TOWARDS A HIGH-LEVEL C++ ABSTRACTION TO UTILIZE THE READ-COPY-UPDATE PATTERN

Gábor MÁRTON*, Imre SZEKERES**, Zoltán PORKOLÁB***

*Department of Programming Languages and Compilers, Faculty Informatics, Eötvös Loránd University, H-1117 Pázmány Péter sétány 1/C, E-mail: martongabesz@gmail.com
**Budapest University of Technology and Economics, Hungary, E-mail: iszekeres.x@gmail.com
***Department of Programming Languages and Compilers, Faculty Informatics, Eötvös Loránd University, H-1117 Pázmány Péter sétány 1/C, E-mail: gsd@elte.hu

ABSTRACT

Concurrent programming with classical mutex/lock techniques does not scale well when reads are way more frequent than writes. Such situation happens in operating system kernels among other performance critical multithreaded applications. Read copy update (RCU) is a well know technique for solving the problem. RCU guarantees minimal overhead for read operations and allows them to occur concurrently with write operations. RCU is a favourite concurrent pattern in low level, performance critical applications, like the Linux kernel. Currently there is no high-level abstraction for RCU for the C++ programming language. In this paper, we present our C++ RCU class library to support efficient concurrent programming for the read-copy-update pattern. The library has been carefully designed to optimise performance in a heavily multithreaded environment, in the same time providing high-level abstractions, like smart pointers and other C++11/14/17 features.

Keywords: C++, threads, synchronisation, RCU

1. INTRODUCTION

Read-copy-update is a concurrent design pattern [1][2] which allows extremely low run-time overhead for readers. Updates can happen concurrently with reads as they leave the old versions of the data structure intact; this way the pre-existing readers can finish their work. Thus, updates might require more overhead than reads and their effect might be delayed. In contrast to readers-writers lock [3] RCU does not block the writers if there are concurrent readers.

Classical RCU first appeared in the Linux kernel in 2002 [4][5]. It provides the following reader side primitives: rcu_read_lock() and rcu_read_unlock(). Read-side critical sections may use rcu_dereference() to access RCU protected pointers.

On the update side we may use the synchronize_rcu() primitive and rcu_assign_pointer() to assign values to protected pointer. Pointers stored by rcu_assign_pointer() can be fetched from within read-side critical sections by rcu_dereference().

The pseudo code in Figure 1 demonstrates how these primitives can be used to implement the lookup and the remove operations on a simple linked list of key-value pairs. This implementation is a simplified excerpt of McKenney’s pre-BSD routing table example [5]. With rcu_read_lock() and rcu_read_unlock() we indicate the reader side critical section. In this read-side critical section we traverse through the list (find()) and once we found the key we return with the associated value. In the implementation of find() we have to use rcu_dereference() to access the elements in the list. It might happen that the key is not in the list, in that case we again close the critical section and then return with a special value indicating the element is not in the list.

In remove() we have to use a spin lock in order to protect the list from concurrent write operations. The block which is protected by the spin lock is the write-side critical section. We iterate over the list trying to find the key and if we found it then we unlink (remove_node()) it from the list. In the realization of the remove_node() we have to use the rcu_assign_pointer() primitive. After the removal, with the synchronize_rcu() primitive we wait all pre-existing RCU read-side critical sections to completely finish. Then we can deallocate the list node which is no longer needed and this way can close the write-side critical section by releasing the lock.

```
void remove (List list, Key key) {
  Node* node;
  synchronize_rcu();
  if (node = find(list, key)) {
    remove_node(list, node);
    spin_unlock(lock);
  } else {
    return not_found;
  }
}
```

```
void remove (List list, Key key) {
  Node* node;
  spin_lock(lock);
  if (node = find(list, key)) {
    remove_node(list, node);
    spin_unlock(lock);
  } else {
    return;
  }
}
```

Fig. 1 Usage of RCU in a linked list

Classic RCU requires that read-side critical sections obey the same rules obeyed by the critical sections of pure spinlocks: blocking or sleeping of any sort is strictly prohibited. Since 2002 many different RCU flavours have appeared in the Linux kernel which relax this strict require-
ment. Using realtime RCU [6–8] read-side critical sections may be preempted and may block while acquiring spin-locks. Sleepable RCU allows more, it permits arbitrary sleeping (or blocking) within RCU read-side critical sections [9,10].

The different RCU flavours in the Linux kernel are naturally dependent on the kernel internals, for example on the scheduler. Obviously they cannot be used in user space. Userspace RCU (URCU) [11,12] was created by Desnoyers in 2009 and has a similar API to the kernel space RCU flavours. URCU has different variants and implementations. For instance the Quiescent-State-Based Reclamation RCU (QSBR) provides near-zero overhead but the price of minimal overhead is that each thread in an application is required to periodically invoke rcu_quiescent_state() to announce that it resides in a quiescent state [13]. The general-purpose user space realization can be used in applications where we cannot guarantee that each threads will invoke rcu_quiescent_state() sufficiently often. However, this versatility has its own price, general-purpose RCU has to use memory barriers in the read-side. A third variant uses POSIX signals to eliminate these barriers, obviously this flavour cannot be used on non-POSIX systems.

URCU has been proposed to be incorporated into the C and C++ standard with the C API provided by Desnoyers realization [14]. URCU provides a low level C API, therefore it is more prone to errors in C++ programs than a well established high-level C++ API can be. For instance, it is easy to forget to call rcu_read_unlock() on all return paths. In URCU there is no automatic memory reclamation: to deallocate memory, first we have to use the synchronize_rcu() primitive. (Note that besides Desnoyers realization there are a surprisingly large number of other lesser known userspace RCU implementations, and more are being created all the time. E.g. [15,16].)

In this paper we present an alternative implementation for user space RCU as a C++ smart pointer, thus there is no need to manually deallocate memory. Our realization provides a high-level abstraction C++ API to the users, so they can use a simple construct which is not prone to errors, still its performance is satisfying for most of the use cases. Our paper is organized as follows. In section 2 we present the steps which lead from using a mutex to the concept of a high-level smart pointer for the RCU semantics. We describe the details and difficulties with the implementation of the smart pointer in section 3. Section 4 contains the description of our testing methods. We write about ongoing and future work in section 5. Our paper concludes in 6.

2. TOWARDS A HIGHER LEVEL ABSTRACTION FOR RCU

Let us suppose we have a collection that is shared among multiple readers and writers in a concurrent manner (Figure 2). It is a common way to make the collection thread safe by holding a lock until the iteration is finished (on the reader thread). This approach does not scale well, especially when reads are way more frequent than writes [5]. Instead of a simple lock_guard we could use a readers-writers lock [3], but that would scale badly as well, especially when we have multiple concurrent writers [5].

The first idea to make it better is to have a shared pointer and hold the lock only until that is copied by the reader or updated by the writer (Figure 3).

```c
class X {
    std::vector<int> v;
    mutable std::mutex m;
  public:
    int sum() const { // read operation
        std::lock_guard<std::mutex> lock(m);
        return std::accumulate(v.begin(), v.end(), 0);
    }
    void add(int i) { // write operation
        std::lock_guard<std::mutex> lock(m);
        v.push_back(i);
    }
};
```

Fig. 2 A shared collection

```c
class X {
    std::shared_ptr<std::vector<int>> v;
    mutable std::mutex m;
  public:
    int sum() const { // read operation
        std::lock_guard<std::mutex> lock(m);
        local_copy = v;
        // assume processing the data takes longer
        // than copying it
        return std::accumulate(local_copy->begin(),
                                local_copy->end(), 0);
    }
    void add(int i) { // write operation
        std::lock_guard<std::mutex> lock(m);
        local_copy.push_back(i);
        v = local_copy;
    }
};
```

Fig. 3 Using a shared pointer in the collection

```c
void add(int i) { // write operation
    std::shared_ptr<std::vector<int>> local_copy;
    std::lock_guard<std::mutex> lock(m);
    local_copy = v;
    local_copy->push_back(i);
    v = local_copy;
}
```

Fig. 4 Deep copy
Now we have a race on the pointee itself during the write. So we need to have a deep copy (Figure 4). The copy construction of the underlying data (vector<int>) is thread safe, since the copy constructor parameter is a constant reference to vector<int>.

Still, there is one more problem: if there are two concurrent write operations then we might miss one of them. We should check whether the other writer has done an update after the actual writer has loaded the local copy. If it did then we should load the data again and try to do the update again. This leads to the idea of using an atomic_compare_exchange in a while loop. We could use an atomic_shared_ptr if that was included in the current C++ standard, but until then we have to be satisfied with the free function overloads for shared_ptr (Figure 5). These free function overloads take a simple shared_ptr as a parameter and perform the specific atomic operations:

```cpp
template <class T>
std::shared_ptr<T> atomic_load(
    const std::shared_ptr<T> *p);

template <class T>
bool atomic_compare_exchange_strong(
    std::shared_ptr<T> * p,
    std::shared_ptr<T> * expected,
    std::shared_ptr<T> * desired);
```

Note, atomic_shared_ptr class template which would replace these free functions might be included in the C++20 standard [17].

```cpp
class X {
    std::shared_ptr<vector<int>> v;

    public:
        X() : v(std::make_shared<vector<int>>()) {}
        int sum() const {
            return std::accumulate(v->begin(), v->end(), 0);
        }
        void add(int i) {
            auto local_copy = std::atomic_load(&v);
            return std::accumulate(local_copy->begin(), local_copy->end(), 0);
        }

    void another_sum() const {
        return std::accumulate(v->begin(), v->end(), 0);
    }
};
```

This is definitely a race condition and a problem. To avoid this user error and to hide the sensitive technical details we created a smart pointer which we named as rcu_ptr. This smart pointer provides a general higher level abstraction above atomic_shared_ptr. Figure 6 represents how we can use rcu_ptr in our running example.

```cpp
class X {
    rcu_ptr<vector<int>> v;

    public:
        X() : v(std::make_shared<vector<int>>()) {}
        int sum() const {
            return std::accumulate(v->begin(), v->end(), 0);
        }
        void add(int i) {
            auto local_copy = v->read();
            return std::accumulate(local_copy->begin(), local_copy->end(), 0);
        }

    void another_sum() const {
        return std::accumulate(v->begin(), v->end(), 0);
    }
};
```

Fig. 6 Usage of rcu_ptr

The read() method of rcu_ptr returns a shared_ptr<const T> by value, therefore it is thread safe. The existence of the shared_ptr in the scope enforces that the read object will live at least until this read operation finishes. By using the shared pointer this way, we are free from the ABA problem [20-21] since the memory address associated with the object cannot be reused until the object itself is reclaimed [22]. The copy_update() method receives a lambda. This lambda is called whenever

---

1 Spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g., load-locked store-conditional machines [18].
an update needs to be done, i.e. it will be called continuously until the update is successful. The lambda receives a T* for the copy of the actual data. We can modify the copy of the actual data inside the lambda.

3. SMART POINTER FOR RCU SEMANTICS

In Figure[7] we present the simplified implementation of the rcu_ptr class template. The complete implementation is available and free to use at [23].

```cpp
template <typename T> class rcu_ptr {
    std::shared_ptr<const T> sp;
public:
    rcu_ptr() = default;
    "rcu_ptr"() = default;
    rcu_ptr(rcu_ptr &&rhu) = delete;
    rcu_ptr &
    operator=(rcu_ptr &&) = delete;
    rcu_ptr &operator=(rcu_ptr &rhu) = delete;
    rcu_ptr(const std::shared_ptr<const T> &sp_)
        : sp(sp_) {}
    rcu_ptr(const std::shared_ptr<const T> & &sp_)
        : sp(std::move(sp_)) {}
    std::shared_ptr<const T> read() const {
        return std::atomic_load_explicit(
            &sp, std::memory_order_consume);
    }
    void copy_update(R &&fun) {
        std::shared_ptr<T> r;
        if (sp_l) {
            // deep copy
            r = std::make_shared<T>(*sp_l);
        }
        // update
        std::forward<R>(fun)(r.get());
    }
    template <typename R>
    void copy_update(R &fun) {
        std::shared_ptr<const T> sp_l =
            std::atomic_load_explicit(
                &sp, std::memory_order_acquire);
        do {
            std::shared_ptr<T> r;
            if (sp_l) {
                // deep copy
                r = std::make_shared<T>(*sp_l);
            }
            // update
            std::forward<R>(fun)(r.get());
        } while (;
            atomic_compare_exchange_strong_explicit(
                &sp, &sp_l,
                std::shared_ptr<const T>(
                    std::move(r),
                    std::memory_order_release,
                    std::memory_order_consume));
    }
};
```

Fig. 7 The rcu_ptr class template

We provide a default constructor and a default destructor (lines 5 and 6). The move and copy operations are deleted (lines 8-12) because rcu_ptr is essentially a wrapper around an atomic type (we plan to support atomic_shared_ptr as soon as it is included in the standard). And all atomic types are neither copyable nor movable (because there is no sense to assign meaning for an operation spanning two separately atomic objects) [24][25].

We can create an rcu_ptr from an Ivalue or rvalue reference of shared_ptr< const T> (lines 14-17). These functions just simply copy or move their parameter into the member shared_ptr. There is no need to make these constructors thread safe, because the construction can be done only by one thread.

Lines 24-33 is the realization of the reset() methods which receive a shared_ptr< const T> as an Ivalue or rvalue reference parameter. We can use it to reset the wrapped data to a new value independent from the old value (e.g. vector.clear()). Actually, with the parameter we overwrite the currently contained shared_ptr. The overwrite has to be an atomic operation in order to protect the member from concurrent reset() calls.

In lines 19-22, the read() method atomically loads the member shared_ptr and returns with a copy of that. The copy_update() function template (lines 35-60) receives an rvalue reference to an instance of a callable type. First we create a local copy of the member as ap_1 (lines 38-40). If this local copy is set (i.e the rcu_ptr instance is initialized) then we create a deep copy, that is we copy the pointee itself and we create a new shared_ptr<T> (denoted as r) pointing to the copy (lines 44-47). Note, that this is a non-constant shared pointer. On line 50 we call the callabe and we pass a non-constant pointer to the new copy as a parameter. Then in lines 53-59 we exchange the member shared pointer with a shared_ptr to the deep copy if we find that the member still points to the same object of which we created the copy. If it turns out that is not the case (i.e. another thread was faster), then we repeat the whole deep copy update sequence until we succeed (line 43). The callers of the copy_update() function must be aware that in case of an unset (or default initialized) rcu_ptr the callable will be called with a null pointer as an argument. Also, a call expression with this function is invalid, if the wrapped data type T is a non-copyable type.

3.1. Memory Ordering

A memory_order_release store is said to synchronize with a memory_order_acquire load if that load returns the value stored or in some special cases, some later value [18][26]. When a memory_order_release store synchronizes with a memory_order_acquire load, any memory reference preceding the memory_order_release store will happen before any memory reference following the memory_order_acquire load [18][26]. This property allows a linked structure to be locklessly traversed by using memory_order_release stores when updating pointers to reference new data elements and by using memory_order_acquire loads when loading pointers while locklessly traversing the data structure [26]. A memory_order_release store is dependency ordered before a memory_order_consumes load when that load returns the value stored, or in some special cases, some later value [18][26]. Then, if the load carries a dependency to some later memory reference, any mem-
ory reference preceding the memory_order_release statement will happen before that later memory reference \[26,26\] [26]. This means that when there is dependency ordering, memory_order_consume gives the same guarantees that memory_order_acquire does, but possibly at lower cost [26].

In the classical RCU, the rcu_dereference() primitive implements the notion of a dependency ordered load, which suppresses aggressive code-motion compiler optimizations and generates a simple load on any system other than DEC Alpha, where it generates a load followed by a memory-barrier instruction. The rcu_assign_pointer() primitive implements the notion of store release, which on sequentially consistent and total-store-ordered systems compiles to a simple assignment [17].

In our implementation of rcu_ptr::copy_update() function we can also use the release and consume semantics. We cannot use relaxed ordering because in case of that if the fun is inlined and fun itself is not an ordering operation or it does not contain any fences then the load or the compare_exchange might be reordered into the middle of fun. Also we need to “see” the latest updates so we can copy and update the “most recent” version. Though, there is a data dependency chain: mp_l->r->compare_exchange(..,r). So if all the architectures were preserving data dependency ordering, than we would be fine with relaxed. However, some architectures do not preserve data dependency ordering (e.g. DEC Alpha), therefore we need to explicitly state that we rely on that neither the CPU nor the compiler will reorder data dependent operations. This is what we express with the consume-release semantics. Consequently, during all the atomic load operations in the rcu_ptr class template we can use memory_order_consume and during all atomic store operations (including the read-modify-write operation) we use memory_order_release. If the definition of the fun callable is unseen by the compiler (i.e. it is defined in an other translation unit) then the user have to annotate the declaration of the callable with the \[[\text{carries\_dependency}]\] attribute [18]. Otherwise, the compiler may assume that the dependency chain is broken during the call and consequently it would fall back to the safer but less efficient acquire semantics [18].

Unfortunately the consume memory order is temporarily deprecated in C++17. It is widely accepted that the current definition of memory_order_consume in the C++11/14 standard is not useful. All current compilers essentially map it to memory_order_acquire. The difficulties appear to stem both from the high implementation complexity and from the fact that the current definition uses a fairly general definition of “dependency” [26,27]. As such, the consume ordering has to be redefined. While this work is in progress, hopefully ready for the next revision of C++, users are encouraged to not use this ordering and instead use acquire ordering, so as to not be exposed to a breaking change in the future. As for our rcu_ptr, in order to reach the consume semantics we may use hardware specific instructions in the future to overcome the mentioned problem.

3.2. Lock Free atomic_shared_ptr

Our rcu_ptr can be used with the free functions overloads of the atomic_ prefix [18] section 20.8.2.6 for std::shared_ptr. Since the atomic_shared_ptr [17] is still in experimental phase, we use our own wrapper template class around the free functions. The free functions are implemented in terms of a spinlock in the currently available standard libraries. Having a lock-free atomic_shared_ptr would be really beneficial. However, implementing a lock-free atomic_shared_ptr in a portable way can have extreme difficulties [28]. Though, it is easier on architectures where the double word CAS operation is available as a CPU instruction as we can see that with Anthony Williams implementation [29]. We can use Williams’ implementation with our rcu_ptr class template as well if a double word CAS operation is available.

4. PERFORMANCE EVALUATION

We executed performance measurements on a dual CPU system (two Intel® Xeon® X5670 CPUs). Each CPU had 6 physical cores with hyper-threading enabled, this sums up to 24 threads. Also each CPU had 12MB cache. We used Ubuntu 14.04 operating system (Linux kernel 3.13).

We took the class X from the running example (presented in Figure 2) and slightly changed it:

```cpp
class X {
  std::vector<int> v;
  const int default_value = 1;
  mutable std::mutex m;

  public:
  X(size_t vec_size) : v(vec_size, default_value) {}
  int read_one(unsigned index) const {
    // read operation
    std::lock_guard<std::mutex> lock(m);
    return v[index];
  }
  void update_all(int value) {
    // write operation
    std::lock_guard<std::mutex> lock(m);
    for (auto &e : v) e = value;
  }
};
```

We added a constructor via which we can setup the size of the vector. We modified the read operation to read only one value from the vector. We also changed the write operation to update all elements in the vector. We implemented this modified class in terms of several different synchronization mechanisms:

- **std mutex.** Standard mutex from the C++ Standard Template Library (STL). We used the STL implementation libstdc++ from GNU Compiler Collection (version 5.4). On POSIX systems, std::mutex uses pthread_mutex_lock and pthread_mutex_unlock functions from the pthread library. On Linux, these pthread functions are implemented in terms of futex (fast userspace mutex) system call. It provides very fast uncontended lock acquisition and release. The futex state is stored in a user-space variable. Atomic operations are used in order to change the state of the futex in the uncontended case without the overhead of a syscall.
In the contended cases, the kernel is invoked to put tasks to sleep and wake them up.

- **tbb qrw mutex.** Intel® TBB queuing reader-writer mutex [31]. A queuing_rw_mutex is scalable, in the sense that if a thread has to wait to acquire the mutex, it spins on its own local cache line. A queuing_rw_mutex is fair. Threads acquire a lock on a queuing_rw_mutex in the order that they request it.

- **tbb srw mutex.** Intel® TBB spin reader-writer mutex [31]. A spin_rw_mutex is not scalable or fair. It is ideal when the lock is lightly contended and is held for only a few machine instructions. If a thread has to wait to acquire a spin_rw_mutex, it busy waits, which can degrade system performance if the wait is long. However, if the wait is typically short, a spin_rw_mutex significantly improves performance compared to other mutexes.

- **rcuptr.** Our rcu_ptr with non-lock-free atomic shared pointer. We use a wrapper template class which encapsulates the free function overloads for atomic operations on a standard shared_ptr.

- **rcuptr jss.** Our rcu_ptr with Anthony Williams’ lock-free atomic shared pointer [29]. Note that the examined Intel CPU has the double word CAS operation.

- **urcu bp.** Bulletproof version of the URCU library. We used the bulletproof version because that is the general version of URCU. The "bulletproof" version is the only one which can be used even when we cannot register individual threads with the URCU library.

We created a separate test binary for each mechanism. Each test binary consists of a timer thread which ticks approximately after one second, one writer thread and several reader threads (configurable number). As for the measure metrics we count how many times a reader or writer thread finishes its operation during the elapsed time period. The timer thread sets an atomic stop flag while all the other threads read this flag continuously and they stop when it is set. We used relaxed memory ordering for writing and reading this flag in order to make sure that the cache system is not affected by the measurement itself. We executed each test binary with different number of reader threads and with different vector sizes. We executed one test binary with a specific configuration (number of threads, vector size) five times. During the evaluation of each performance indicator value we dropped the smallest and the largest values and we took the average of the remaining three values. The measurement scripts and the source code for the test binaries are readily available at [32], thus our measurements are easily replicable on any other hardware.

We experienced that if the size of the vector is really small (smaller than 4KiB) then the read-side performance of the RCU mechanisms are outperformed by a simple standard mutex. However, as the data grows, the RCU mechanism is getting advantage over the standard mutex and over the read-write mutexes (Figure 8). In Figure 8 we display the performance of the different techniques when the size of

![Fig. 8 Read-side performance, data size: 32KiB](image)

![Fig. 9 Read-side performance, data size: 512KiB](image)

![Fig. 10 Read-side performance, data size: 4MiB](image)
the used data is 32KiB (i.e. the vector has 8192 elements). Note that the y axis presents a logarithmic scale. The x axis presents how many reader threads were active during the measurement. Similarly to Figure 8, Figure 9 and 10 show the read performance in case of 512KiB and 4MiB data size respectively. Figure 9 and 10 illustrate that our rcu_ptr implementation can outperform the traditional mutex based implementation with more than two orders of magnitude. Also, rcu_ptr can outperform the read-write mutex based realizations with more than one order of magnitude. The rcu_ptr based techniques have some degradation until the readers number is less than 5 (approximately). From that point, the performance has no or minimal degradation. This is in contrast to the read-write mutex and the URCU based methods, where the performance is growing continuously as the number of the readers grows. Compared to URCU, our technique can be outperformed up to two orders of magnitudes. This is the price we pay for the higher level of abstraction and for the general usability: we loose most of the performance because of the extra administration done with the reference counting in the underlying shared_ptr implementations while the bulletproof URCU uses only memory barrier instructions.

Figure 11 presents that RCU write-side performance is outperformed by the mutex variants (32KiB data size). This is the expected behaviour since RCU solutions are tuned for the read-side performance, but this implies some trade-offs on the write-side. However, our technique can outperform the bulletproof version of URCU in write-side performance. E.g. when the data size is 512 KiB then our method can be twice as fast (Figure 12). This is because with urcu bp one cannot use the call_rcu() to deallocate memory asynchronously, thus the writer thread must wait for all the pre-existing readers to be completed. This wait is done by synchronize_rcu() function and the duration actually waited is called an RCU grace period. Regarding to write-side performance we measured that all RCU based approaches are outperformed by the all the mutex based solutions. The difference can be up to 20x, based on the used RCU and mutex implementation and on the size of the data. Interestingly, our measurements show that the lock-free implementation of shared_ptr does not provide higher read or writer performance compared to the non lock-free version.

![Fig. 11 Write-side performance, data size: 32KiB](image1)

However, our technique can outperform the bulletproof version of URCU in write-side performance. E.g. when the data size is 512 KiB then our method can be twice as fast (Figure 12). This is because with urcu bp one cannot use the call_rcu() to deallocate memory asynchronously, thus the writer thread must wait for all the pre-existing readers to be completed. This wait is done by synchronize_rcu() function and the duration actually waited is called an RCU grace period. Regarding to write-side performance we measured that all RCU based approaches are outperformed by the all the mutex based solutions. The difference can be up to 20x, based on the used RCU and mutex implementation and on the size of the data. Interestingly, our measurements show that the lock-free implementation of shared_ptr does not provide higher read or writer performance compared to the non lock-free version.

![Fig. 12 Write performance of RCU, data size: 512KiB](image2)

5. CORRECTNESS AND TESTING

To validate the correctness of our data structure we used different testing methods. We executed unit tests in a sequential manner (i.e. no parallel execution) to validate the basic behaviour of the class template. We used oriented stress testing [33] and sanitizers from the LLVM/Clang infrastructure [34] to verify behaviour during concurrent execution. During our stress tests we focused on pairs of public methods of rcu_ptr and we executed these functions from different threads. We executed the operations in a loop on each thread and we added random delays in between each calls. This way we tested different execution timings and we could make race windows slightly larger.

6. FUTURE WORK

It is our ongoing work to create performance measurements of our rcu_ptr on a weekly ordered architecture like ARMv7 as well. In order to reach the consume semantics in rcu_ptr we may use hardware specific instructions in the future to overcome the problem of the deprecated memory_order_consume.

7. CONCLUSION

RCU is a technique in concurrent programming which is getting used more and more often nowadays. It has been introduced in the Linux kernel first, but the efficiency of the technique became proven so people demanded an implementation which could be used in user space too. The current available user space RCU solutions do not provide a mechanism for automatic memory reclamation, also they provide a low level C API, which may be prone to errors. In this paper we presented a high-level C++ implementation for the read-copy-update pattern, which provides automatic memory deallocation. Our technique complements the existing user space RCU implementation by providing a well
performing safe and hard-to-misuse library. Thus, this library may be a good default choice by C++ developers who expect more readers than writers in their application.

REFERENCES


Towards a High-level C++ Abstraction to Utilize the Read-Copy-Update Pattern


Received May 5, 2018, accepted July 10, 2018

BIOGRAPHIES

Gábor Márton received his M.Sc degree in 2007 in Information Technology from the Budapest University of Technology and Economics. Currently, he is a PhD candidate at the Department of Programming Languages and Compilers at the Faculty of Informatics, Eötvös Loránd University (ELTE), Budapest, Hungary. His research focuses on compiler technology and concurrency.

Imre Szekeres graduated (M.Sc) in 2015 at the Department of Computer Science and Information Theory of the Faculty of Electrical Engineering and Informatics at the Budapest University of Technology and Economics. He aims to start his PhD studies soon. Currently, he is working in algorithmic trading solutions; striving for low latency and efficiency in modern C++ in a concurrent environment.

Zoltán Porkoláb received his M.Sc and PhD degrees in Information Technology in 1988 and 2003, respectively. At present, he is an Associate Professor of the Department of Programming Languages and Compilers at the Faculty of Informatics, Eötvös Loránd University (ELTE), Budapest, Hungary. His main research topics are C++ metaprogramming and compiler technology.