Read-Copy-Update As A Possible Locking Strategy In Scala

GERGELY NAGY and ZOLTÁN PORKOLÁB, Eötvös Loránd University

Concurrent programming with classical mutex/lock techniques does not scale well when reads are way more frequent than writes. In such situations the read-copy-update (RCU) locking pattern guarantees minimal overhead for read operations and allows them to occur concurrently with write operations thus may outperform classical mutexes or reader-writer locks. RCU is well-known technique among C programmers implementing performance critical low level multithreaded applications, like operating system kernels. Up to now, RCU pattern was rarely applied for high level programming languages. In this paper, we argue in favor of applying RCU for higher level object-oriented constructions and present our experimental Scala RCU class library. The library has been carefully designed to optimize performance in a heavily multithreaded environment, in the same time providing high-level abstractions, applicable in the multiparadigm environment where Scala programming language is typically utilized. We evaluated our library implementing a concurrent `HashMap` container.

1. MOTIVATION

Ever since we have started approaching the practical limits of single-threaded CPU performance, concurrent programming has been in the focus of researchers and industry parties. Concurrent programming with shared mutable state is universally considered hard [Sutter and Larus 2005]. There are many reasons for this, starting from the numerous abstractions designed to solve specific problems that don't work well together, the unpredictability of scheduling the concurrent modifications, to the complexity of reasoning about temporality of concurrent software. New approaches and concurrent programming models are being invented in hopes of reducing the complexity for software developers.

1.1 Amdahl's law

Amdahl's law [Amdahl 1967] is a formula describing the correlation bertween the theoretical speed gain of a system under a specific work load and the improvement of said system's resources. It is often used to estimate performance gains of multiple processors in parallel computing. Amdahl's law has been formulated [Rodgers 1985] as seen on figure 1, where $S_{latency}$ is the estimated speedup, s is the speedup of the part that benefits from the resource improvements and p is the original proportion of execution time without the improvements.

We also have to consider the parts of the system that don't benefit from the improvements. A good example is a software system whose task can be divided into two categories: one that does I/O opera-

Author's addresses:
Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers, Pázmány Péter sétány 1/C, Budapest, Hungary, H-1177, njeasus@caesar.elte.hu, gsd@elte.hu

This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2018: 7th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, 27–30.8.2018. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

$$S_{latency}(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

Fig. 1. Amdahl's law formulated.

tions, the other runs computations. In this case, adding multiple processor cores will not benefit the I/O parts, but will theoretically increase the performance of the concurrently ran computations.

1.2 Scalability

Amdahl's law describes the theoretical maximum improvement in a system's speed, but in typical real-world usage, even the most sophistaced approaches will have worse scaling. The reasons are various and arise from each concurrency model's fundamental properties. If we consider the actor model [Hewitt et al. 1973], one of its main characteristics is supporting messaging-based communication between actors that don't share any data. This results in copying data when actors communicate and if the actors don't share the same address space (because, for example they run on separate machines), we also have to consider network overhead.

Looking at possibly the most frequently used concurrency model using shared mutable state between concurrently executing parts of a program, we have to make sure that each thread has a consistent and correct view of the shared mutable state. For this reason, we have to *mutually exclude* modifications. To support this, we have several possible primitives, including locks, mutexes and semaphores, but using these constructs correctly is a non-trivial problem. For this reason, we have to increase the abstraction level of these for typical use-cases.

In this paper we evaluate the *read-copy-update* locking mechanism in Scala. The original implementations of RCU provide low-level APIs that require caution to avoid any concurrency issues, including data races or deadlocks. Our proposed API matches Scala's idioms by providing a higher-level abstraction while it doesn't sacrifice performance.

This paper is organized as follows: In Section 1 we discuss the motivation for expanding concurrency to answer the ever increasing need to solve problems by computers; Amdahl's Law that gives an estimation on the maximum speed gain we can expect from concurrency and the scalability problems with shared memory models. The read-copy-update locking pattern, that aims to help cases when the number of readers of a given data structure greatly outnumber the writers, is introduced in Section 2. We suggest a high-level RCU interface for Scala and describe our prototype implementation in Section 3. In Section 4 we evaluate our RCU library implementing a concurrent `HashMap` implementation comparing it to the `ConcurrentHashMap` of the Java Standard Library. Our paper concludes in Section 5.

2. READ-COPY-UPDATE

A non-trivial synronization approach has been introduced to the Linux kernel in 2002 called read-copy-update (RCU) [McKenney and Walpole 2007; McKenney 2010], although similar techniques have been previously used for garbage collection algorithms [Kung and Lehman 1980], CPU TLB implementations [Rashid et al. 1988], concurrent systems with hard real-time requirements [John 1995] and many others. RCU aims to alleviate synchronization overhead for situations when there are more read than write operations. It allows this by not blocking reads; if a write happens while there are readers, the write operation will be applied to a new memory address, keeping all readers with an intact state of the underlying data structure. When the write operation finishes, new readers will be pointed to the new memory address. After all readers of the stale data finish reading, its memory space can be freed up.

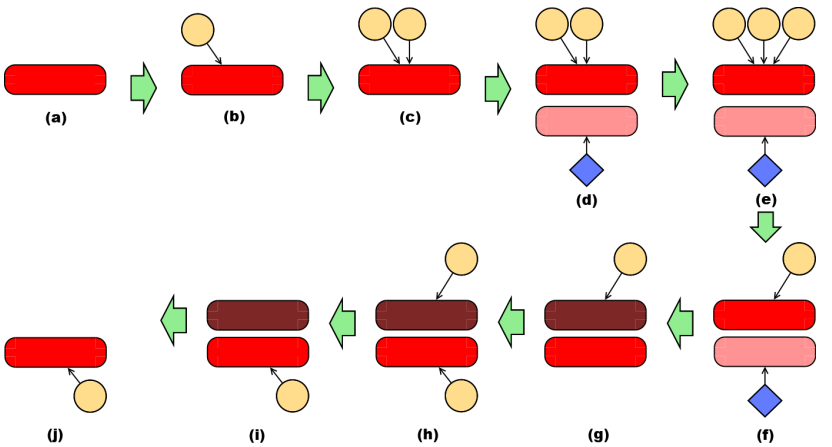

Fig. 2. RCU operations example.

On figure 2 we have displayed an example of RCU-backed concurrent reads and writes. At first we have the initial version of the data *(a)*. Later a reader *(b)* then another one start *(c)* reading the data. While they are executing their reads, a writer starts updating the data by copying the initial version *(d)* – but this doesn't block the existing readers: a third one joins *(e)*. In the meanwhile, the first two readers finish their operations *(f)*, but the third reader still refers to the old version. As the next step, the writer finishes updating the memory contents*(g)*, so when the next reader comes along, *(h)* it is pointed to the new version of the data. The last reader of the old data then completes *(i)*, meaning this memory can be discarded *(j)*.

RCU is an example of space-time trade-off where memory space is traded for performance gains: neither readers will block writers, nor a writer will block readers until it finished a possibly expensive operation. RCU does not provide conventional temporal mutual exclusion, but its handling of concurrent data access is based on spaciality: readers and writers are separated on the memory space, henceforth they can operate even while overlapping in time.

Numerous implementations can be found in various operating systems and even a user-space library is available [lib 2018]. The C-based implementation in the Linux kernel provides a fairly low-level API that can be mapped to the following operations or methods:

—`rcu_read_lock()`. Marking the beginning of a read operation so we know when this data can be freed up if it has been update by a write

—`rcu_read_unlock()`. Signaling that a read operation has been finished

—`synchronize_rcu()`. Blocks until all *pre-existing* read operations finish. In some implementations this method can be called with a callback that will be applied when the reads have finished instead of blocking

—`rcu_assign_pointer()`. Writers use this method to update the underlying data managed by RCU

—`rcu_dereference()`. Readers can access the RCU-guarded data via this method

Based on the short description of this API, it is easy to see that using a read-copy-update implementation is a non-trivial task and needs special attention from its users. To help developers avoid typical pitfalls, an implementation with a higher-level API has been completed in C++ that uses some of the new features of C++14 [Márton et al. 2017].

```scala
trait ReadCopyUpdate[A] {
  def modify(modifier: A => A): A
  def get(): A
  def map[B](f: A => B): B
  def set(value: A): Unit
}
```

Fig. 3. RCU Scala API

```scala
trait ReadCopyUpdateNaive[A] extends ReadCopyUpdate[A] {
  protected val data: AtomicReference[A]
  private val modifierLock = new AnyRef

  def modify(modifier: A => A): A =  {
    val newData = modifierLock.synchronized {
      modifier(data.get())
    }
    data.set(newData)
    newData
  }

  def get(): A = {
    data.get()
  }

  def map[B](f: A => B) = f(get())

  def set(value: A): Unit = {
    data.set(value)
  }
}
```

Fig. 4. RCU using basic locking logic

3. POSSIBLE READ-COPY-UPDATE IMPLEMENTATIONS IN SCALA

As Scala provides high levels of abstraction, one of the goals of our read-copy-update implementation was to hide the low-level details from the users. This would require us to handle all logic related to the internal locking mechanisms of RCU, but in the meanwhile support all use-cases without performance penalties. We have also been trying to provide an idiomatic Scala API that borrows patterns from similar holder objects in the standard library, such as `Option[+A]` [opt 2018]. This would allow users to easily understand a familiar API and would prevent them from implementing concurrency-related bugs.

3.1 The RCU API

Here we list the API of our generic read-copy-update trait: The methods listed here cover the operations in other RCU implementations, but hide the low-level locking mechanisms needed for correctness. The `get()` and `set()` methods cover the basic reference operations of getting and setting values of the guarded memory location. To update the RCU data structure, users need to provide a modifier function that receives the current state as its input and produces the new data. This passed-in function or lambda represents the actual *write* operation and the `modify()` method handles all necessary logic internally. `map()` is a simple convenience function that acts as a reader and transforms the data.

3.2 The naïve implementation

The naïve version uses an `AtomicReference` for the backing data since it provides lock-free reference updates. Writes themselves need to be synchronized amongst each other since a new write operation can be requested while another one is still working on modifying the underlying data. To prevent data collisions or overwriting using an older version at a later time, the complete write operations need to have a mutual exclusion. Setting the reference can happen outside of the write synchronization as the atomicity is guaranteed by `AtomicReference`.

```scala
trait ReadCopyUpdateRWL[A] extends ReadCopyUpdate[A] {
  protected var data: A
  private val l = new ReentrantReadWriteLock(false)
  private val rl = l.readLock()
  private val wl = l.writeLock()
  private val modifierLock = new AnyRef
  def modify(modifier: A => A): A = {
    modifierLock.synchronized {
      val newValue = modifier(data)
      try {
        wl.lock()
        data = newValue
        newValue
      } finally {
        wl.unlock()
      }
    }
  }

  def get(): A = {
    try {
      rl.lock()
      data
    } finally {
      rl.unlock()
    }
  }

  def map[B](f: A => B) = f(get())

  def set(value: A): Unit = {
    wl.lock()
    data = value
    wl.unlock()
  }
}
```

Fig. 5. RCU using read-write locks

3.3 Using read-write locks

We have also implemented RCU using a more advanced, ReadWriteLock-based locking scheme. The only difference compared to the naïve version is that we use two different locks for reading the reference and setting it. Theoretically this should favor readers even more.

4. EVALUATION

To evaluate our read-copy-update class, we have implemented one the most fundamental data structures used in software, the `HashMap` in a thread-safe way. Both the Java and Scala standard libraries provide a version of it, helping us compare both the ease-of-use and the performance of the proposed solution. In this section we will analyze the implementations of the two basic operations of a `HashMap` – `get()` and `+=` – between the standard version and our RCU-based implementation. We will also discuss the performance characteristics of these two different versions.

4.1 Concurrent `HashMap` in Scala

Scala's concurrent `HashMap` is a wrapper around the Java `java.util.concurrent.ConcurrentHashMap<K, V>` class, providing a Scala-like interface. Converting the Java implementation is fairly simple based on the decorators found in `scala.collection.convert.decorateAsScala`: one can simply call `new ConcurrentHashMap[K, V]().asScala` to get a `scala.collection.concurrent.Map` object.

`ConcurrentHashMap` [con 2018] provides a lock-free retrieval operation while writes never lock the whole internal table. This is achieved by introducing segments to the entry array, where each segment represents a range of the possible hash values. When write operations happen simultaneously, in ideal cases they will only touch different segments, avoiding lock contention. This of course might not be the case in real-world applications. The number of segments –or as the class documentation refers to it, the level of supported concurrency– can be controlled from client code, providing the possibility of fine-tuning performance characteristics of the class.

```
protected val table: ReadCopyUpdate[Array[Entry[K, V]]]

override def +=(kv: (K, V)): RcuHashMap.this.type = {
  val (key, value) = kv
  table.modify{ t =>
    val index = indexFor(hash(key.hashCode()), t.length)
    val entry = t(index)
    if(entry != null) {
      entry.addNext(key, value)
      noOfElements += 1
      t
    } else {
      val res = addEntryToTable(t, key, value, index)
      noOfElements += 1
      res
    }
  }
  this
}

private def addEntryToTable(t: Array[Entry[K, V]], key: K, value: V, index: Int): Array[Entry
      [K, V]] = {
  val tableLength = t.length
  val array = if((tableLength * loadFactor) <= noOfElements) {
    val newArray = new Array[Entry[K, V]](t.length * 2)
    Array.copy(t, 0, newArray, 0, tableLength)
    newArray
  } else { t }
  array(index) = new Entry[K, V](key, value)
  array
}
```

Fig. 6. RCUHashMap += implementation

```
override def get(key: K): Option[V] = {
  val entry = table.map{t => t(indexFor(hash(key.hashCode()),t.length))}
  if(entry != null) entry.get(key)
  else None
}
```

Fig. 7. RCUHashMap `get()` implementation

4.2 RCU-based Concurrent `HashMap` in Scala

Our `concurrent.Map` is based on the non-thread-safe Java `HashMap` class [map 2018]. Most of the considerations put into the implementation details of that class are copied over to our version, including bucket implementation and hash-distribution. The main difference is that our hash table is now a `ReadCopyUpdate` instance, and both `HashMap` reads and writes will be delegated through the RCU API. We have listed the code snippets for both of these methods for reference.

Comparing these methods to the original ones reveals how simply one can convert single-threaded code to use RCU. Furthermore, since all locking-related logic is handled in the `ReadCopyUpdate` class, the readability of the client code doesn't decrease significantly, allowing developers to focus on the domain problem.

4.3 Benchmarks

We have used JMH [jmh 2018] as our benchmark harness to reduce possible variability to the minimum. The test code concurrently reads and writes to the map object using multiple threads. Since the goal was to benchmark RCU performance, we were focusing on cases where there is minimal hash collision, resulting in a flat bucket structure. This allows us to test the characteristics of the RCU data structure.

The test machine had 80 cores of Intel Xeon E5-2698 v4 2.20 GHz with 512 gigabytes of memory. The benchmarked method creates 50000 `Future` instances, of which a given number will be writers, the rest of the operations will be readers and it will block until all `Futures` complete.

4.4 Results

On figures 8 and 9 we show how Java's `ConcurrentHashMap` implementation compares to the RCU-based one. We have run these benchmarks with a fixed number of threads and changed the ratio of read to write operations. The Y-axis shows values of ms/op, while the X-axes show that every nth operation is a write.

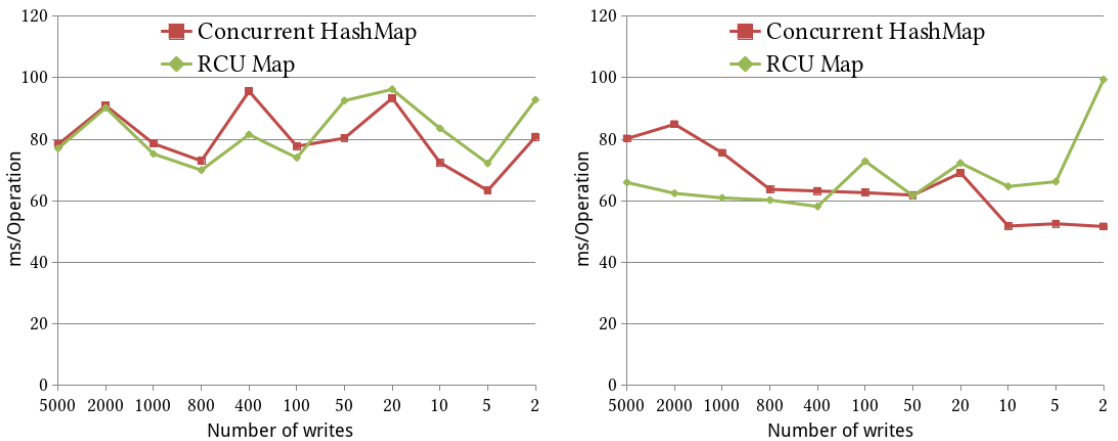

Fig. 8. Benchmark results using 64 (left) and 32 (right) threads.

As the charts show, the performance of our implementation is comparable to `ConcurrentHashMap`'s. RCU proves to be usually faster when read operations dominate, and it usually keeps up pace even when write operations become more frequent.

We have also tested how well RCU scales if we increase the worker threads. These results can be found on figure 9. Here, on the X-axis we displayed the number of threads with every 8000th operation being a write.

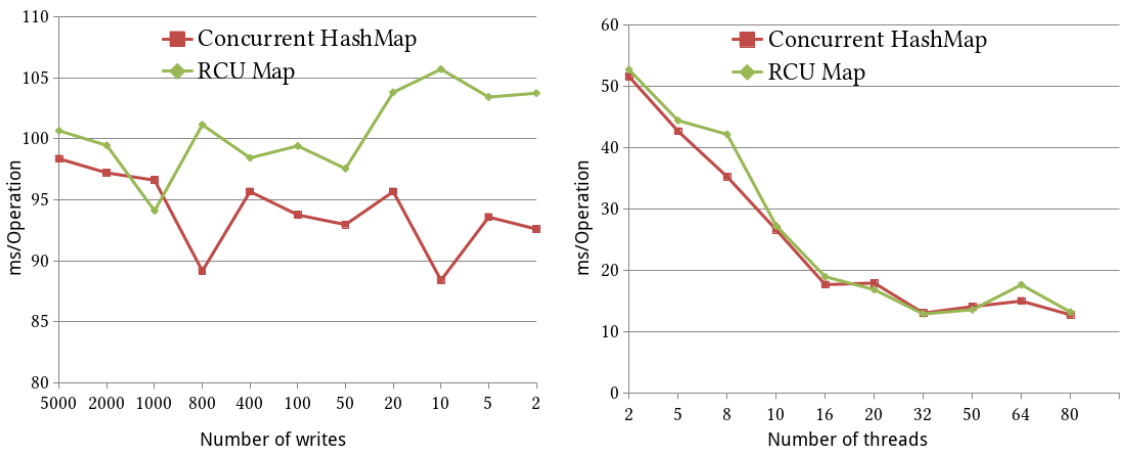

Fig. 9. Benchmark results using 4 threads (left) and scaling with 8000 writers (right).

There is no realistic way of eliminating all variance of real-world benchmarks; as our data show, we have observed some discrepancies in our measured values, even though we used JMH's warm-up capabilities as well as calculated average results over 15 iterations of the benchmarked method.

5. CONCLUSION

Concurrent programming with classical mutex/lock techniques does not scale well when reads are more frequent than writes. For such situations programs working on low abstraction level successfully apply the read-copy-update locking pattern. As the original implementations of RCU provide relatively low level API, there is a growing need for creating a high level solution. In this paper we presented our RCU library for the Scala programming language providing a higher-level abstraction while doesn't sacrifice performance.

We evaluated our library creating a concurrent `HashMap` that uses our RCU implementation to handle concurrency on its internal hash table. We compared its performance to Java's standard `ConcurrentHashMap` and found that it is comparable, in some cases the RCU-based solution even proves to be faster.

As future development, we can widen the number of investigated data structures, trying to find typical use-cases where RCU can benefit its users. We can also improve the RCU `HashMap` implementation to leverage further benefits provided by the read-copy-update locking mechanism.

REFERENCES

2018. OpenJDK ConcurrentHashMap implementation file. (2018). https://github.com/dmlloyd/openjdk/blob/jdk/jdk/src/java.base/share/classes/java/util/concurrent/ConcurrentHashMap.java

2018. OpenJDK HashMap implementation file. (2018). https://github.com/dmlloyd/openjdk/blob/jdk/jdk/src/java.base/share/classes/java/util/HashMap.java

2018. OpenJDK JMH: A Java harness for building, running, and analysing nano/micro/milli/macro benchmarks. (2018). http://openjdk.java.net/projects/code-tools/jmh/

2018. Scala Option API documentation. (2018). https://www.scala-lang.org/api/2.12.2/scala/Option.html

2018. Userspace RCU: A Linux-based userspace RCU (read-copy-update) library. (2018). http://liburcu.org

Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS 67 (Spring)*. ACM Press. DOI:http://dx.doi.org/10.1145/1465482.1465560

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. Session 8 Formalisms for Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Advance Papers of the Conference*, Vol. 3. Stanford Research Institute, 235.

Aju John. 1995. Dynamic Vnodes-Design and Implementation. *USENIX* (1995), 11–23.

H. T. Kung and Philip L. Lehman. 1980. Concurrent Manipulation of Binary Search Trees. *ACM Trans. Database Syst.* 5, 3 (Sept. 1980), 354–382. DOI:http://dx.doi.org/10.1145/320613.320619

G. Márton, I. Szekeres, and Z. Porkoláb. 2017. High Level C++ Implementation of the Read-Copy-Update Pattern. *INFORMATICS 2017: 2017 IEEE 14th International Scientific Conference on Informatics Proceedings.* (2017), 243–348.

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

Paul E. McKenney and Jonathan Walpole. 2007. What is RCU, Fundamentally? (17 December 2007). Available: http://lwn.net/Articles/262464/ [Viewed December 27, 2007].

Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. 1988. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Trans. Comput.* 37, 8 (1988), 896–908.

David P. Rodgers. 1985. Improvements in multiprocessor system design. *ACM SIGARCH Computer Architecture News* 13, 3 (jun 1985), 225–231. DOI:http://dx.doi.org/10.1145/327070.327215

Herb Sutter and James Larus. 2005. Software and the Concurrency Revolution. *Queue* 3, 7 (Sept. 2005), 54–62. DOI:http://dx.doi.org/10.1145/1095408.1095421