistent memory model, like the C11/C++1x, and unlike other URCU algorithms [5, 17, 9], they allow for sharing of the grace period in `synchronize_rcu()`.

Until now, *Memory Barrier URCU* was the only generic URCU algorithm which allowed multiple threads calling `synchronize_rcu()` to make progress with a single version increment, called *grace sharing*. *Memory Barrier URCU* achieves this at the expense of acquiring a mutual exclusion lock, where the thread which acquired the lock is responsible for advancing the grace period, thus causing the other threads calling `synchronize_rcu()` to block. Our algorithms are the only ones capable of sharing a grace period without threads calling `synchronize_rcu()` blocking each other.

2 Grace Version Algorithm

In this technique, shown in Algorithm 1, there is a global version, named `updaterVersion`, and an array where each *reader* thread — a thread calling `rcu_read_lock()` and `rcu_read_unlock()` — has a unique entry to publish its state, named `readersVersion`. This state can either be `NOT_READING` or the latest read value of `updaterVersion`. Each *updater* — a thread calling `synchronize_rcu()` — does a CAS to increment the atomic variable `updaterVersion` (line 21) and then spins while waiting for all the ongoing readers to complete by scanning the array of their versions, named `readersVersion`. Upon calling `rcu_read_lock()`, a reader will read the current value in `updaterVersion` and set its uniquely assigned entry in the `readersVersion` array to the same value (line 9), and then re-check that the version has changed and if so, update the `readersVersion` with the new entry. When the read operation is done, it will call `rcu_read_unlock()` which will set its entry in `readersVersion` back to `NOT_READING` (line 15), a constant with a special value that means the reader is not currently active.

A non-trivial detail in `rcu_read_lock()` is the purpose of the load and store in lines 10 and 11, which is related to instruction re-ordering. According to the C++ Memory Model, the sequentially consistent (seq-cst) store in line 9 could be re-ordered with regular code inside the read-side critical section, therefore, to prevent this incorrect re-ordering from occurring, we do the seq-cst load on line 10. The relaxed store in line 11 is not necessary for the algorithm's correctness but can yield better performance for *updaters* with very little cost to readers.

The GraceVersion algorithm allows multiple threads calling `synchronize_rcu()` to share a grace period [10]. In line 21 of `synchronize_rcu()`, if the CAS incrementing the `updaterVersion` fails, then some other thread calling `synchronize_rcu()` has succeeded. This can occur simultaneously for multiple threads, thus allowing them all to share the same increment of `updaterVersion`, therefore sharing a single grace period.

One disadvantage of this algorithm is its need of prior registration for the readers, such that each reader thread requires a unique entry in the `readersVersion` array. Depending on the application, assigning a specific thread id to each thread may be easy, or it may be extremely difficult. The Two-Phase algorithm on the next section simplifies this task by allowing the usage of counters.

3 Two-Phase Algorithm

This algorithm uses a two-phase approach and requires two *ReadIndicator* (RI) instances [3], with the simplest possible ReadIndicator being an atomic seq-cst counter. The idea consists of having two ReadIndicators, that the readers increment/decrement as they enter/leave the

Algorithm 1 GraceVersion URCU in C++14

```
class URCUGraceVersion {
  const uint64_t NOT_READING = 0xFFFFFFFFFFFFFFFE;
  std::atomic<uint64_t> updaterVersion { 0 };
  std::atomic<uint64_t> readersVersion[MAX_THREADS];

public:
  int rcu_read_lock(const int tid) noexcept {
    const uint64_t rv = updaterVersion.load();
    readersVersion[tid].store(rv);
    const uint64_t nrv = updaterVersion.load();
    if (rv != nrv) readersVersion[tid].store(nrv, memory_order_relaxed);
  }

  void rcu_read_unlock(const int tid) noexcept {
    readersVersion[tid].store(NOT_READING, memory_order_release);
  }

  void synchronize_rcu() noexcept {
    const uint64_t waitForVersion = updaterVersion.load();
    uint64_t tmp = waitForVersion;
    updaterVersion.compare_exchange_strong(tmp, waitForVersion+1);
    for (int i=0; i < MAX_THREADS; i++) {
      while (readersVersion[i].load() <= waitForVersion) { // spin }
    }
  }

};
```

read-side critical sections (lines 8 and 13 of Algorithm 2). The RI instance to increment is chosen using the first bit of `updaterVersion` (line 7), which is atomically incremented by an updater using a Compare-And-Swap (CAS).

An updater will only advance the `updaterVersion` (line 25) after checking that older readers moved to the RI that is now active (lines 19, 20, 21). It will then advance `updaterVersion` with a single CAS. In the event that the CAS does not succeed, it is still guaranteed that another thread has succeeded and `updaterVersion` has been incremented. This mechanism allows for multiple updaters to share the same grace period, as on the GraceVersion algorithm. This approach improves the scalability of `synchronize_rcu()`, which reduces contention on the `updaterVersion` and reduces cache misses for readers. Finally, the updater will make sure all readers moved to the new RI, by checking the opposite RI (line 27), or if the `updaterVersion` has advanced (line 28) which means that some other updater has seen that RI as empty.

The Two-Phase algorithm has a few non-obvious details: In line 21, it is safe to break out of the loop if the current `updaterVersion` has been incremented. It means another thread has seen the RI as empty and incremented the `updaterVersion` (line 25); In line 23, there is no need to increment the `updaterVersion` if some other thread has done it already. It also means another thread has seen the first RI as empty; In lines 28 and 20, it is safe to return if the current `updaterVersion` is two or more values (phases) above the original `updaterVersion` value. If the `updaterVersion` has advanced at least twice, it guarantees that another updater that shared this same grace period has already validated

Algorithm 2 Two-Phase URCU in C++14

```
template<typename RI> class URCUTwoPhase {
  std::atomic<int64_t> updaterVersion { 0 };
  RI readIndicator[2];

public:
  int rcu_read_lock() noexcept {
      const int index = (int)(updaterVersion.load() & 1);
      readIndicator[index].arrive();
      return index;
  }

  void rcu_read_unlock(const int index) noexcept {
      readIndicator[index].depart();
  }

  void synchronize_rcu() noexcept {
      const int64_t currUV = updaterVersion.load();
      const int64_t nextUV = (currUV+1);
      while (!readIndicator[(int)(nextUV&1)].isEmpty()) { // spin
          if (updaterVersion.load() > nextUV) return;
          if (updaterVersion.load() == nextUV) break;
      }
      if (updaterVersion.load() == currUV) {
          auto tmp = currUV;
          updaterVersion.compare_exchange_strong(tmp, nextUV);
      }
      while (!readIndicator[(int)(currUV&1)].isEmpty()) { // spin
          if (updaterVersion.load() > nextUV) return;
      }
  }
};
```

that all readers from that grace period have completed, and that an updater is checking for the next grace period. This implies the impossibility of having readers in-flight on this grace period.

In the code shown in Algorithm 2, we allow for any ReadIndicator [3, 7, 16], with two different ReadIndicators being shown in Figure 1: PerThread where each thread has a unique entry in a cache line padded array; CountersArray where the thread's id is hashed to obtain an index in an array of counters, with size 2x the number of cores, that is incremented/-decremented using Fetch-And-Add.

3.1 ReadIndicators

Many different ReadIndicator implementations exist [3, 7, 16]. One of the simplest implementations is the usage of a single atomic variable which is incremented upon calling `arrive()` and decremented when calling `depart()`. This implementation has low memory footprint and great flexibility, requiring no prior thread registration, however, it suffers from contention when the read-side critical sections are short in time. Example code is shown in Algorithm 3.

Algorithm 3 ReadIndicator: Single atomic counter

```
class RIAtomicCounter
    std::atomic<uint64_t> counter { 0 }; {

public:
    void arrive(void) noexcept {
        counter.fetch_add(1);
    }

    void depart(void) noexcept {
        counter.fetch_add(−1);
    }

    bool isEmpty(void) noexcept {
        return counter.load() == 0;
    }
};
```

To address the issue of contention, an array of counters can be used, with one counter per cache line so as to avoid false-sharing effects. In some applications, doing prior thread registration can be cumbersome or nearly impossible due to having an unknown number of threads that may be created or destroyed at runtime, therefore, instead of assigning an entry in the array of counters to a unique thread, we can make a hash of the thread's id to statistically reduce contention. This implementation is shown in Algorithm 4.

The number of entries in the array can be tweaked according to the application's requirements. An increase in the size of the array will typically improve read-side throughput for the calls to `arrive()` and `depart()` but increase memory usage and decrease throughput for `isEmpty()`.

When it comes to the lowest contention possible for the readers, the best is to use an array where each entry is in its own cache line and is statically assigned to a thread. This may be an unwanted solution if the number of threads is high, or when memory usage is

Algorithm 4 ReadIndicator: Array of Atomic Counters

```
class RIAtomicCounterArray {

private:
    static const int MAX_THREADS = 64;
    static const int CLPAD = (128/sizeof(std::atomic<uint64_t>));
    static const int COUNTER_SIZE = 2*MAX_THREADS;
    static std::hash<std::thread::id> hashFunc;
    alignas(128) std::atomic<uint64_t> counters[COUNTER_SIZE*CLPAD] ;

public:
    RIAtomicCounterArray() {
        for (int i=0; i < COUNTER_SIZE; i++) {
            counters[i*CLPAD].store(0, std::memory_order_relaxed);
        }
    }

    void arrive(void) noexcept {
        const uint64_t tid = hashFunc(std::this_thread::get_id());
        const int icounter = (int)(tid % COUNTER_SIZE);
        counters[icounter*CLPAD].fetch_add(1);
    }

    void depart(void) noexcept {
        const uint64_t tid = hashFunc(std::this_thread::get_id());
        const int icounter = (int)(tid % COUNTER_SIZE);
        counters[icounter*CLPAD].fetch_add(−1);
    }

    bool isEmpty(void) noexcept {
        for (int i = 0; i < COUNTER_SIZE; i++) {
            if (counters[i*CLPAD].load() > 0) return false;
        }
        return true;
    }
};
```

bounded, but otherwise, it should provide the highest throughput possible for readers, as can be seen on the next section. An implementation of this ReadIndicator is shown in Algorithm 5.

Algorithm 5 ReadIndicator: One entry per Thread

```
class RIEntryPerThread {

private:
    enum State { NOT_READING=0, READING=1 };
    static const int MAX_THREADS = 64;
    static const int CLPAD = (128/sizeof(std::atomic<uint64_t>));
    alignas(128) std::atomic<long> states[MAX_THREADS*CLPAD];

public:
    RIEntryPerThread() {
        for (int i = 0; i < MAX_THREADS; i++) {
            states[i*CLPAD].store(NOT_READING, std::memory_order_relaxed);
        }
    }

    void arrive(int tid) {
        states[tid*CLPAD].store(READING);
        // Do seq-cst load to prevent re-ordering of code below arrive()
        states[tid*CLPAD].load();
    }

    void depart(int tid) {
        states[tid*CLPAD].store(NOT_READING, std::memory_order_release);
    }

    bool isEmpty(void) {
        for (int tid = 0; tid < MAX_THREADS; tid ++) {
            if (states[tid*CLPAD].load() == READING) {
                return false;
            }
        }
        return true;
    }
};
```

4 Correctness

The correctness of a Userspace RCU algorithm [12] is based on the premise that a call to `synchronize_rcu()`, executed by an updater thread u, is allowed to return to the callee only *after* all reader threads that called `rcu_read_lock()` *before* the updater started the call to `synchronize_rcu()`, have completed their read-side critical section by calling `rcu_read_unlock()`, where *after* and *before* relate to ordering in a linearizable global history [8].

Let's consider that all methods should appear to *take effect* instantaneously at some moment between its invocation and response [8]. We define $rlock_t^{(i)}$ as the execution of the i-th call to `rcu_read_lock()` performed by thread t and $runlock_t^{(i)}$ the execution of the corresponding i-th call to `rcu_read_unlock()` performed by thread t. The execution of `synchronize_rcu()` by updater thread u is represented as $sync_u$.

The following condition must hold: *For every read-side critical section i of thread t, if the i-th execution of `rcu_read_lock()` by reader thread t takes effect before the execution of `synchronize_rcu()` by updater thread u, then the execution of `synchronize_rcu()` will have to take effect after the corresponding i-th execution of `rcu_read_unlock()`.*

$$\forall i \in \mathbb{N}, t \neq u \colon rlock_t^{(i)} \prec sync_u \implies runlock_t^{(i)} \prec sync_u \tag{1}$$

Methods in our execution history are ordered by the happens-before relation, denoted $\prec$. Based on this condition, we will now describe the correctness proofs for our two URCU algorithms.

4.1 GraceVersion Correctness

Assuming that thread t satisfies the condition $rlock_t^{(i)} \prec sync_u$ then, the `updaterVersion` published by thread t, in line 9 of Algorithm 1, will be equal or lower than the `updaterVersion` read by an updater thread u to the local variable `waitForVersion`, in line 19. Based on the value read in line 19, thread u will be blocked in a loop, at line 23, until thread t executes `rcu_read_unlock()`. Subsequently, the call to `synchronize_rcu()` will have to take effect after $runlock_t^{(i)}$.

The proof of condition 1 can also be done by contrapositive, if $sync_u \prec runlock_t^{(i)}$ then $sync_u \prec rlock_t^{(i)}$. Considering that $sync_u \prec runlock_t^{(i)}$, `synchronize_rcu()` happened before `rcu_read_unlock()` was executed. This means that thread u was able to complete before thread t executed `rcu_read_unlock()`, this event can only be possible if thread t has published an `updaterVersion` higher than `waitForVersion`, value read by thread u at line 19, or t published `NOT_READING`, in either case `synchronize_rcu()` will have to take effect before $rlock_t^{(i)}$.

To provide a linearizable history, the load and store on lines 8 and 9 must not be re-ordered, hence the implicit usage of `std::memory_order_seq_cst` for both operations, nor can the loads in lines 19 and 23 be re-ordered, which use the same `std::memory_order_seq_cst` ordering.

To guarantee starvation-free progress, the updater has to insure `updaterVersion` is incremented, by itself or another updater, line 21.

4.2 TwoPhase Correctness

Assuming that thread t satisfies the condition $rlock_t^{(i)} \prec sync_u$ then, thread t has arrived before `synchronize_rcu()` was invoked, line 8 of Algorithm 2. Because thread t may have arrived on either of the ReadIndicators, the updater thread u will have to verify that both are empty, lines 19 and 27, or another updater thread has done the validation, lines 20, 21 and 28. If at any point in time the `updaterVersion` has advanced to a value higher than `nextUV`, value read by thread u at line 18, it means that another updater thread has validated that both ReadIndicators were empty for the grace period `currUV`, line 17. This guarantees that the call to `synchronize_rcu()` by updater thread u will have to take effect after $runlock_t^{(i)}$.

The proof of condition 1 can also be done by contrapositive, if $sync_u \prec runlock_t^{(i)}$ then $sync_u \prec rlock_t^{(i)}$. Considering that $sync_u \prec runlock_t^{(i)}$, the response to the call to `synchronize_rcu()` happened before `rcu_read_unlock()` was executed. This means that thread u was able to complete before thread t executed `rcu_read_unlock()`. This is possible only if thread t has not arrived, or has arrived to the ReadIndicator after updater thread u has completed the validation of that ReadIndicator, in either case $sync_u \prec rlock_t^{(i)}$.

To guarantee starvation-free progress, the updater starts by waiting for the opposite ReadIndicator to be empty, line 19. By contradiction, if the validation were to be done first at the current ReadIndicator, it could happen that new reader threads would continuously arrive and depart without the current ReadIndicator ever being empty, thus starving the updater thread. Also, as soon as `updaterVersion` is incremented by one (line 21), or more (lines 20 or 28), it will terminate the validation on one or both of the ReadIndicators, respectively.

This mechanism with two phases and two ReadIndicators, where each ReadIndicator is drained successively, has been used before on other works [15, 14].

5 Performance and Discussion

Figure 1 shows the median for 5 runs of the number of operations per second, on a microbenchmark that iterates for 20 seconds on each run, per number of threads. The benchmark was compiled with GCC 5.2.0 and ran on an AMD Opteron 6272 server with a total of 32 cores, running Ubuntu 14.04. A thread can do a read operation (call to `rcu_read_lock()`/`rcu_read_unlock()`) or an update operation (call to `synchronize_rcu()`). For the top plot of Figure 1, a read operation consists of a call to `rcu_read_lock()` followed by a seq-cst load on a global variable and then a call to `rcu_read_unlock()`. For the bottom rightmost plot, a read operation consists of a call to `rcu_read_lock()` followed by the sum of an array of 100000 integers, and a call to `rcu_read_unlock()`, a read operation with a *long* time duration. This is an ideal scenario to show the effects of grace sharing on each URCU.

Figure 1 shows a comparison of our GraceVersion and Two-Phase algorithms in C++ versus the BulletProof and default implementations of MemoryBarrier in the URCU libray [4] version 0.9.0, shown as `URCU-Lib BulletProof` and `URCU-Lib MemoryBarrier`. As seen from Figure 1, our GraceVersion algorithm surpasses the default and BulletProof URCU-Lib implementations for all tested scenarios for both short and long reads, while the Two-Phase algorithm surpasses for scenarios where reads are short or non existent (bottom leftmost plot), and matches the default URCU-Lib for long read operations (bottom rightmost plot). Unlike the Two-Phase algorithm, GraceVersion requires a static assignment of each thread to an entry of an array, making its usage inflexible.

Our algorithms are better than the BulletProof URCU in all tested scenarios, sometimes going up to a 1000x improvement as shown on the bottom rightmost plot. They are always as good as, or better than, the default URCU (MemoryBarrier), up to 25x as shown on the bottom leftmost plot of Figure 1.

Both our algorithms allow for multiple URCU instances to co-exist. For example, when applied as a memory reclamation technique to a data structure, one URCU instance can be created per data structure instance, such that each updater has to wait only for the readers acting on that particular data structure, as opposed to having to wait for all the threads currently accessing any of the data structure instances.

The adaptability of the TwoPhase algorithm to any kind of ReadIndicator means that it can be deployed in different scenarios by choosing a different ReadIndicator, like for example NUMA-aware [3]. The choice of ReadIndicator can depend on the desired characteristics, if low memory usage is desired, then a single atomic counter can be used as an RI.

Although blocking for updaters, like all RCU are, in our algorithms an updater is

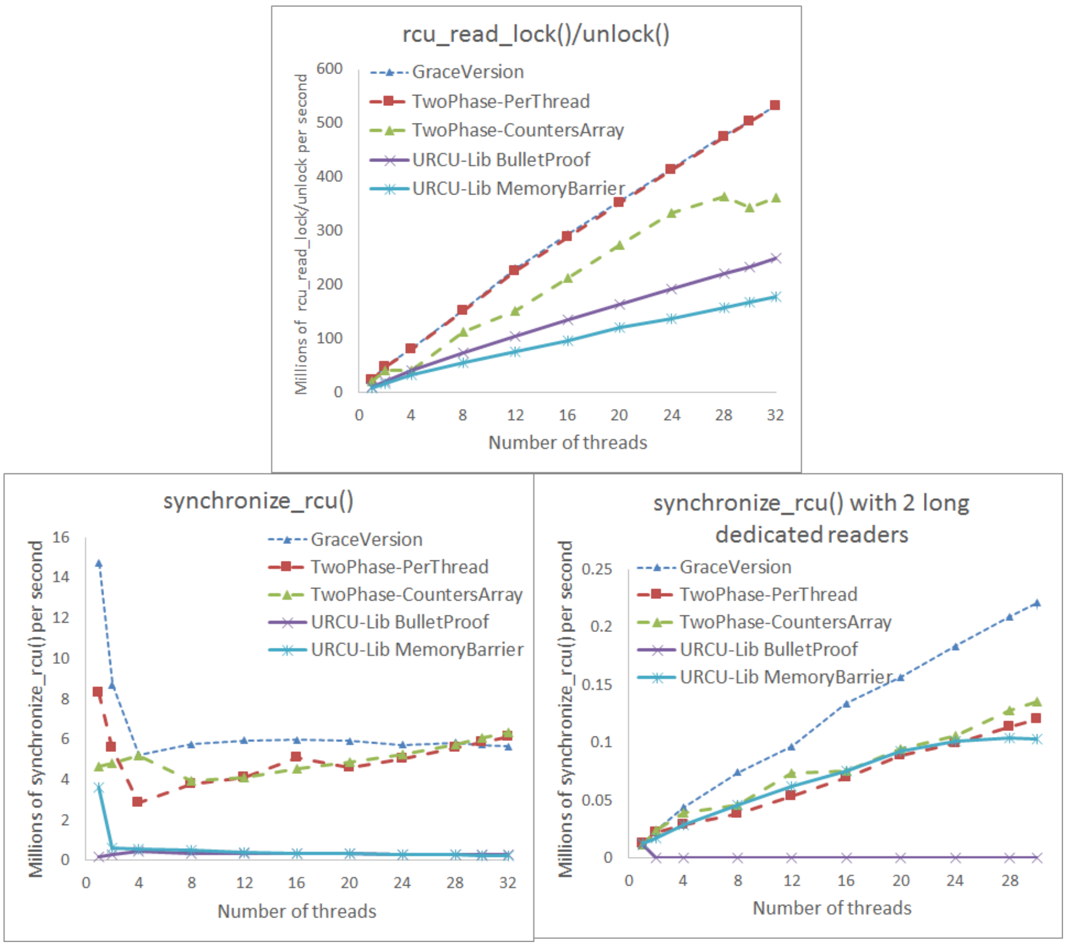

Figure 1 Comparison of four different URCU implementations: our GraceVersion, our TwoPhase using an entry per thread or arrays of counters as ReadIndicators, the Userspace-RCU library's Bullet Proof, and Memory Barrier (default) implementation. The top plot shows read-only operations. The bottom leftmost plot show measurements for update-only operations. The bottom rightmost plot shows the number of update operations when two other threads are continuously executing *long* read-only operations. Higher is better.

starvation-free and is not blocked by other updaters, and may share a grace period, with improved scalability when compared with state of the art URCU implementations.

References

1 Alistarh, D., Eugster, P., Herlihy, M., Matveev, A., and Shavit, N. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 25.

2 Braginsky, A., Kogan, A., and Petrank, E. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures* (2013), ACM, pp. 33–42.

3 Calciu, I., Dice, D., Lev, Y., Luchangco, V., Marathe, V. J., and Shavit, N. NUMA-aware reader-writer locks. *PPoPP 2013* (2013).

4 Desnoyers, M., and McKenney, P. E. Userspace rcu. `http://liburcu.org/`, 2015.

5 Desnoyers, M., McKenney, P. E., Stern, A. S., Dagenais, M. R., and Walpole, J. User-level implementations of read-copy update. *Parallel and Distributed Systems, IEEE Transactions on 23*, 2 (2012), 375–382.

6 Detlefs, D. L., Martin, P. A., Moir, M., and Steele Jr, G. L. Lock-free reference counting. *Distributed Computing 15*, 4 (2002), 255–271.

7 Ellen, F., Lev, Y., Luchangco, V., and Moir, M. Snzi: Scalable nonzero indicators. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (2007), ACM, pp. 13–22.

8 Herlihy, M., and Wing, J. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (1990), 463–492.

9 McKenney, P. E. Is parallel programming hard, and, if so, what can you do about it? *Linux Technology Center, IBM Beaverton* (2011).

10 McKenney, P. E. What happens when 4096 cores all do synchronize_rcu_expedited()? https://www.youtube.com/watch?v=1nfpjHTWaUc, 2016.

11 McKenney, P. E., Appavoo, J., Kleen, A., Krieger, O., Russell, R., Sarma, D., and Soni, M. Read-copy update. In *AUUG Conference Proceedings* (2001), AUUG, Inc., p. 175.

12 McKenney, P. E., and Walpole, J. Rcu semantics: A first attempt.

13 Michael, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on 15*, 6 (2004), 491–504.

14 Ramalhete, P., and Correia, A. Left-Right- A Concurrency Control Technique with Wait-Free Population Oblivious Reads. https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/left-right-2014.pdf, 2014.

15 Ramalhete, P., and Correia, A. Brief Announcement: Left-Right- A Concurrency Control Technique with Wait-Free Population Oblivious Reads. *Distributed* (2015), 663.

16 Ramalhete, P., and Correia, A. Concurrencyfreaks github repository. https://github.com/pramalhe/ConcurrencyFreaks/tree/master/Java/com/concurrencyfreaks/readindicators, 2015.

17 Ramalhete, P., and Correia, A. Poster: Poor man's urcu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2017), ACM, pp. 451–452.

A Comparison with URCU-Lib algorithms

The `userspace-rcu` library (URCU-Lib for short) contains several algorithms for implementing userspace RCU synchronization. We did not compare with the QSBR flavor of RCU because it requires each active reader thread to periodically call `rcu_quiescent_state()`, a somewhat limiting API for most applications. We also did not compare with the Signal flavor of RCU because POSIX style signals need to be supported by the Operating System, thus limiting its applicability. We compared our algorithms with the BulletProof and MemoryBarrier flavors of RCU because these variants require just atomic instructions and memory fences.

A.1 URCU-Lib Bullet Proof

Of the two generic algorithms compared in the URCU Library, Bulletproof is the more versatile, allowing it to be used without the user explicitly doing prior thread registration. Source code for the methods `rcu_read_lock()` and `rcu_read_unlock()` can be seen here `https://github.com/urcu/userspace-rcu/blob/master/include/urcu/static/urcu-bp.h` while `synchronize_rcu()` can be seen here `https://github.com/urcu/userspace-rcu/blob/master/src/urcu-bp.c`.

The first thing to notice is that `rcu_read_lock()` will internally call `rcu_bp_register()` to register a reader thread (line 169 of `urcu-bp.h`) if it is not registered already. This requires holding a mutex lock named `rcu_registry_lock`, but this is a one-time call. In applications with thread creation and destruction, it is imperative that each thread does its own de-registration before being deleted, otherwise the list of threads will grow indefinitely, dragging down the throughput when calling `synchronize_rcu()` and needlessly consuming memory. Our TwoPhase URCU with RICountersArray, which requires no prior thread registration, has no such issue.

In Bulletproof, the call to `rcu_read_lock()` has potentially two full fences (`MFENCE` instruction on x86), as can be seen in lines 149 and 170 of `urcu-bp.h`. In both of our URCU algorithms there is one single fence in `rcu_read_lock()`, implemented as either a full fence or as an atomic increment (`XADD` on x86).

In Bulletproof, the call to `rcu_read_unlock()` has two full fences. In our GraceVersion algorithm, there is no fence on `rcu_read_unlock()`, while on the TwoPhase algorithm there is a single fence in the form of atomic increment when using a ReadIndicator with counters.

This difference in fences may seem minor, but it is likely the sole responsible for the performance difference observed on the top plot of Figure 1.

Regarding the `synchronize_rcu()` implementation in Bulletproof in `urcu-bp.c`, it grabs two mutual exclusion locks, named `rcu_gp_lock` and `rcu_registry_lock` (lines 271 and 273), releasing them only near the end of the call. Notice that `rcu_registry_lock` is the same lock required by the internal thread registration called from `rcu_read_lock()`, meaning that in practice, a thread calling `synchronize_rcu()` is capable of blocking readers that have called `rcu_read_lock()` for the first time, i.e. the call to `rcu_read_lock()` in BulletProof is blocking. The `rcu_read_lock()` in the TwoPhase algorithm is lock-free when using a ReadIndicator with counters and on a platform that does not have a native atomic increment instruction (in all other cases it is wait-free). None of our URCU algorithms grab any mutual exclusion lock in their calls to `synchronize_rcu()`.

Besides the synchronization required to acquire two mutual exclusion locks, the `synchronize_rcu()` method in BulletProof executes 4 full fences (lines 281, 295, 312 and 330 of `urcu-bp.c`). Both the GraceVersion and TwoPhase URCUs have a single synchronized instruction in their calls to `synchronize_rcu()`, when executing the `compare_exchange_strong()` (in line 21 of Algorithm 1 or line 25 of Algorithm 2 respectively).

Because most of the code in `synchronize_rcu()` in BulletProof is executed with a mutual exclusion lock held, only one thread at a time is effectively executing it, preventing sharing of the grace period. Moreover, the BulletProof algorithm was not designed to share grace periods and would not work correctly without a mutual exclusion lock. This inability to share grace periods is the reason why the data points for BulletProof remain flat on the bottom rightmost plot of Figure 1. Without sharing the grace period, each thread calling `synchronize_rcu()` must first wait for the other callers of `synchronize_rcu()` to complete, before itself can wait for the (then) current ongoing readers.

The GraceVersion and TwoPhase URCUs are capable of sharing grace periods and therefore multiple threads calling `synchronize_rcu()` can work simultaneously, without having to wait for each other, waiting only for the current ongoing readers. The more threads we add the more work can complete, thus the linear scaling in the bottom rightmost plot of Figure 1 for these two algorithms.

A.2 URCU-Lib Memory Barrier

Similarly to BulletProof, the Memory Barrier (the default) algorithm in URCU-Lib uses atomics and fences, although it requires thread registration. Source code for the methods `rcu_read_lock()` and `rcu_read_unlock()` can be seen here `https://github.com/urcu/userspace-rcu/blob/master/include/urcu/static/urcu.h` while `synchronize_rcu()` can be seen here `https://github.com/urcu/userspace-rcu/blob/master/src/urcu.c`.

In MemoryBarrier, the call to `rcu_read_lock()` has potentially two full fences, as can be seen in lines 223 and 203 of `urcu.h`. The call to `rcu_read_unlock()` has at least one full fence (line 261) though potentially three full fences (lines 261, 240 and 242).

Relative to BulletProof, this extra fence seems to be responsible for the slightly lower throughput in the top plot of Figure 1.

In terms of synchronization, the `synchronize_rcu()` implementation in MemoryBarrier in `urcu.c`, does the following synchronized operations: adds to a stack using an atomic exchange (`XCHG` in x86, line 392); grabs the mutual exclusion lock of `rcu_gp_lock` (line 402); moves the waiters using a single atomic exchange (line 407); executes 6 full fences (line 420, 436, 443, 454, 462 and 481); Other synchronized operations may occur in uncommon code paths.

In MemoryBarrier it is possible to share grace periods, in which case a single atomic exchange and full fence will be needed and the code path is much simpler and faster (lines 392 to 397 of `urcu.c`). Because of this, the MemoryBarrier has nearly the same throughput as our URCUs on the bottom rightmost plot of Figure 1, providing some scalability in such scenarios.

However, looking at the bottom leftmost plot of Figure 1 the throughput of MemoryBarrier is extremely low when compared with GraceVersion and TwoPhase. The reason for this high difference in the throughput of synchronize_rcu() is due to the large overhead in the call of synchronize_rcu() in MemoryBarrier, because it uses a `futex` system call to wait for the thread that does the grace period validation to complete and *notify* the other waiting threads that it is now safe to continue. Futexes are blocking calls, while GraceVersion and TwoPhase URCUs use only non-blocking function calls. In both GraceVersion and TwoPhase, the only thing that may cause a call to synchronize_rcu() to block, is a reader that has not yet finished, while on MemoryBarrier even though the grace period may be shared, it is up to a single thread to complete the grace period and notify the other threads when it is safe to proceed, thus causing all threads that called synchronize_rcu() (except one) to block, waiting for the one thread.

B Fences and the C11 / C++ Memory Model

The Memory Model and respective atomics API introduced in C11 and C++11 allows software developers to write code that is correct in any hardware architecture and CPU, so long as there is a C11 or C++11 compiler for that specific platform.

Under these memory models, there is no need to add fences to the code (sometimes called barriers). That job is up to the compiler, which will introduce the appropriate fences where needed. In C++, when calling methods on atomic variables, such as `load()` and `store()`, the default memory ordering is `std::memory_order_seq_cst`.

For example, in line 9 of Algorithm 1, when compiling for an x86 target, the compiler generates a store (`MOV` instruction) followed by an `MFENCE` instruction, thus preventing re-ordering at the CPU level with the following load of `updaterVersion` in line 10.

Another example, in line 15 of the same algorithm, no hardware fence will be generated on x86 for the store, but the compiler will generate the store without re-ordering it with any preceding instruction (though it might decide to re-order it with following instructions) because an ordering of `std::memory_order_release` was chosen. Any user code preceding the call to `rcu_read_unlock()` will not be re-ordered with the store in line 15, thus guaranteeing the expected semantics for this method, i.e. similar to the `read unlock` of a reader-writer lock.

Although it may not be obvious at first sight, we designed our two URCU algorithms such that under the C++ memory model they use the least amount of fences, thus requiring the least amount of synchronization that we could devise. Looking at Algorithm 1 we see that at least on x86 the only fence will be the one generated by the compiler for line 9, and no further synchronized instruction is required. This equals the result shown by Leslie Lamport that a locking method must have at least a *load-store fence*. As for the unlock in line 15, there is no synchronized instruction being generated on x86, seen that a *release store* equals a regular store in this architecture.

Regarding the TwoPhase URCU shown in Algorithm 2 again a single synchronized instruction is done in the call to `arrive()` in line 8, being the `XADD or LOCK ADD` when using a ReadIndicator with counters, or a `MOV` followed by `MFENCE` for the ReadIndicator with one entry per thread (line 17 of Algorithm 5). For the `rcu_read_unlock()`, the ReadIndicators using counters unfortunately require a synchronized instruction to decrement the counter atomically (line 6 of Algorithm 3 and line 20 of Algorithm 4), while for the ReadIndicator PerThread a release store suffices (regular store on x86), as shown in line 23 of Algorithm 5.

All of this means that the read-side critical section in both our URCU algorithms have the least amount of synchronization theoretically possible, and although other algorithms can be devised in the future that have equally low synchronization for readers (and updaters), based on the result by Leslie Lamport, it is impossible to achieve lower synchronization.

C Differences between GraceVersion and TwoPhase

Although our two URCU algorithms GraceVersion and TwoPhase are both based on epoch-like mechanisms and both allow the sharing of grace periods, they have a few differences on how they share and advance the grace period.

In GraceVersion, when multiple updaters attempt to advance the `updaterVersion`, although only one of them will be successful, the unsuccessful updaters know that one of the other updaters has advanced the version, and therefore, advanced the grace period. This works as a kind of *helping mechanism* between simultaneous updaters.

For TwoPhase (Algorithm 2), for scenarios with concurrent updaters, it can happen that one of the updaters succeeds in advancing the `updaterVersion` before the others. The other updaters will see this change simply by reading the current value of `updaterVersion` (lines 20, 21, 23, and 28). This benefits the other updaters because it means they can skip one or more of the two validations needed to complete the call to `synchronize_rcu()`, namely, skip either of the calls to `isEmpty()` (lines 19 or 27). As such, the updaters are more prone to helping each other advance in TwoPhase than they are for GraceVersion.

The simplicity of both algorithms means that the code presented in this paper can be copied into any codebase, such that software developers using C11, C++1x, or D, no longer have to link with URCU libraries to be able to use a fast and scalable URCU in their code.