Grace Sharing Userspace-RCU

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
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 can be 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 the grace periods*. This last property is interesting because without it, a large number of concurrent calls to `synchronize_rcu()` can create a bottleneck.

We present two novel algorithms that are simple, provide wait-free progress for readers, and allow threads calling `synchronize_rcu()` to share a grace period. They implement the core APIs of RCU: `rcu_read_lock()`, `rcu_read_unlock()`, and `synchronize_rcu()`, which is enough for most applications. We executed a simplistic microbenchmark that shows these algorithms scale well even in high contention scenarios.

Categories and Subject Descriptors D.4.1.f [*Operating Systems*]: Synchronization

General Terms RCU, atomics, memory model

Keywords RCU, atomics, memory model

1. Introduction

When implementing lock-free and wait-free 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 [1, 2, 5, 6, 9, 10] with RCU [5, 9] 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 condition, and the other its low impact on latency with high scalability. RCU is unfortunately not a fully generic technique because it requires support in the operating system.

Fortunately, there is Userspace RCU (URCU) [5] which has 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 specific memory barriers for certain architectures, and although many architectures are supported, the support is not universal, and other languages may have difficulty linking with the library.

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 simple URCU algorithms [12] they allow for sharing of the grace period in `synchronize_rcu()`. Our algorithms implement the core functionality of RCU: `rcu_read_lock()`, `rcu_read_unlock()`, and `synchronize_rcu()`.

2. Grace Version Algorithm

In this technique, shown in Algorithm 1, there is a global version, named `updaterVersion`, and there is 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 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 not 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 that the reader is not currently active.

Algorithm 1 GraceVersion Grace Sharing URCU in C++14

```
1  class URCUGraceVersion {
2    static const uint64_t NOT_READING = 0xFFFFFFFFFFFFFFFE;
3    std::atomic<uint64_t> updaterVersion { 0 };
4    std::atomic<uint64_t> readersVersion[MAX_THREADS];
5
6  public:
7    int rcu_read_lock(const int tid) {
8      const uint64_t rv = updaterVersion.load();
9      readersVersion[tid].store(rv);
10     const uint64_t nrv = updaterVersion.load();
11     if (rv != nrv) readersVersion[tid].store(nrv, std::memory_order_relaxed);
12   }
13
14   void rcu_read_unlock(const int tid) {
15     readersVersion[tid].store(NOT_READING, std::memory_order_release);
16   }
17
18   void synchronize_rcu() {
19     const uint64_t waitForVersion = updaterVersion.load()+1;
20     uint64_t tmp = waitForVersion-1;
21     updaterVersion.compare_exchange_strong(tmp, waitForVersion);
22     for (int i=0; i < MAX_THREADS; i++) {
23       while (readersVersion[i].load() < waitForVersion) { // spin }
24     }
25   }
26 };
```

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* (threads calling `synchronize_rcu()`) with very little cost to readers.

The GraceVersion algorithm allows multiple threads calling `synchronize_rcu()` to share a grace period [8]. 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 read-side critical sections (lines 8 and 13 of Algorithm 2). The RI instance to increment is chosen based on the last bit of the `updaterVersion` variable (line 7), which is atomically incremented by any 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. As explained in the GraceVersion algorithm, this mechanism allows for multiple updaters to share the same grace period, which 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 that all reads 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 the corresponding grace period.

In the code shown in Algorithm 2, we allow for any ReadIndicator [3, 7, 11], 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.

Algorithm 2 Two-Phase Grace Sharing URCU in C++14

```
1  template<typename RI> class URCUTwoPhase {
2    std::atomic<int64_t> updaterVersion { 0 };
3    RI readIndicator[2];
4
5  public:
6    int rcu_read_lock() {
7        const int index = (int)(updaterVersion.load() & 1);
8        readIndicator[index].arrive();
9        return index;
10   }
11
12   void rcu_read_unlock(const int index) {
13       readIndicator[index].depart();
14   }
15
16   void synchronize_rcu() {
17       const int64_t currUV = updaterVersion.load();
18       const int64_t nextUV = (currUV+1);
19       while (!readIndicator[(int)(nextUV&1)].isEmpty()) { // spin
20           if (updaterVersion.load() > nextUV) return;
21           if (updaterVersion.load() == nextUV) break;
22       }
23       if (updaterVersion.load() == currUV) {
24           auto tmp = currUV;
25           updaterVersion.compare_exchange_strong(tmp, nextUV);
26       }
27       while (!readIndicator[(int)(currVI&1)].isEmpty()) { // spin
28           if (updaterVersion.load() > nextUV) return;
29       }
30   }
31 };
```

4. Performance and Discussion

Figure 1 shows the median for 5 runs of the number of operations per second, per number of threads, on a microbenchmark that iterates for 20 seconds on each run. 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 leftmost 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 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 scenario is ideal to test the effects of grace sharing on each URCU implementation.

Figure 1 shows a comparison of our GraceVersion and Two-Phase algorithms in C++ versus the the Bullet-Proof and default implementations in the URCU libray [4], shown as `URCU-Lib-BulletProof` and `URCU-Lib-default`. As seen from Figure 1, our GraceVersion algorithm surpasses the default and Bullet Proof 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 (middle plot), and matches the default URCU-Lib for long read operations (right-side plot). Unlike the Two-Phase algorithm, GraceVersion requires a static assignment of each thread to an entry of an array, making its usage inflexible. In summary, our algorithms are better than the Bullet Proof URCU in all tested scenarios, sometimes going up to a 100x improvement, and they are always as good as, or better than, the default URCU in the userspace-rcu library.

Both 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.

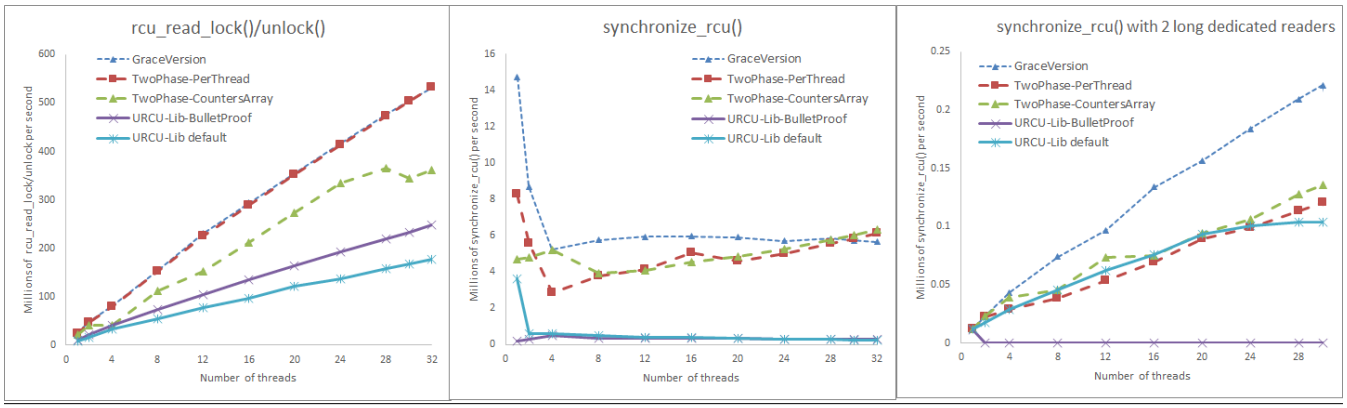

Figure 1. Comparison of four different URCU implementations: our GraceVersion, our TwoPhase using an entry per thread or arrays of counters as ReadIndicators, the Userspace-RCU library's Bullet Proof, and default implementations. The leftmost plot shows read-only operations. The middle plot show measurements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

The adaptability of the Two-Phase 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 be made depending 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 starvation-free and it is not blocked by other updaters, and may share a grace period which increases scalability. The simplicity of both algorithms means that the code presented in this paper can be pasted into any codebase, and 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.

References

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

[2] A. Braginsky, A. Kogan, and E. Petrank. 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*, pages 33–42. ACM, 2013.

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

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

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

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

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

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

[9] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

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

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

[12] P. Ramalhete and A. Correia. Brief announcement: Left-right-a concurrency control technique with wait-free population oblivious reads. *Distributed*, page 663, 2015.