High-level C++ Implementation of the Read-Copy-Update Pattern

Gábor Márton
Eötvös Loránd University,
Faculty of Informatics
Dept. of Programming Languages
and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
Email: martongabesz@gmail.com

Imre Szekeres
Budapest University of Technology
and Economics
Budapest, Hungary
Email: iszekeres.x@gmail.com

Zoltán Porkoláb
Eötvös Loránd University,
Faculty of Informatics
Dept. of Programming Languages
and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
Email: 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.**

I. Introduction

Read-copy-update is a concurrent design pattern [1], [2] which allows extremely low overhead for readers. Updates can happen concurrently with reads as they leave the old versions of the data structure intact; this way the already 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. 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.

```
SPINLOCK(lock);

Value lookup(List list, Key key) {
  Node* node;
  Value local_value;
  rcu_read_lock();
  // iterate over the list and return the value
  // of the found element
  if (node = find(list, key)) {
    local_value = node->value
    rcu_read_unlock();
    return local_value;
  }
  rcu_read_unlock();
  return not_found;
}

void remove (List list, Key key) {
  Node* node;
  spin_lock(lock);
  // iterate over the list and find the key
  if (node = find(list, key)) {
    remove_node(list, node);
    spin_unlock(lock);
    synchronize_rcu();
    free(node);
    return;
  }
  spin_unlock(lock);
}
```

Fig. 1. Usage of RCU in a linked 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.

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 requirement. Using realtime RCU [6]–[8] read-side critical sections may be preempted and may block while acquiring spinlocks. 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 in 2009 and has a similar API to the kernel space RCU flavours. Userspace RCU has different variants and implementations. For instance the Quiescent-State-Based Reclamation RCU (QSBR) provides near-zero read-side 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 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.

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 II 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 III. Section IV contains the description of our testing methods. We write about ongoing and future work in section V. Our paper concludes in VI.

II. 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].

```
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

```
class X {
  std::shared_ptr<std::vector<int>> v;
  mutable std::mutex m;

public:
  X()
      : v(std::make_shared<
            std::vector<int>>()) {}
  int sum() const { // read operation
    std::shared_ptr<std::vector<int>>
        local_copy;
    {
      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::shared_ptr<std::vector<int>>
        local_copy;
    {
      std::lock_guard<std::mutex> lock{m};
      local_copy = v;
    }
    local_copy->push_back(i);
    {
      std::lock_guard<std::mutex> lock{m};
      v = local_copy;
    }
  }
};
```

Fig. 3. Using a shared pointer in the collection

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). 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 had 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

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

Fig. 4. Deep copy

```
1  class X {
2    std::shared_ptr<std::vector<int>> v;
3
4  public:
5    X()
6        : v(std::make_shared<
7              std::vector<int>>()) {}
8    int sum() const { // read operation
9      auto local_copy = std::atomic_load(&v);
10     return std::accumulate(local_copy->begin(),
11                            local_copy->end(),
12                            0);
13   }
14   void add(int i) { // write operation
15     auto local_copy = std::atomic_load(&v);
16     auto exchange_result = false;
17     while (!exchange_result) {
18       // we need a deep copy
19       auto local_deep_copy =
20           std::make_shared<std::vector<int>>(
21               *local_copy);
22       local_deep_copy->push_back(i);
23       exchange_result =
24           std::atomic_compare_exchange_strong(
25               &v, &local_copy, local_deep_copy);
26     }
27   }
28 };
```

Fig. 5. Using atomic shared pointer

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:

```
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 [14]. Since both during the read operation and the write operation we do not modify the pointee the element type of the member `shared_ptr` can be changed to be a constant:

```
class X {
  std::shared_ptr<const std::vector<int>> v;
  // ...
};
```

In the write operation we do the update on the copy of the original pointee (line 22 of Figure 5) and not on the pointee

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

public:
  X()
      : v(std::make_shared<
            std::vector<int>>()) {}
  int sum() const { // read operation
    std::shared_ptr<const std::vector<int>>
        local_copy = v.read();
    return std::accumulate(local_copy->begin(),
                           local_copy->end(),
                           0);
  }
  void add(int i) { // write operation
    v.copy_update([i](std::vector<int> *copy) {
      copy->push_back(i);
    });
  }
};
```

Fig. 6. Usage of rcu_ptr

of the member.

We might notice that we can move construct the third parameter of `atomic_compare_exchange_strong`, therefore we can spare a reference count increment and decrement:

```
exchange_result =
    std::atomic_compare_exchange_strong(
        &v, &local_copy,
        std::move(local_deep_copy));
```

Regarding the write operation, since we are already in a while loop we could replace `atomic_compare_exchange_strong` with `atomic_compare_exchange_weak`. That can result in a performance gain on some platforms [15], [16]. However, `atomic_compare_exchange_weak` can fail spuriously[1]. Consequently, we might do the deep copy more often than needed if we used the weak counterpart.

In the current form of class `X` nothing stops an other programmer (e.g. a naive maintainer of the code years later) to add a new reader operation, like this:

```
int 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 can we use `rcu_ptr` in our running example. 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 [17], [18] since the memory address associated with the object cannot be reused until the object itself is reclaimed [19]. The `copy_update()` method receives a lambda. This lambda is called whenever an update needs to be done, i.e. it will be called continuously until the update is successful. The lambda

[1]Spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g., load-locked store-conditional machines [15]

```
1  template <typename T> class rcu_ptr {
2    std::shared_ptr<const T> sp;
3
4  public:
5    rcu_ptr() = default;
6    ~rcu_ptr() = default;
7
8    rcu_ptr(const rcu_ptr &rhs) = delete;
9    rcu_ptr &
10   operator=(const rcu_ptr &rhs) = delete;
11   rcu_ptr(rcu_ptr &&) = delete;
12   rcu_ptr &operator=(rcu_ptr &&) = delete;
13
14   rcu_ptr(const std::shared_ptr<const T> &sp_)
15       : sp(sp_) {}
16   rcu_ptr(std::shared_ptr<const T> &&sp_)
17       : sp(std::move(sp_)) {}
18
19   std::shared_ptr<const T> read() const {
20     return std::atomic_load_explicit(
21         &sp, std::memory_order_consume);
22   }
23
24   void
25   reset(const std::shared_ptr<const T> &r) {
26     std::atomic_store_explicit(
27         &sp, r, std::memory_order_release);
28   }
29   void reset(std::shared_ptr<const T> &&r) {
30     std::atomic_store_explicit(
31         &sp, std::move(r),
32         std::memory_order_release);
33   }
34
35   template <typename R>
36   void copy_update(R &&fun) {
37
38     std::shared_ptr<const T> sp_l =
39         std::atomic_load_explicit(
40             &sp, std::memory_order_consume);
41
42     std::shared_ptr<T> r;
43     do {
44       if (sp_l) {
45         // deep copy
46         r = std::make_shared<T>(*sp_l);
47       }
48
49       // update
50       std::forward<R>(fun)(r.get());
51
52     } while (
53       !std::
54         atomic_compare_exchange_strong_explicit(
55                 &sp, &sp_l,
56                 std::shared_ptr<const T>(
57                     std::move(r)),
58                 std::memory_order_release,
59                 std::memory_order_consume));
60   }
61 };
```

Fig. 7. The rcu_ptr class template

receives a `T*` for the copy of the actual data. We can modify the copy of the actual data inside the lambda.

III. Smart Pointer for RCU semantics

In Figure 7 we present the implementation of 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) [20], [21].

We can create an `rcu_ptr` from an lvalue 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 lvalue 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 `sp_l` (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 callable 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.

A. 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 [15], [22]. 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 [15], [22]. 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 [22]. A `memory_order_release` store is *dependency ordered before* a `memory_order_consume` load when that load returns the value stored, or in some special cases, some later value [15], [22]. Then, if the load carries a dependency to some later memory reference, any memory reference preceding the `memory_order_release` store will happen before that later memory reference [15], [22]. This means that when there is dependency ordering, `memory_order_consume` gives

the same guarantees that `memory_order_acquire` does, but at lower cost [22].

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 [11].

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: `sp_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 `[[carries_dependency]]` attribute [15]. 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 [15].

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" [22], [23]. 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 plan to use hardware specific instructions in the future to overcome the mentioned problem.

B. Lock Free atomic_shared_ptr

Our `rcu_ptr` relies on the free functions overloads with the `atomic_` prefix [15, section 20.8.2.6] for `std::shared_ptr`. It would be nice to use an `atomic_shared_ptr` [14], but currently that is still in experimental phase. We use atomic `shared_ptr` operations which are implemented in terms of a spinlock (that is how it is implemented 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 [24]. Though, it might be easier on architectures where the double word CAS operation is available as a CPU instruction as we can see that with Anthony Williams implementation [25].

IV. 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 [26] and sanitizers from the LLVM/Clang infrastructure [27] 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.

V. Future Work

It is our ongoing work to create concrete and precise performance measurements. We aim to measure the performance of our `rcu_ptr` on a weekly ordered architecture like ARMv7. Our target is to do measurements on different dimensions because the performance may depend on the architecture, the number of reader or writer threads, the ratio of the readers/writers, the size of the wrapped data, etc. Also, we plan to compare our implementation with URCU and readers-writers lock in different use cases. The complexity and the huge variegation of possible measurements drive us to publish the future results in a different paper.

VI. 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 while providing a safer and hard-to-misuse API.

ACKNOWLEDGMENT

The authors would like to thank to Péter Bolla for having valuable discussions about the implementation, and the public interface. We would like to thank also to for Máté Cserna, for his really helpful comments on the library implementation.

REFERENCES

[1] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, 1998, pp. 509–518.

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

[3] J. M. Mellor-Crummey and M. L. Scott, "Scalable reader-writer synchronization for shared-memory multiprocessors," *SIGPLAN Not.*, vol. 26, no. 7, pp. 106–113, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/109626.109637

[4] P. E. McKenney and J. Walpole, "What is RCU, fundamentally?" December 2007, available: http://lwn.net/Articles/262464/ [Viewed December 27, 2007].

[5] P. E. McKenney, *Is Parallel Programming Hard, And, If So, What Can You Do About It?* Corvallis, OR, USA: kernel.org, 2010. [Online]. Available: http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

[6] P. McKenney, "The design of preemptible read-copy-update," October 2007, available: http://lwn.net/Articles/253651/ [Viewed October 25, 2007].

[7] P. E. McKenney, D. Sarma, I. Molnar, and S. Bhattacharya, "Extending rcu for realtime and embedded workloads," in *Ottawa Linux Symposium, pages v2*, 2006, pp. 123–138.

[8] P. E. McKenney and D. Sarma, "Adapting rcu for real-time operating system usage," Oct. 23 2007, uS Patent 7,287,135.

[9] P. E. McKenney, "Sleepable RCU," October 2006, available: http://lwn.net/Articles/202847/ Revised: http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf [Viewed August 21, 2006].

[10] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole, "The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux," *IBM Systems Journal*, vol. 47, no. 2, pp. 221–236, May 2008.

[11] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 375–382, 2012.

[12] M. Desnoyers, "[RFC git tree] userspace RCU (urcu) for Linux," February 2009, http://lttng.org/urcu.

[13] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *J. Parallel Distrib. Comput.*, vol. 67, no. 12, pp. 1270–1285, 2007.

[14] H. Sutter, "Atomic smart pointers, rev. 1," ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep. n4162, Oct. 2014.

[15] ISO, *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2014.

[16] stackoverflow.com, "Understanding std::atomic::compare_exchange_weak() in c++11," 2017. [Online]. Available: https://goo.gl/jwjgGC

[17] R. K. Treiber, *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[18] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and effectively preventing the aba problem in descriptor-based lock-free designs," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*. IEEE, 2010, pp. 185–192.

[19] A. Williams, "Why do we need atomic_shared_ptr?" August 2015, available: https://www.justsoftwaresolutions.co.uk/threading/why-do-we-need-atomic_shared_ptr.html.

[20] Anthony Williams, *C++ concurrency in action: practical multithreading*. Manning Publ., 2012.

[21] stackoverflow.com, "Why are std::atomic objects not copyable?" 2017. [Online]. Available: https://goo.gl/fvuY3f

[22] P. E. McKenney, T. Riegel, J. Preshing, H. Boehm, C. Nelson, O. Giroux, and L. Crowl, "Towards implementation and use of memory_order_consume," ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep. P0098R0, 2015.

[23] H.-J. Boehm, "Temporarily deprecate memory_order_consume," ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep. P0371R0, May 2016.

[24] M. McCarty, "Implementing a lock-free atomic_shared_ptr," 2016, cppNow 2016. [Online]. Available: https://goo.gl/qErf1h

[25] A. Williams, "Implementation of a lock-free atomic_shared_ptr class template as described in n4162," 2016. [Online]. Available: https://bitbucket.org/anthonyw/atomic_shared_ptr

[26] M. Desnoyers, "Proving the correctness of nonblocking data structures," *Communications of the ACM*, vol. 56, no. 7, pp. 62–69, 2013.

[27] llvm.org. (2017) clang: a c language family frontend for llvm. [Online]. Available: http://clang.llvm.org