Expertise
and insight
for the future

Dan Suman

Design and implementation of a concurrent RingBuffer in Scala

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

6 January 2019

Author Title Number of Pages Date	Dan Suman Design and implementation of a concurrent RingBuffer in Scala 37 pages 06 January 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Peter Hjort, Senior Lecturer

Queues are a significant algorithmic component to many systems and applications that enable decoupling of producers and consumers. Some of its real-life applications are ticketing (as a waiting list), a queue of packets in data communication or a queue of operating system processes. With the rapid growth of parallel processing and event-driven applications, concurrent implementations of this data structure have gained significant importance. Despite the relevance, many available implementations in JVM ecosystem employ expensive operations such as thread blocking (BlockingQueue) or true unboundedness (ConcurrentLinkedQueue), a characteristic which discards the usage of the data structure where consumer backpressure is needed. This thesis is an attempt to address such concerns, by design and implement a lock-free concurrent bounded queue, or RingBuffer, in Scala.

The thesis focuses first on exploring the concurrency mechanisms that describe the JVM ecosystem. Then it proceeds to the lock-free RingBuffer implementations, employing different concurrency primitives and two different underlying data structures (Scala's immutable Queue and Java's mutable Array). The final section is dedicated to benchmarks comparing RingBuffer implementations and Java's concurrent queue implementations. Measurements done for this section showed a significant improvement in speed over the available data structure although the API is presenting a few key differences.

Keywords	RingBuffer, CAS, synchronized, thread-safe, Queue, Scala

Contents

List of Abbreviations

CAS	Compare-And-Swap represents a concurrency primitive
CPU	Central Processing Unit is the computer's main processor
CSP	Communicating Sequential Processes
FAA	Fetch-And-Add represents a concurrency primitive
FIFO	First-In-First-Out. The terminology used to describe the functionality of a queue
JLS	Java Language Specification
JMM	Java Memory Model
JVM	Java Virtual Machine
MESI	Modified/ exclusive/ shared/ invalid is a cache coherence protocol
MOESI	Modified/ owned/ exclusive/ shared/ invalid is a cache coherence protocol
TAS	Test-And-Set represents a concurrency primitive

1 Introduction

With the exponential growth of computing power and the consequent work distribution across several CPU threads, logical or physical, concurrency and parallelism have become topics of vital importance when developing new applications. Concurrency patterns, such as Golang's goroutines based on the theory of Communicating Sequential Processes proposed by Tony Hoare in 1978 (Hoare, 1978) or the actor model proposed by Carl Hewitt in 1973 (Hewitt, Bishop and Steiger, 1973), aim to enable parallelism by proposing different alternatives with regards to task distribution across multiple threads. The main idea in the CSP solution is that two processes acting independently can share a channel where one process adds data into, and one consumes. The actor model is based on the actor entities passing asynchronously data, stored in each other's mailbox. A common characteristic of both approaches is that both the channel in the CSP case and the actor mailbox work as a queue of tasks. One focus for this thesis was the designing of the data structure mentioned above in terms of features and behavior: data was placed into the queue by multiple processes and was read by multiple processes as well.

The above-described behavior is known in computer science as the producer-consumer problem (Herlihy and Shavit, 2012), or the bounded buffer problem. The buffer stores the data put by the producer processes until the consumers remove it. If the buffer is empty, the consumers will pause. Vice versa, if the buffer is full, the producers will pause. Considered that the bounded buffer, referred to as RingBuffer throughout this thesis, is a shared resource, all involved processes must cooperate to ensure the data integrity. In this thesis, the data structure's exposed API was lock-free. This implies that the producer and consumer process would have to race for the shared resource, but the result of the operation in case the RingBuffer was full or empty would be returned immediately, without blocking any thread. The implementation, however, required careful considerations to ensure data integrity. Scala and Java have multiple approaches to solve the concurrency problem in a multithreaded environment, such as using low-level concurrency primitives that manage access and modifications directly to memory addresses, as well as high-level synchronization blocks. Chapter 2 of the thesis describes and compares the different considerations to be done regarding synchronization and locking.

The RingBuffer was implemented as a queue. A queue represents a linear collection of objects that are inserted and removed according to the first-in-first-out principle. An excellent example of a queue is a line of students at the cafeteria. New additions to a line are made to the back of the queue, while removal (or serving) happens at the front. The underlying structure for a queue could be an Array, a Vector, an ArrayList, a LinkedList, or any other collection. Similarly, the RingBuffer was implemented using the different data structure to analyze the performance gain or loss.

The main goal for this thesis was the implementation of a concurrent RingBuffer, a bounded queue implementation in Scala. The underlying hypothesis behind this focus is that it would indeed be possible to design a more performant data structure in the JVM landscape (Scala in this research) than the available Java concurrent queue implementations. The main critique for the current solutions is that the majority of implementations are either blocking the producer/consumer thread (LinkedBlockingQueue, ArrayBlockingQueue), a quite expensive operation in concurrent programming or provide true unboundedness (ConcurrentLinkedQueue), which might imply a penalty cost if the needed implementation requires boundness. Therefore, the secondary goal was to explore the concurrent aspect of the language and different techniques available in the JVM landscape and to apply them where it was possible to boost the different RingBuffer's implementations' performance. Finally, different underlying data structures were tested for the RingBuffer implementation for performance reasons. As a conclusion, a benchmark against the more performant and previously mentioned data structures was done to validate the base assumption that performance could be improved while maintaining reliability under constant high concurrent load.

2 Concurrency in Java

2.1 Thread safety

The most popular form of structured programming is imperative programming based on the concept of sequential execution and a mutable state. It is directly derived from the Von Neumann architectural conceptions. Threads are often seen because of this notion, which allows multiple control flows at the same time. In addition to more heavy processes, threads are the main parallel structures provided by operating systems and hardware architectures. (Erb, 2012) Threads represent the main mechanism for concurrency in most programming languages. However, the programming model based on threads shared state and locks poses a series of challenges, which will be described in the following paragraphs.

In concept, a thread defines a sequential control flow, which is isolated at first glance from other events. In contrast to processes, however, threads have the same address space. This means that several separate threads can have access to the same arguments and states simultaneously. What is considered even worse is that sequential programming is based on the concept of a mutable state, meaning that multiple threads can also compete in data writing. Multi-threading is mainly used with pre-emptive scheduling, with the exact switches and interconnections between threads not known beforehand. This constitutes a strong indeterminacy and may result in verification of race conditions. Such a case can be verified when multiple threads compete for the same resource, whose state is shared across threads. Due to race condition the resource state may result incoherent. For example, a thread can read the outdated state while another thread proceeds with updating it. If several threads modify the state at the same time, with only a subset of the changes being persisted, the outcome will be inconsistent. In conclusion, mechanisms to sentinel and impose synchronized access to resources are needed. (Erb, 2012)

Consider the following simple example: Student A has a bank account holding a total amount of 20 EUR. Student A authorizes both students B and C to request and withdraw 20 EUR each from his bank account. Students B and C concurrently try to withdraw the money, first by checking the current balance, 20 EUR and consequently setting the new balance of 0 EUR. Since both students B and C saw the initial balance of 20 EUR, both were authorized the operation. To put this into perspective, students B and C are two

threads modifying a stateful object concurrently, causing a race condition resulting in an unexpected and incorrect state in our application.

An implementation can be considered thread safe if, when called from different threads and without considering the different execution contexts at runtime environment, it can be accessed and modified correctly without any additional locking or synchronization mechanism needed. Such classes shall implement any additional synchronization to ensure thread safety on the client side.

2.2 JMM

In the Java Memory Model objects exist in shared memory, with each thread being in possession of a cached copy of read and written fields in its private memory. Without explicit synchronization across threads, a resource state modification from a single thread might not be immediately propagated to the shared memory. Similarly, the cached copy of the resource might not be updated for a thread if the resource was updated in the main memory. Java Virtual Machine often keeps the cached copies in sync, even if not constrained to do so. Synchronization events (described in chapter 2.3) imply some form of atomicity or mutual exclusion. In Java, it also means reconciling the cached memory of a thread with the shared memory. Some synchronization events result in the thread updating the cached changes to shared memory, broadcasting the changes to the other threads. Other synchronization events will result in the thread invaliding its private cached memory, forcing a full update from the shared memory before modifying the resource. (Herlihy and Shavit, 2012)

2.2.1 Cache coherence

If a system has many processors with each processor having their private cached memory, if the main memory data is shared with processor's independent memory, a high degree of inconsistency might be verified if there are changes in the shared data. However, if the processors read only from the same memory address, there is no inconsistency problem. As an example, if a field is modified by a processor in the main memory, the field needs to be invalidated in all the other processors as well to ensure that the modified field is not read by other processors. This problem is called cache

coherence. There are multiple cache coherence protocols: MSI, MESI, MOSI, MOESI, MERSI. This section will focus, however, on two main protocols implemented in modern hardware: MESI (Modified-Exclusive-Shared-Invalid) and MOESI (Modified-Owned-Exclusive-Shared-Invalid). (Dey and S. Nair, 2014) In the MESI protocol the cache lines are described by a state that reflects certain characteristics, which are the following:

- Modified: The field in the main memory has been modified, the current state of the line is dirty
- Exclusive: The field is present in one cache only, the current state of the line is clean
- Shared: The field is present in multiple caches, the current state of the line is clean
- Invalid: The field is invalid, and usage is prohibited

The MOESI protocol has an extra state, namely the Owned state. The cache line is available for all the other caches. However, only the current cache is entitled to make changes. Would any changes occur to the cache line, the cache state is set to Owned before broadcasting the modified cached field with the remaining caches. An additional advantage of this is that fields labeled as dirty can be shared with other caches without updating the main memory value. (Dey and S. Nair, 2014)

2.2.2 Data-Race Free Guarantee

The sequential consistency defined by Lamport (1979) provides an ample guarantee., by specifying two core properties for any execution result:

- Execution is consistent from a single thread perspective
- Execution trace is ordered and unique from all thread's perspective

Sequential consistency is particularly strong as it requires all memory operations to be propagated instantly and visible to all threads atomically: this has a strong impact on what optimizations can be made and how efficient memory operations are, as it effectively implies that all threads are synchronized globally. Since each thread is

mapped to a processor on the host, this requirement automatically propagates at the hardware level. (Cs.umd.edu, 2004)

The need for sequential consistency avoided expensive optimizations that the compiler and the hardware could make and seriously impeded the performance of any but the most trivial programmes. The system must be able to reorder the instructions provided by the program to minimize memory access latency.

The Java Memory Model offers a weaker guarantee called Data-Race Free Guarantee. If all sequentially consistent executions of a program do not imply data races, all executions will be sequentially consistent. This guarantee is defined by the happens-before relationship: if two conflicting accesses (at least one access is writing at the same memory location) are not ordered from the happens-before perspective, it is a data race. A program with all sequentially consistent executions without data races should be synchronized correctly. (Cs.umd.edu, 2004)

2.3 Locking

Java standard library offers a locking mechanism to ensure atomic operations, namely the synchronized block. A synchronized block consists of two parts: an object reference acting as the lock and a block of code to be executed safe-guarded by the lock.

```
synchronized (lock) {

// Access or modify shared state guarded by a lock

}
```

Listing 1. A synchronized code block in Java.

According to Goetz (2006), “Every Java object can implicitly act as a lock for purposes of synchronization; these built-in locks are called intrinsic locks or monitor locks. The lock is automatically acquired by the executing thread before entering a synchronized block and automatically released when control exits the synchronized block, whether by the normal control path or by throwing an exception out of the block. The only way to acquire an intrinsic lock is to enter a synchronized block or method guarded by that lock”.

In Java, intrinsic locks behave as mutual exclusion locks, which means that only a thread can possess the lock at a time. When the thread A tries to acquire a thread B lock, A must wait or block until it is released by thread B. A waits forever if B never releases the lock.

If a thread is holding a lock, by entering a new synchronized block coordinated by the same lock, the next block will automatically be executed. This property is known as reentrancy. Java performs this operation by combining the lock acquisition count and thread ownership with each lock. The thread owning the lock is recorded and the count is incremented by one and the given thread acquires the lock for the first time. In the circumstance when the lock is to acquire the same lock again, the count will be incremented. Similarly, if the thread is to exit the synchronized block, the count is decremented and the lock is released. The lock's owning thread details are used for determining if the thread requesting the locks owns it already. In java.util.concurrent.locks, there are multiple locking implementations with the same basic reentrant behavior and semantics as the synchronization implicit monitor lock but with extended capabilities and control over locking semantics and timing. (Goetz., 2006)

Synchronization offers two features: visibility and mutual exclusion. Mutual exclusion ensures that only a single thread can have access at a time when several threads compete for a single resource, as mentioned before. Visibility also provides synchronization. The guarantee that the resource updated by one thread will be visible to other threads cannot be provided in the absence of synchronization (or volatility). As Goetz (2006) mentions "… even see stale values in case of 64-bit primitives like long and double as the value may be written as two 32-bit values. By synchronizing the write and read to a variable, the latest value written to a variable by one thread will be visible to all other threads, and there will not be any stale value". In addition to mutual exclusion, synchronization also provides memory visibility. If only visibility is required, and not mutual exclusions, the resource can be declared as volatile as it provides the identical advantages as synchronization does without the mutual exclusion, with a performance benefit over synchronization.

Mutual exclusion lock ensures that only a single thread can enter a critical section at a time. If a new thread competes for the same resource sentineled by the lock, the execution is blocked until the thread is notified to try again. A semaphore is a generalization of blockages of mutual exclusion. Every Semaphore has an initial capacity

provided for the constructor. The semaphore acts as a sentinel for critical sections, instead of allowing a single thread at a time, it allows as many as the value of the capacity provided during initialization. The Semaphore class has two methods: a thread calls acquire() to request authorization to enter the critical block, and release() to notify that the execution has finished. The Semaphore itself is just a counter: it tracks the number of threads allowed to enter. If a new acquire() call exceeds the capacity, the calling thread will be suspended until the room is available. (Herlihy and Shavit, 2012)

2.4 Fairness

The built-in waiting and notification methods in JLS offer no guarantees of fairness. There is no guarantee for which thread in a waiting set will be notified or which one will be able to first to acquire the lock and notify the remaining threads of the operation. (Herlihy and Shavit, 2012)

This flexibility in the JVM implementations allowed by the JLS makes it impossible to infer the exact thread execution process. However, in most contexts, this is not a practical problem. Considered for example a concurrent buffer application, it is irrelevant to point the exact thread from a set will execute the specific code block.

However, it is sensible in the management of a resource pool to ensure that the threads waiting for necessary resources are not constantly removed by others because they are unfair in their choice of which threads are to be blocked in the underlying notification process. Many synchronous channel applications have similar concerns. (Kwiatkowska, 1989)

Guaranteeing that the host system will be executing a specific thread or process exceeds the JLS minimum requirements. From a pragmatical standpoint, this is unlikely to be an important issue. Most JVM implementations aim to offer sensible planning policies that go far beyond the JLS set of minimum requirements. They show weak, limited or probabilistic fairness regarding the execution of running threads. For a language specification it is difficult to cover all the possibilities; hence, at least regarding JLS, this matter is left as a problem of implementation.

2.5 Concurrency primitives

Processors perform synchronization operations to guarantee data integrity and consistency in the shared memory with the support of hardware implemented primitives. Synchronization overhead (such as atomic update) represents one of the problems to be tackled when designing and planning for performance scalability in the context of shared memory multiprocessors.
Several atomic primitives co-exist on the hardware level to provide atomic modifications to different memory addresses we distinguish mainly three concurrency primitive functions, namely Test-And-Set, Fetch-And-Add, Compare-And-Swap.

```
function TAS(value_pointer:pointer to word, value:word):word
       atomic do
            old_value=*value_pointer;
            *value_pointer=value;
            return old_value;
```

Listing 2. Test-And-Set pseudocode.

As illustrated in Listing 2 the primitive takes effectively two parameters, the destination pointer, and the value to be set. It returns the old value pointed to by value_pointer and updates the old value to the new at the same time. The key is that the operation sequence is carried out atomically. The reason it is called " test and set " is that it allows " testing " the old value (which is the returned value) while " setting " the memory location to a new value simultaneously.

```
procedure FAA(address:pointer to number, number:integer): integer
       atomic do
            old_value=*value_pointer;
            *value_pointer=value + number;
            return old_value;
```

Listing 3. Fetch-And-Add pseudocode.

In listing 3, we have the pseudocode of the Fetch-And-Add primitive instruction, which atomically increases value by returning the old value at a certain address.

```
function CAS(address:pointer to word, oldvalue:word, newvalue:word):boolean
       atomic do
              if *address = oldvalue then *address := newvalue; return true;
```

```
        else return false;
```

Listing 4. Compare-And-Swap pseudocode.

Another primitive hardware that some systems provide is the comparison and swap instruction or the comparison and exchange instruction. For this single instruction, the pseudocode is found in Listing 4. Compare-And-Swap atomically compares a memory location's content with a certain value, in case they are the same the memory location's content gets set to a new value and the instruction returns true, false otherwise.

2.6 Java concurrent queue implementations

A concurrent queue is generally a thread-safe implementation which supports multiple producers to enqueue items, and multiple consumers to dequeue them. Thread safety implies that the state of the queue is synchronized across its clients. To better describe the pattern, let's go back to the cafeteria example. Student A enqueues in the line, and both Cashier A and Cashier B try to serve Student A. Cashier A, and Cashier B do not talk to each other, so they possibly end up charging Student A twice, and that is something to be avoided.

A blocking queue is a queue that blocks when you try to dequeue from it, and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

Java Concurrent Linked Queue represents an unbounded thread-safe queue based on linked nodes (ConcurrentLinkedQueue, 2018). The implementation achieves the concurrency without using a locking mechanism, by using the Compare-And-Swap concurrency primitive. This separates the producer queue operations from the consumer ones, effectively being up to scale the threads without blocking concerns. The implementation is based on a LinkedList data structure. Hence, size and random access are operations that require linear time, while enqueuing and dequeuing operations are done in constant time.

Java Linked Blocking Queue represents an optionally-bounded blocking queue based on linked nodes (LinkedBlockingQueue, 2018). The queue will block either the producer or the consumer when the queue is full or empty, putting the working threads to sleep until a value is produced. However, the blocking feature is potentially expensive in scenarios of multiple producers/consumers. Every queue and dequeue operation is lock contended.

Java Array Blocking Queue represents a bounded blocking queue with an underlying Array as the primary data structure (ArrayBlockingQueue, 2018). The array has a fixed size and operates as a “rounded buffer.” Elements are enqueued and dequeued, the head and the queue tail being determined by two cursors, based on which the queue can resolve its capacity. Attempting to deque an element from an empty queue will result in a blocking operation, likewise enqueue operation will block on the full queue. A substantial difference from the LinkedList implementation is that all operations on this data structure are constant time.

3 Immutable queue-based RingBuffer

A RingBuffer also referred to as a circular buffer or circular queue, is a circular data structure, although the implementation is linear. The RingBuffer is a common queue implementation. While a RingBuffer is represented as a circle, in the underlying code, a RingBuffer is linear. A RingBuffer exists as a fixed-length collection of objects with two pointers: the first representing the head of the queue, and the second representing the tail. In a queue, elements are added to the queue tail in a FIFO type of ordering. In the order they have been added, the first elements from the top of the queue are removed.

3.2 Specifications

Before deep diving into implementation details of the RingBuffer implementation, we ought to define a set of specifications that will eventually determine the success or failure of the implementation. This set of the ground will have to be rigidly followed and will represent a base case for the testing and benchmarking of the different implementations:

1. The data structure can contain elements of different type
2. The RingBuffer initial size has to be greater than zero
3. The RingBuffer has a fixed limit set upon creation
4. On enqueue operation, an element is added at the end of the list and size is increased
5. On dequeue operation, an element from the queue head is taken, and size is decreased
6. If the queue has reached its limit, the enqueue operation will return false
7. If the RingBuffer is empty, a dequeue operation will return a None value

3.3 Interface

The first requirement to be implemented is the type generalization. Hence the RingBuffer will be implemented using a type parameter. Provided that the implementation of this data structure will be immutable, the type parameter has to be covariant, specified in Scala with the + annotation. Covariance regarding an A type parameter permits the flexibility to pass a queue of subtyped elements to functions allowing [A] type.

(Docs.scala-lang.org, 2019) As an example, a RingBuffer[Apple] could be passed as an argument to functions requiring a RingBuffer[Fruit] (Listing 5).

```
class RingBuffer[+A](val size: Int, val capacity: Int)
```

Listing 5. RingBuffer class definition

The next step would be the definition of the enqueue method. At first glance, a simple operation would suffice (Listing 6).

```
def enqueue(a: A): RingBuffer [A]
```

Listing 6. Simple enqueue method

However, this leads to a compiler error since the input is in a contravariant position. In other words, the compiler will not be able to completely infer the type of return Ring-Buffer. To address this issue, we need to define a new type parameter constraint that will be a supertype of A. The interface will now compile. However, as mentioned in the specifications, the enqueue method will effectively have to return false if the RingBuffer is full. Hence, we ought to change our return signature including the Boolean value.

```
def enqueue[B >: A](element: B): (Boolean, RingBuffer[B]) = ???
```

Listing 7. RingBuffer enqueue method

To complete our interface definition, we need to define the dequeue operation (Listing 8).

```
def dequeue: Option[(A, RingBuffer[A])] = ???
```

Listing 8 RingBuffer dequeue method

3.4 Naïve implementation

RingBuffer, as the Queue data structure, represents only an interface per se, and an underlying data structure is needed for the implementation. The most straightforward data structure that would indeed provide the immutability and the FIFO (first in first out) ordering is the Scala's standard library immutable queue. Its size method runtime is

indeed O(N), however, in the interface definition, we define size as a class parameter (Listing 9), which indeed will reduce our size runtime to O(1). Size runtime is important for our enqueue method implementation considered that RingBuffer is a closed data structure and therefore size needs to be constantly checked for every enqueue operation.

```
case class RingBuffer[+A](val size: Int, val capacity: Int, queue: Queue[A])
```

Listing 9 RingBuffer implementation class definition

```
def enqueue[B >: A](element: B): (Boolean, RingBuffer[B]) = {
            if (size < capacity) true -> RingBuffer(capacity, size + 1,
queue.enqueue(element))
            else false -> RingBuffer(capacity, size, queue)

}
```

Listing 10 RingBuffer enqueue implementation

In the enqueue implementation (Listing 10) the most significant check regards the size against the capacity of the RingBuffer. If it allows for new elements, it returns a Boolean flag signaling the success of the operation and the new immutable RingBuffer, a false and the old reference otherwise

```
def dequeue: Option[(A, RingBuffer[A])] = {
            queue.dequeueOption.map {
    case (el, newQueue) => el -> RingBuffer (capacity, size - 1, newQueue)
  }
}
```

Listing 9 RingBuffer dequeue implementation

The pure implementation of our dequeue method is fairly simpler both in reasoning and actual implementation. Considered that successful implementation of all the required methods as previously described we can now summarize by providing a full snippet of the code in addition to the class constructor.

```
import scala.collection.immutable.Queue
```

```
case class RingBuffer[+A] private (val size: Int, val capacity: Int, queue:
Queue[A]) {
def enqueue[B >: A](element: B): (Boolean, RingBuffer[B]) = {
            if (size < capacity) true -> RingBuffer(capacity, size + 1,
queue.enqueue(element))
            else false -> RingBuffer(capacity, size, queue)

}

def dequeue: Option[(A, RingBuffer[A])] = {
queue.dequeueOption.map {
    case (el, newQueue) => el -> RingBuffer (capacity, size - 1, newQueue)
                        }
}
}

object RingBuffer {
def empty[A](capacity: Int): RingBuffer [A] = RingBuffer (capacity, 0,
Queue.empty[A])

def apply[A](capacity: Int)(els: A*): RingBuffer [A] = {
    val elements = if (xs.size <= capacity) els else els.takeRight(capacity)
    Ring(capacity, elements.size, Queue(elements: _*))
  }
}
```

Listing 10 RingBuffer full naïve implementation

From a pure performance analysis, the RingBuffer implementation is quite efficient. The chosen data structure for the implementation is the Scala standard library immutable queue which has a set of significant advantages:

- Enqueue and dequeue values in a first-in-first-out (FIFO) manner
- Provides immutability which simplifies concurrency issues
- The queue in its scala immutable implementation consists as a pair of Lists, one holding the "in" elements and the second the "out" elements.
- Both the enqueuing and the dequeuing operations have always cost O(1) (except for the particular scenario where a list pivoting is required, the cost for the operation in that case being O(n), n being the number of values already added in the queue)

The main disadvantage of this implementation is the rather complicated user-facing API, as a matter of fact, both the enqueue and dequeue operations return the new RingBuffer rather than modifying the current one. This might lead to mutability references in the client application.

3.5 Concurrent access and modification

In a real-world application a purely immutable queue has limited usages, most of the time it is used as a backpressure mechanism for a continuous stream of data. Because the tasks would be incessantly added and taken from the queue, having a unique queue reference with the mutability reference abstracted from the developer would greatly ease and improve application code readability and reasoning. To achieve this, the RingBuffer immutable and the mutable references need to be separated. The immutable class value that does not need to change is the capacity, being defined during the queue creation time. Size and the Queue references, however, will need to be extracted into a separate case class (Listing 11) for clarity reasons.

```
case class IQueue[A] (size: Int, queue: Queue[A])
```

Listing 11 IQueue case class

To ensure the atomic reference of the IQueue object whenever it is modified by different threads at once, the java AtomicReference (java.util.concurrent.atomic.AtomicReference) will have to be introduced for our IQueue case class. This will provide the necessary concurrency primitives for correctly resolving and modification of the mutable references.

```
import java.util.concurrent.atomic.AtomicReference
import scala.collection.immutable.Queue

case class IQueue[A] (size: Int, queue: Queue[A])

case class RingBufferRef[A] private (capacity: Int, queue:
AtomicReference[IQueue[A]]) {

  def enqueue(element: A): Boolean = {
    val queueReference = queue.get()
    if (queueReference.size < capacity) {
```

```
      queue.set(IQueue(
        queueReference.size + 1, queueReference.queue.enqueue(element)))
      true
    }
    else false
  }

  def dequeue: Option[A] = {
    val queueReference = queue.get()
    queueReference.queue.dequeueOption match {
      case Some((el, newQueue)) =>
        queue.set(IQueue(queueReference.size - 1, newQueue))
        Some(el)
      case None =>
        None
    }
  }
}

object RingBufferRef {

  def empty[A](capacity: Int): RingBufferRef[A] = RingBufferRef(capacity, new
AtomicReference[IQueue[A]](
    IQueue(0, Queue.empty[A])
  ))

  def apply[A](capacity: Int)(els: A*): RingBufferRef[A] = {
    val elements = if (els.size <= capacity) els else els.takeRight(capacity)
    RingBufferRef(capacity, new AtomicReference[IQueue[A]](
      IQueue(elements.size, Queue(elements: _*))))
  }
}
```

Listing 12 RingBufferRef full implementation

In Listing 12, the initial RingBuffer is reimplemented to use the reference mutability with regards to IQueue object but introduces atomic operations for modifying the content of the RingBuffer. In both enqueue and dequeue methods a fresh copy of the queue is fetched (via queue.get()), and a new queue is set to be referenced in the next iteration of the different methods. However, this implementation has a major drawback, race conditions. The queue.set() method will set the new object reference to whatever value it is given, regardless of what it currently withholds. To put this in perspective, let's imagine a current queue with one value. Thread A and Thread B both try to enqueue a

new value. However, since they are accessing the reference concurrently, they are not aware of the modifications themselves, and both get the queue with one value to modify. No matter which thread is first to enqueue their value, the resulting queue size will always have a size of 2 whereas it should have three values, effectively dropping one accidentally.

The solution to the race condition problem is to introduce a concurrency primitive, compare and swap, already described in paragraph 2.3. Using this particular instruction, the operation could be looped over or tail recursed until a successful result.

```
case class RingBufferCAS[A] private (capacity: Int, queue:
AtomicReference[IQueue[A]]) {

  def enqueue(element: A): Boolean = {
    var compareAndSetResult = false
    var result = false
    while (!compareAndSetResult) {
      val queueReference = queue.get()
      if (queueReference.size < capacity) {
        compareAndSetResult = queue.compareAndSet(queueReference, IQueue(
          queueReference.size + 1, queueReference.queue.enqueue(element)))
        result = true
      } else {
        compareAndSetResult = true
        result = false
      }
    }
    result
  }

  def dequeue: Option[A] = {
    var compareAndSetResult = false
    var result: Option[A] = None
    while (!compareAndSetResult) {
      val queueReference = queue.get()
      queueReference.queue.dequeueOption match {
        case Some((el, newQueue)) => {
          compareAndSetResult = queue.compareAndSet(queueReference,
IQueue(queueReference.size - 1, newQueue))
          result = Some(el)
        }
        case None =>
          {
```

```
                compareAndSetResult = true
                result = None
              }
            }
          }
          result
        }
      }
```

Listing 13 RingBufferCAS full implementation

A refactored version of the RingBufferRef (Listing 12) can be seen in Listing 13. Both enqueue and dequeue methods are implemented using a while loop that withholds as its main condition the mutable result of the compare_and_swap instruction, always trying to eventually enqueue the items until a successful reference is solved.

3.6 A-B-A problem

In multi-threaded computing, the ABA problem occurs during synchronization when a location is read twice; both reads have the same value and " value is the same " indicates that there was no modification. However, another thread can execute between the two reads and change the value, do other work and then change the value back, thus fooling the first thread into thinking that " nothing has changed, " even if the second thread does not work. The problem with ABA occurs when multiple threads (or processes) have access to shared data. The following is the sequence of events that lead to the problem of ABA:

- Process T1 reads value A from shared memory
- T1 is preempted, allowing process T2 to run
- T2 modifies the shared memory value A to value B and back to A before preemption
- T1 begins execution again, sees that the shared memory value has not changed and continued.

Although T1 can continue to execute, due to the " hidden " modification of the shared memory, the behavior may not be right. A common case of ABA problems is found when a lock-free data structure is implemented. If an item is deleted from the list and a new item is assigned and added to the list, it is common for the assigned object to be at the same location as the deleted object due to optimization. A new item pointer is therefore sometimes the same as an old item pointer, which is an ABA problem. This problem arises however mostly not garbage collected languages where the CAS comparison is value based. In Java, every atomically referenced object has a unique stamp updated during the set() method. To make the stamp comparison and setting explicit AtomicStampedReference is provided by java.util.concurrent. (Herlihy and Shavit, 2012)

3.7 Thread safety via synchronization

Despite CAS being overall a great concurrency primitive, it does not represent the only solution for the concurrent access and modification. As previously described in paragraph 2.2 synchronized block represents an intrinsic lock for safeguarding object modification from multiple threads.

```
package immutable
import java.util.concurrent.atomic.AtomicReference
import scala.collection.immutable.Queue

case class IQueue[A] (size: Int, queue: Queue[A])

case class RingBufferSynchronized[A] private (capacity: Int, queue:
AtomicReference[IQueue[A]]) {

  def enqueue(element: A): Boolean = synchronized {
    val queueReference = queue.get()
    if (queueReference.size < capacity) {
      queue.set(IQueue(queueReference.size + 1,
queueReference.queue.enqueue(element)))
      true
    } else {
      false
    }
  }

  def dequeue: Option[A] = synchronized {
    val queueReference = queue.get()
```

```
    queueReference.queue.dequeueOption match {
      case Some((el, newQueue)) =>
        queue.set(IQueue(queueReference.size - 1, newQueue))
        Some(el)
      case None => None
    }
  }
}

object RingBufferSynchronized {

  def empty[A](capacity: Int): RingBufferRef[A] = RingBufferRef(capacity, new
AtomicReference[IQueue[A]](
    IQueue(0, Queue.empty[A])
  ))

  def apply[A](capacity: Int)(els: A*): RingBufferRef[A] = {
    val elements = if (els.size <= capacity) els else els.takeRight(capacity)
    RingBufferRef(capacity, new AtomicReference[IQueue[A]](
      IQueue(elements.size, Queue(elements: _*))))
  }
}
```

Listing 14 RingBufferSynchronized full implementation

Like the previous implementation (Listing 12) of the RingBuffer with AtomicReference, the reference is purely updated to the new object while the concurrent thread modification of the different methods is safeguarded by the synchronization block. By citing the counterexample mentioned in paragraph 3.3, two different threads will not be able to access at the same time the resource, making this implementation thread-safe.

4 Mutable array-based RingBuffer

An array-based implementation of the RingBuffer differentiates from the previous immutable queue implementation in one key aspect, the underlying data structure. To simulate the previously linked list behavior of enqueuing and dequeening in constant time, we need to constantly keep track of the head pointer and the tail pointer.

- The front pointer will always denote to the oldest inserted element in the queue
- The rear pointer will always denote to the newest inserted element in the queue
- Every time a new element is inserted into the queue rear is incremented by one
- Every time an element is deleted from the queue front is incremented by one

4.1 Specifications

The set of rules used to insert a new element in a circular queue are similar to the rules in the case of a linear queue, with some modifications:

1. The first thing to do in the case of inserting a new element is to check if the circular queue is full or not. This is accomplished by performing the following equation: Front = (Rear + 1) % Max Size, where max size is the total number of slots in the circular queue.

2. The user will insert the new element.

3. If the inserted element is the first element in the circular queue, the front, and rear pointers will denote to the first location which zero. The circular queue is empty if the front pointer is equal -1.

4. If the newly inserted element is not the first element in the circular queue, the location of this element will be calculated as follows: Rear = (Rear + 1) % Max Size.

The deletion method for a circular queue also requires some modification as compared to a linear queue. The used rules are:

1. Check if the value of the front pointer equals -1 or not. If yes, the circular queue is empty, and the underflow situation existed.

2. Check if the value of front and rear pointers are the same. If yes, this means there is only one element in the circular queue, and it is deleted by setting the value of front and rear pointers to -1.
3. Otherwise, the front pointer value will be modified as follows:
   Front = (Front + 1) % Max Size

4.2 Implementation

```
import java.util.concurrent.atomic.AtomicReference
import scala.reflect.ClassTag

case class RingBufferValues(front: Int, rear: Int)

case class ArrayRingBufferCAS[A](capacity: Int, array: Array[A]) {

  private val ringBufferSettings = new
AtomicReference[RingBufferValues](RingBufferValues(-1, 0))

  def size(ringBufferValues: RingBufferValues): Int = {
    ringBufferValues.rear - ringBufferValues.front
  }

  def enqueue(value: A): Boolean = {

    var compareAndSetResult = false
    var result = false

    while(!compareAndSetResult) {
      val ringBufferValues = ringBufferSettings.get()
      val rearArrayValue = (ringBufferValues.rear + 1) % array.length
      if (!(size(ringBufferValues) >= capacity)) {
        val frontValue =
          if(ringBufferValues.front == -1) 0
          else ringBufferValues.front
        compareAndSetResult = ringBufferSettings
          .compareAndSet(ringBufferValues, RingBufferValues(frontValue,
rearArrayValue))
        if(compareAndSetResult) {

          array.update(ringBufferValues.rear, value)
```

```
        }
        result = true
      }
      else {
        compareAndSetResult = true
        result = false
      }
    }

    result
  }

  def dequeue(): Option[A] = {
    var compareAndSetResult = false
    var result: Option[A] = None
    while (!compareAndSetResult) {

      val ringBufferValues = ringBufferSettings.get()
      if (ringBufferValues.front == -1) {
        compareAndSetResult = true
        result = None
      } else if(ringBufferValues.rear == ringBufferValues.front) {
        val tmp = array(ringBufferValues.front % array.length)
        compareAndSetResult = ringBufferSettings
          .compareAndSet(ringBufferValues , RingBufferValues(-1, 0))
        result = Some(tmp)
      } else {
        val tmp = array(ringBufferValues.front % array.length)
        val newFront = ringBufferValues.front + 1
        compareAndSetResult = ringBufferSettings
          .compareAndSet(ringBufferValues          ,RingBufferValues(newFront,
ringBufferValues.rear))
        result = Some(tmp)
      }
    }
    result
  }

}

object ArrayRingBufferCAS {

  def empty[A: ClassTag](capacity: Int): ArrayRingBufferCAS[A] = {
    val array = Array.ofDim[A](capacity)
    ArrayRingBufferCAS[A](capacity, array)
```

```
  }

  def apply[A: ClassTag](capacity: Int)(els: A*): ArrayRingBufferCAS[A] = {
    val elements = if (els.size <= capacity) els else els.takeRight(capacity)
    val array = Array.ofDim[A](capacity)
    val arrayRingBuffer = ArrayRingBufferCAS[A](capacity, array)
    elements.foreach(el => arrayRingBuffer.enqueue(el))
    arrayRingBuffer
  }

}
```

Listing 15 ArrayRingBufferCAS full implementation

The array-based RingBuffer implementation defined in Listing 13 is and should be like our compare and swap immutable implementation defined in Listing 11, with a few key differences. The underlying data structure is a mutable java array instead of the scala's very own immutable queue. In practice, this means that while we have a unique reference to it, it can undergo modifications at any given moment. The second key difference is the introduction of the atomic reference for the front and rear pointer. These two values effectively represent our queue, being the distinctive way of determining if the queue is full, if it has values or if it is empty. Hence, they need to be atomically modified. The insertion and deletion rules defined in paragraph 4.1 are implemented in the enqueue and respectively dequeue method.

5 Benchmarking

To thoroughly test the immutable queue-based and the mutable array-based RingBuffers, basic unit tests are needed to ensure the correct Queue behavior, be it in a single or multi-thread parallel environment. Also, benchmarking tests will be performed comparing the RingBuffer implementations, and the presented Java Concurrent thread-safe concurrent queues in paragraph 2.4. The unit tests will be performed using scalatest, the most popular testing tool in the Scala ecosystem. The benchmarking framework chosen for the testing is ScalaMeter, a regression test, and micro benchmarking for the JVM platform.

5.1 Unit testing

RingBuffer's API exposes to methods that modify circular buffer's internal state, namely the queue and enqueue methods. To progress with the benchmarking tests, the correct functionality needs to be validated. For this unit tests, a new list of elements (of type Int) will be created. The list will be parallelly traversed (to test the concurrency aspect) until all the elements will be enqueued. For the dequeue operation the auxiliary JavaConcurrentLinkedQueue will be used to store the taken elements. The initial list will be once again traversed in parallel and the dequeued items will be added to the auxiliary data structure. The full snippet of the unit tests is provided below (Listing 16).

```
import java.util.concurrent.ConcurrentLinkedDeque

import immutable.{RingBufferCAS, RingBufferSynchronized}
import mutable.ArrayRingBufferCAS
import org.scalatest.FunSuite
import org.scalatest.concurrent.Eventually
import org.scalatest.time.{Millis, Seconds, Span}

class MainTest extends FunSuite with Eventually {

  override implicit val patienceConfig: PatienceConfig =
    PatienceConfig(timeout =  Span(3, Seconds), interval = Span(5, Millis))

  test("RingBuffer CAS enqueue and dequeue") {

    val list = List(1,2,3,4,5,6,7,8,9,10)
```

```
  val queue = RingBufferCAS.empty[Int](20)
  list.par.foreach{
    el =>
      queue.enqueue(el)
  }

  val concurrentLinkedQueue = new ConcurrentLinkedDeque[Int]()
  list.par.foreach{ _ =>
    val el = queue.dequeue
    if (el.isDefined) concurrentLinkedQueue.add(el.get)
  }

  eventually {
    assert(concurrentLinkedQueue.size==10)
    list.foreach{
      el =>
        assert(concurrentLinkedQueue.contains(el))
    }
  }
}

test("RingBuffer Synchronized enqueue and dequeue") {

  val list = List(1,2,3,4,5,6,7,8,9,10)

  val queue = RingBufferSynchronized.empty[Int](20)
  list.par.foreach{
    el =>
      queue.enqueue(el)
  }

  val concurrentLinkedQueue = new ConcurrentLinkedDeque[Int]()
  list.par.foreach{ _ =>
    val el = queue.dequeue
    if (el.isDefined) concurrentLinkedQueue.add(el.get)
  }

  eventually {
    assert(concurrentLinkedQueue.size==10)
    list.foreach{
      el =>
        assert(concurrentLinkedQueue.contains(el))
    }
  }
}
```

```
  test("ArrayRingBuffer CAS enqueue and dequeue") {

    val list = List(1,2,3,4,5,6,7,8,9,10)

    val queue = ArrayRingBufferCAS.empty[Int](20)
    list.par.foreach{
      el =>
        queue.enqueue(el)
    }

    val concurrentLinkedQueue = new ConcurrentLinkedDeque[Int]()
    list.par.foreach{ _ =>
      val el = queue.dequeue()
      if (el.isDefined) concurrentLinkedQueue.add(el.get)
    }

    eventually {
      assert(concurrentLinkedQueue.size==10)
      list.foreach{
        el =>
          assert(concurrentLinkedQueue.contains(el))
      }
    }
  }
}
```

Listing 16 Unit tests code

The assertions done in the unit tests are the following:

- Compare the size of the auxiliary data structure and the size of the initial list of elements. For the tests to pass, verify that the two sizes are the same value
- Traverse the initial list of elements and assert that each element is contained in the auxiliary list

Both conditions were satisfied during the unit tests to run, establishing that the concurrency did not lead to data corruption and the output of the queue is that same that is to expect from a canonical queue. Considered the parallel aspect of element insertion the ordering of the elements cannot be guaranteed; hence, this aspect was to be omitted. Unit tests were run on a MacBook Pro 13' 2014, with an i5 dual-core CPU, Scala version 2.12.7, scalatest version 3.0.5 and OpenJDK version jdk1.8.0_192.

5.2 Benchmark test specifications

ScalaMeter's API provides easy specifications for dataset generation. The performance test run will use a big enough range of data, starting from 300000 elements up to 1500000, by introducing a step of 500000 records between the multiple datasets. The datatype of the record will be represented by Integers (the elements to be enqueued and dequeued).

```
  val sizes = Gen.range("size")(300000, 1500000, 500000)

  val ranges = for {
            size <- sizes
  } yield 0 until size
```

Listing 17 Benchmarking Specifications

In Listing 17 is represented as a simple generator of records using Scalameter. A set of sizes is created using the step function, after which the ranges that are going to be used for the enqueuing/dequeuing are generated.

```
object PerformanceTest extends Bench.LocalTime {

  val sizes = Gen.range("size")(300000, 1500000, 500000)

  val ranges = for {
            size <- sizes
  } yield 0 until size

  performance of "ArrayRingBufferCAS" in {
            measure method "enqueue" in {
              using(ranges) in {
            r =>
            val arrayRingBuffer = ArrayRingBufferCAS.empty[Int](r.size)
            arrayRingBuffer.map{ i => arrayRingBuffer.enqueue(i)}
            }}}
}
```

Listing 18 Local Benchmarking test snippet

Listing 18 provides a simple test snippet for a performance test. The performance is logged directly in the console during the test run. This test, however, has some limitations:

- Enqueuing and dequeuing is sequential; parallelism is needed to provide meaning for RingBuffer's concurrent functionality
- The tests are executed sequentially in the same JVM; this implies for both JVM warmup period (tests running cold JVM will be less performant) as well as the cache coherence

ScalaMeter offers various alternatives for the second problem specified in the previous paragraph. The PerformanceTest Object should extend the Bench trait, which in turn would allow defining the three main parts of the testing pipeline: the measurer, the persistor and the executor. The executor decides how the tests will be executed. To guarantee tests isolation (running a separate JVM for each test), the executor will be changed to a special SeparateJvmsExecutor. The heap size for new JVM containers has a default heap size set to 2GB. The Executor used in this test executes a fixed number of measurements, takes the minimum execution time as the default measure and will apply the same JVM warming period for each test.

To simulate the multiple producer-multiple consumer scenarios, the source list of records can be parallelized. Scala's List interface provides a utility method to specify the parallelism number for the parallel execution as well as implement a custom ForkJoinPool where it is possible to specify the desired number of execution threads.

```
import java.io.File
import java.util.concurrent.{ArrayBlockingQueue, ConcurrentLinkedDeque,
ForkJoinPool, LinkedBlockingQueue}

import mutable.ArrayRingBufferCAS
import immutable.{RingBufferCAS, RingBufferSynchronized}
import org.scalameter.api._
import org.scalameter.picklers.Implicits._
import org.scalameter.Bench.OfflineReport

import scala.collection.parallel.ForkJoinTaskSupport


object PerformanceTest extends OfflineReport {

  override val executor = SeparateJvmsExecutor(
    New Executor.Warmer.Default,
    Aggregator.min,
    new Measurer.Default
  )

  override val reporter: Reporter[Double] = Reporter.Composite(
    new RegressionReporter(
      RegressionReporter.Tester.OverlapIntervals(),
      RegressionReporter.Historian.ExponentialBackoff() ),
    HtmlReporter(true)
  )

  override def persistor: Persistor = JSONSerializationPersistor(new
File("target/benchmarks/sun"))

  val sizes = Gen.range("size")(300000, 1500000, 500000)
  val taskSupport = new ForkJoinTaskSupport(new ForkJoinPool(5))

  val ranges = for {
    size <- sizes
  } yield 0 until the size
```

```
  performance of "ArrayRingBufferCAS" in {
    measure method "enqueue" in {
      using(ranges) in {
        r =>
          val arrayRingBuffer = ArrayRingBufferCAS.empty[Int](r.size)
          val parallel = r.par
          parallel.tasksupport = taskSupport
          parallel.map{ i => arrayRingBuffer.enqueue(i)}
      }
    }
  }

  performance of "immutable.RingBufferCAS" in {
    measure method "enqueue" in {
      using(ranges) in {
        r =>
          val ringBuffer = RingBufferCAS.empty[Int](r.size)
          val parallel = r.par
          parallel.tasksupport = taskSupport
          parallel.map{ i => ringBuffer.enqueue(i)}
      }
    }
  }

...

  performance of "ArrayRingBufferCAS" in {
    measure method "dequeue" in {
      using(ranges) in {
        r =>
          val arrayRingBuffer = ArrayRingBufferCAS.empty[Int](r.size)
          val parallel = r.par
          parallel.tasksupport = taskSupport
          parallel.map{ i => arrayRingBuffer.enqueue(i)}
          parallel.map{ i => arrayRingBuffer.dequeueOption()}
      }
    }
  }

...

  performance of "immutable.RingBufferSynchronized" in {
    measure method "dequeue" in {
      using(ranges) in {
        r =>
          val ringBuffer = RingBufferSynchronized.empty[Int](r.size)
          val parallel = r.par
          parallel.tasksupport = taskSupport
          parallel.map{ i => ringBuffer.enqueue(i)}
          parallel.map{ i => ringBuffer.dequeue}
      }
    }
  }

  performance of "ConcurrentLinkedQueue" in {
    measure method "dequeue" in {
      using(ranges) in {
        r =>
          var linkedQueue = new ConcurrentLinkedDeque[Int]()
          val parallel = r.par
          parallel.tasksupport = taskSupport
          parallel.map{ i =>
            linkedQueue.add(i)
          }
          parallel.map{ i =>
            linkedQueue.pop()
          }
```

```
        }
      }
    }

  . . .
}
```

Listing 19 Full benchmarking test snippet

Listing 19 provides an almost complete snippet of the benchmarking test. The RingBufferCAS (mutable and immutable) and RingBufferSynchronized data structures implemented in paragraphs 4.2, 3.4 and 3.6 will be benchmarked against the already described Java Concurrent data structures in paragraph 2.6.
The benchmarking tests will be reproduced on a MacBook Pro 13' 2014, with an i5 dual-core CPU, Scala version 2.12.7, scalaMeter version 0.10.1 and OpenJDK version jdk1.8.0_192.

5.3 Results

The results in this section will be presented using two different charts: a linear chart and a bar chart. The linear chart will help to visualize the slope describing the time spent (Y-axis) enqueuing a certain number of elements (X-axis). On the other hand, the bar chart aims to put into perspective the actual performance of the different queue implementations. The graph's legend is on the left of the graph

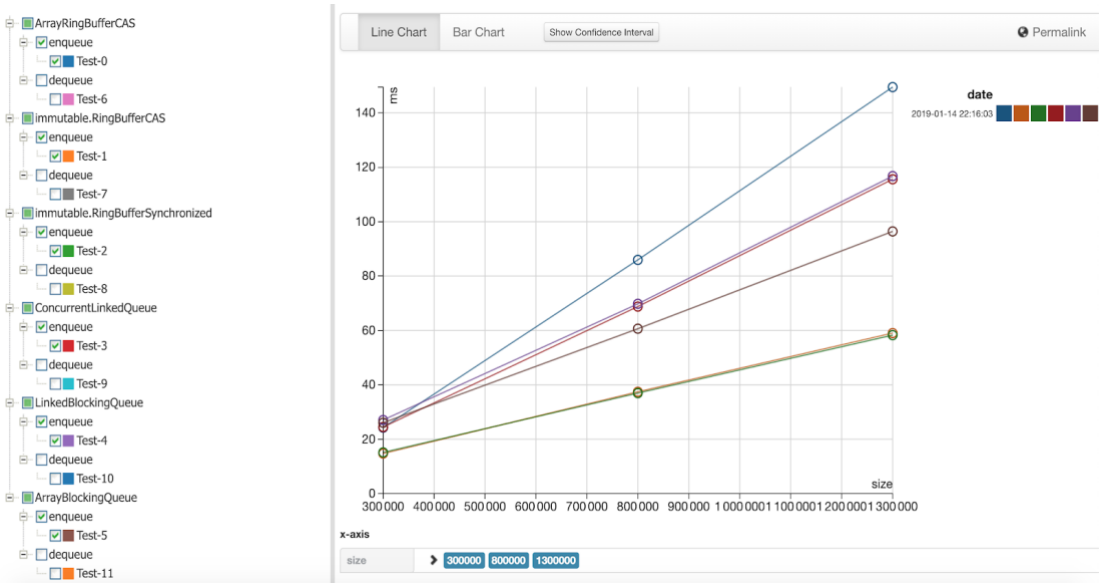

Figure 1 Enqueue performance test - line chart view

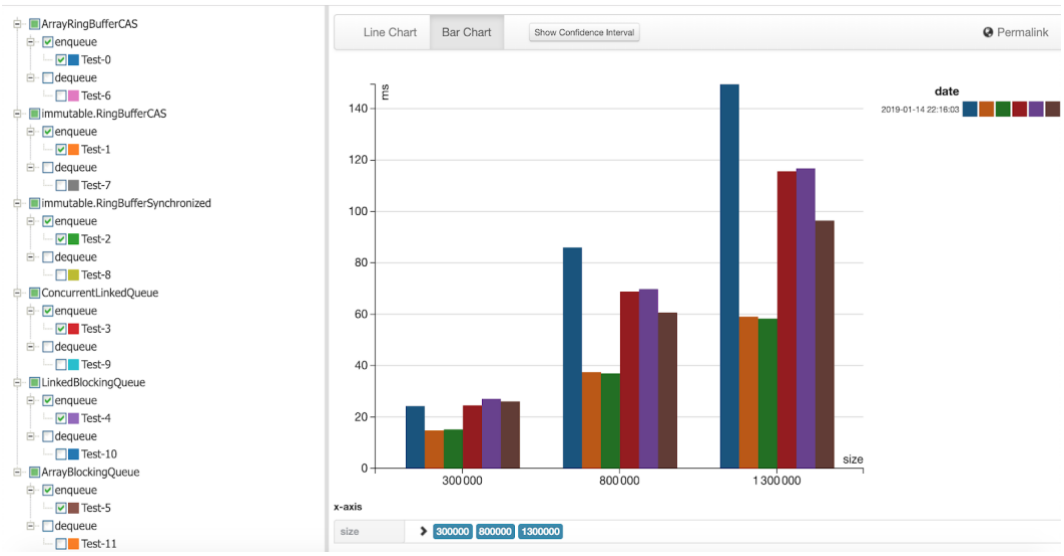

Figure 2 Enqueue performance test - bar chart view

The enqueuing process of the performance tests, where the main measurement is done by timing the parallel offering of elements, is visible in Figure 1 and 2. In the first chart, it is possible to notice the linearity that describes time spent / elements enqueue proportion, in the bar chart instead the difference among the different implementations is more visible. The mutable implementation of the RingBuffer (Array-based) is the worst performer, the growth rate and the time spent on the operation being much higher than the remaining choices. The immutable implementations of the RingBuffer, on the other hand, perform on average better than the java concurrent data structures, with no notable difference among the CAS and Synchronized implementation.

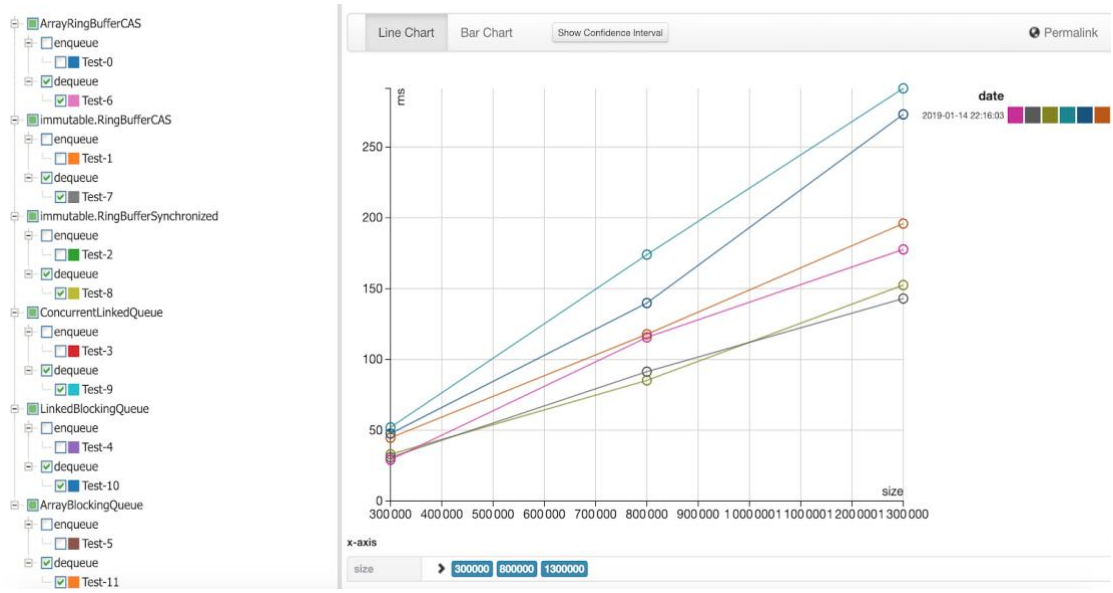

Figure 3 Dequeue performance test - line chart view

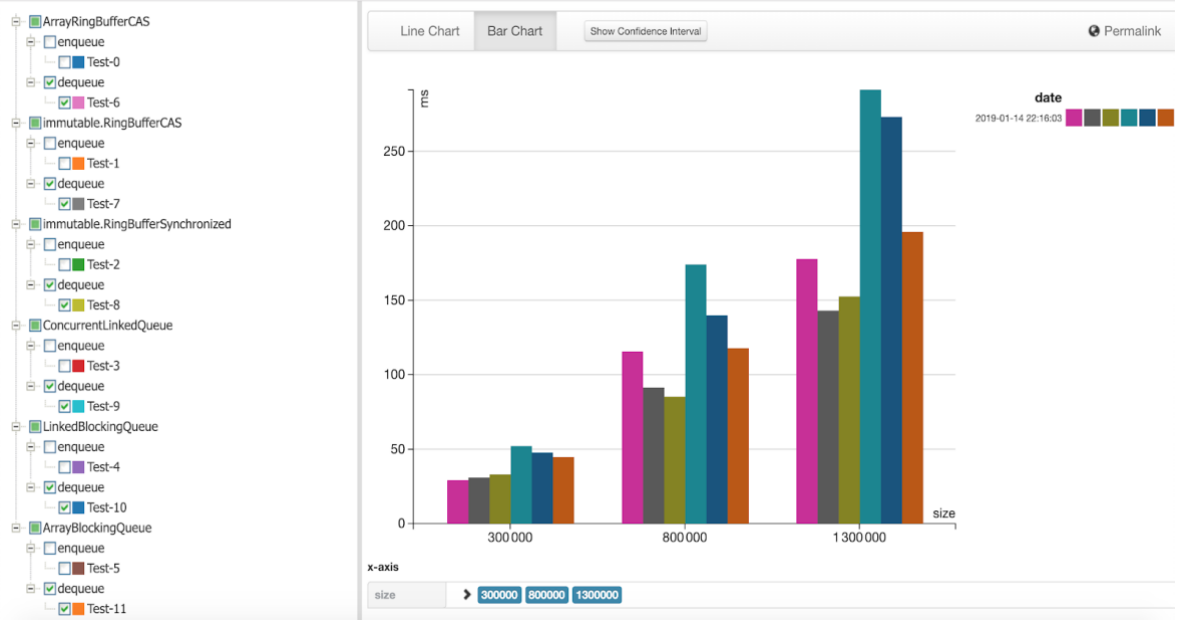

Figure 4 Dequeue performance test - line chart view

In the following test where the dequeuing operation is taking into account as well (Figure 3 and 4) the mutable Array-based RingBuffer's performance is comparable to the ArrayBlockingQueue while being faster than LinkedBlockingQueue and ConcurrentLinkedQueue. The immutable implementations perform similarly to the first tests, outpacing the mutable RingBuffer and the Java implementations.

Considered that the implemented RingBuffer (both immutable and mutable) is the only bounded non-blocking implementation among the Java Concurrent queues, the performance test was designed to only measure the performance while overlooking memory management and specific API methods derived from blocking operations. From the preliminary results listed above, however, it emerged that the immutable RingBuffer implementations can outperform the Java implementations while maintaining a steady growth time rate under more load. To conclude, in the initial outline thesis goals (paragraph 1.1), the base hypothesis was that it is possible to design a faster data structure that would satisfy the concurrent and nonblocking requirements. The preliminary results obtained in this performance test reinforce that assumption with concrete results.

6 Conclusion

The main goal of this thesis was to provide a high performing implementation of an abstract Queue as a RingBuffer, a bounded circular queue. The performance itself is related to the usage of parallelization of tasks, producing and consuming. Given that multiple sources are trying to concomitantly modify the resources, the implementation had to be thread-safe to guarantee the data integrity of the RingBuffer.

In chapter 2, the concurrency peculiarities of JVM and JMM, the available high- and low-level functionalities, as well as, presenting the implications and the reason they were introduced were closely studied. In chapter 3 and 4, starting from a naïve RingBuffer implementation, three different implementations were presented: RingBufferCAS, RingBufferSynchronized, and ArrayRingBufferCAS. While the difference between the first two implementations is minimal in design, as they share the common underlying data structure (Scala's standard library immutable Queue), the third implementation is based on a mutable Array with a fixed size that performs as a circular queue. Chapter 5 provides a benchmark of the previously described implementations and Java's concurrent queue implementations. The benchmark focuses primarily on tracking the speed of enqueuing and dequeuing and does not cover other aspects like memory footprint or API extensibility, which should be taken into consideration when choosing the data structure for the specific problem on hand.

The contribution of this research is to demonstrate that while sound and well-tested implementations of concurrent queues in Java are fast enough for most of the tasks, alternative implementations using underlying immutable data structures and concurrency primitives for concurrent access can be achieved without sacrificing the performance. This promises better results according to the preliminary benchmarks (chapter 5). There are some implied limitations, aimed to be addressed in further research, including memory management, garbage collection overhead, as well as, a complete API testing, to reduce the gap between production-ready data structure implementation and the presented proof of concept in this thesis.

References

Cs.umd.edu., 2004. *JSR-133: JavaTM Memory Model and Thread Specification*. [online] Available at: http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf [Accessed 3 Feb. 2019]

Dey, S. and S. Nair, M. , 2014. Design and Implementation of a Simple Cache Simulator in Java to Investigate MESI and MOESI Coherency Protocols. *International Journal of Computer Applications*, 87(11), pp.6-13

Docs.scala-lang.org., 2019. *Variances | Scala Documentation*. [online] Available at: https://docs.scala-lang.org/tour/variances.html [Accessed 11 Feb. 2019]

Erb, B., 2012. *Concurrent Programming for Scalable Web Architectures*. Graduate. Institute of Distributed Systems Faculty of Engineering and Computer Science Ulm University

Goetz, 2006. *Java Concurrency In Practice*. Pearson Education, pp.23-27.

Herlihy, M. and Shavit, N., 2012. *The art of multiprocessor programming*. Waltham, MA: Morgan Kaufmann

Hewitt, C., Bishop, P. and Steiger, R., 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. [online] p.235. Available at: http://worrydream.com/refs/Hewitt-ActorModel.pdf [Accessed 11 Feb. 2019]

Hoare, C., 1978. Communicating sequential processes. *Communications of the ACM*, [online] 21(8), pp.666-677. Available at: https://www.cs.cmu.edu/~crary/819-f09/Hoare78.pdf

Java™ Platform, Standard Edition 7 API Specification, 2018. *ConcurrentLinkedQueue<E>* [online] Available at: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html [Accessed Oct 3, 2017]

Java™ Platform, Standard Edition 7 API Specification, 2018. *ArrayBlockingQueue<E>*. [online] Available at: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html [Accessed Oct 3, 2017]

Java™ Platform, Standard Edition 7 API Specification, 2018. *LinkedBlockingQueue<E>*. [online] Available at: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html [Accessed Oct 3, 2017]

Kwiatkowska, M., 1989. *Fairness for Non-Interleaving Concurrency*. [online] Pdfs.semanticscholar.org. Available at: https://pdfs.semanticscholar.org/a235/214d3d8e8991b19fa842f087d1d723126bd6.pdf [Accessed 3 Feb. 2019]

Lamport, L., 1979. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. [online] Microsoft.com. Available at: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/How-to-Make-a-Multiprocessor-Computer-That-Correctly-Executes-Multiprocess-Programs.pdf [Accessed 3 Feb. 2019]

Sundell, H. and Tsigas, P., 2004. *Lock-Free and Practical Deques using Single-Word Compare-And-Swap*. [online] Pdfs.semanticscholar.org. Available at: https://pdfs.semanticscholar.org/1576/f9175c495fe8ade14ac4c53eb1940107b723.pdf [Accessed Oct 3, 2017]