# Java concurrency benchmark

## Evaluating performance of Java concurrency mechanisms

Martin Andersson

Department of Computer and Systems Sciences

Degree project 15 HE credits
Degree subject (Computer and Systems Sciences)
Degree project at the bachelor level
Spring term 2017
Supervisor: Peter Idestam-Almquist
Reviewer: Lars Asker
Swedish title: N/A


Stockholm University

# Java concurrency benchmark

**Evaluating performance of Java concurrency mechanisms**

**Martin Andersson**

# Abstract

There are many ways to accomplish thread-safe code in Java. The keyword `synchronized` has long been one of those mechanisms. In year 2004, Java's standard library version 1.5 introduced a wide array of high-level data structures and primitives to accomplish thread-safe code [1].

This paper will benchmark the performance of various constructs to accomplish thread-safe code in a real-world context; using an in-memory non-distributed message queue service. We measure throughput of read and write operations.

We find that concurrent data structures provided by JDK consistently outperform explicit thread synchronization with the keyword `synchronized` or the data type `ReentrantReadWriteLock`. We also find that `synchronized` outperformed `ReentrantReadWriteLock`.

### Keywords

Java, concurrency, performance, benchmark, messaging, queue, thread-safe, synchronized, ConcurrentMap, Lock, ReadWriteLock, JMH, Gradle

### Synopsis

| | |
|---|---|
| *Background* | Concurrency is on the rise and hard to get right. |
| *Problem* | No benchmark available using a real-world application component. |
| *Research question* | How does the performance of Java's mechanisms to achieve thread-safe code compare? |
| *Method* | Experiment using a queue service framework with different implementations. |
| *Result* | `java.util.concurrent` win on total throughput, followed by `synchronized` and `ReentrantReadWriteLock`. |
| *Discussion* | `java.util.concurrent` is almost always a good choice and the slowness of `ReentrantReadWriteLock` compared to `synchronized` might be explained by short read operations. |

# Table of Contents

# List of Figures

# 1 Introduction

## 1.1 Background

There is a lot of pressure on modern applications to be responsive and performant. The average application user "prefer response times well below one second" [2]. It is reported that incremental delays as small as 100 milliseconds "would result in substantial and costly drops in revenue" [3]. As an example of a high performance application, NGINX is a web server that can handle up to "hundreds of thousands of concurrent connections" [4].

CPU manufacturers has reached a point where it is exponentially harder to increase the clock rate of a CPU, finding it more convenient to add more cores to the same CPU instead [5] [6].

A high performance application that wants to make the best use of his host system might find that utilizing more than just one core is an attractive offer. Yet, writing concurrent code is hard [7, p. 59] and synchronizing threads will always impose an overhead that is not free of cost [6].

Java is an object-oriented and strongly-typed language that enable concurrent code. Java has over time gathered a lot of first-class constructs and library-provided mechanisms to make code thread-safe. The plethora of options available to the Java developer is as powerful as it may be confusing.

For more information related to thread-safety and concurrent Java, see "Appendix A: Extended background".

Little research was found that compare performance of various constructs the Java developer may deploy to accomplish thread-safe code. No research was found that take this performance comparison and put it in the context of a real-world application component. Yet, benchmarking code is best done in a real-world context [8]. With a too narrow focus, the results are not descriptive of real-world applications.

An in-memory non-distributed queue service is a good candidate for being a software component that can be made to illustrate and performance-compare Java mechanisms for thread-safe code. The component, under a high concurrent load, will be accessed by a lot of different threads that share messages with each other. A queue service also makes something abstract into a concrete and measurable metric such as "messages per microsecond" (one millionth of a second).

## 1.2 Problem

[9, p. 294] writes that "Java built-in locks [...] have long been a performance concern" and [7, p. 79] notes a growing tendency for developers to use high-level concurrency constructs. Combining these quotes indicates that Java developers are biased into applying the latest tool or mechanism in hope of getting the most performant solution.

For example, a Java developer in need of a mutually exclusive lock might *always* choose to use an implementation of Java's `Lock` interface instead of Java's `synchronized` keyword even at times when `synchronized` would be a plausible alternative. In contrast to some of these preconceived views, Oracle claims that their implementation of `synchronized` is "ultra-fast" and "not a significant performance issue" [10].

As noted in section “1.1 Background”, little research was found that benchmark Java concurrency code in a real-world application component. [11] find that performance testing is not even common in Java-based open source projects.

This study makes a contribution into filling the knowledge gap.

## 1.3 Research Question

Using a queue service framework, how does the performance of Java’s mechanisms to achieve thread-safe code, compare to each other?

These are the combinations this study investigates:

1. The `synchronized` keyword, `HashMap`, `ArrayDeque`
2. `ReentrantReadWriteLock`, `HashMap`, `ArrayDeque`
3. `ConcurrentHashMap`, `ConcurrentLinkedDeque`
4. `ConcurrentHashMap`, `ConcurrentLinkedDeque`, `AtomicReference`

Each combination is put into a concrete implementation of a queue service that will be benchmarked. This paper will present and analyze the number of push- and poll operations per microsecond that each queue service implementation can deliver.

The implementations are named `SynchronizedQS`, `ReadWriteLockedQS`, `ConcurrentQSWithPojoMessage` and `ConcurrentQSWithAtomicMessage` respectively. See section “3.1.3 Queue service being utilized” for a description of each implementation and the framework they have in common.

## 1.4 Limitations

This study is limited to only a handful of Java’s concurrency utilities and only one queue service framework; the architecture we put our mechanisms into (see section “3.1.3 Queue service being utilized”).

Furthermore, this study will not deterministically assert the accuracy of the queue service framework. The queue service implementations derived from the framework is assumed to be correct. If this is not the case, then the results of this study may be invalid.

# 2 Research Strategy and Methods

## 2.1 Chosen Research Strategy

This study conducts a highly reproducible experiment.

According to Denscombe [12, p. 67]:

> the aim is to show that the dependent factor [..] responds to changes in the independent factor [..].

My dependent factor is the performance, as measured in operations per microsecond (throughput).

I have many independent factors. One of big importance is the different mechanisms to achieve thread-safe code. Everything in the experiment environment is influential. For example, how many message producing threads versus message consuming threads there is. The time it takes to produce or consume a message. The queue service implementation. Hardware, Java runtime implementation, operating system, and so on.

Java Microbenchmark Harness (JMH) [13] is used to conduct the experiment by executing benchmarks. The primary reason for this choice is to avoid or minimize the pitfalls of benchmarking, as described in the next section.

The JMH benchmark code and queue service framework is part of a GIT repository that can be cloned from GitHub [14]. The repository makes the experiment and environment highly reproducible.

## 2.2 The pitfalls of benchmarking

Although we wish to see the effects of the benchmark code in isolation, we will suffer from disturbance imposed by things such as thread scheduling, operating system background tasks and garbage collection.

It is hard to eliminate or reduce this disturbance. Instead, we opt to smooth out the noise over many measurement runs, or iterations.

Just-in-time compilation (JIT) introduce optimizations that kick in only after some execution time has passed. In fact, optimized code can even become deoptimized. Drawing conclusions too fast too soon can lead the researcher into underestimating the application's performance and invalidating the reliability as the code being tested would behave differently out in the real world.

One approach to counter JIT and optimizations is to use warmup iterations and start data collection only once it is believed that the benchmark code has reached a close enough steady state.

Some pitfalls require a very good understanding of compilers in order to write code that does not produce a skewed benchmark result. For example, the runtime can remove (not execute) dead code. Similarly, there are other benchmark-related pitfalls that require action in the benchmark setup. For example, but certainly not limited to: Predictable return values may cause constant folding. A loop may be unrolled. Different benchmarks may share code profiles which are the foundation of many optimizations.

JMH is a useful tool to build Java benchmarks (not only used for micro benchmarks!). JMH will not free the code author from all benchmark pitfalls but will make it much easier to handle them. How JMH accomplish this task is described in section “3.1.2 Introduction to JMH”.

## 2.3 Alternative Research Strategies

There are many potential alternative research strategies that may or may not also influence the research question.

One is *grounded theory*. This strategy generates theories through a systematic analysis of the data [12, p. 107]. Denscombe describes the researcher as embarking “on a voyage of discovery” [12, p. 108]. Applied in our context, the researcher could study how software companies write concurrent applications and maybe discover completely new constructs to accomplish thread-safety. This approach does not resonate well with the research question presented in this study.

Being more focused on a particular setting is known as a case study. Denscombe writes that cases are “selected on the basis of known attributes” [12, p. 56]. Through a case study approach, the researcher could find and compare already made queue service implementations. A case study approach could also focus on other more qualitative aspects such as user experience and market penetration of various constructs to accomplish thread-safe code.

Denscombe writes that the “most vulnerable” part of case studies is the “credibility of generalizations made from its findings” [12, p. 62]. Using an experiment increase the credibility since many factors in our environment is controlled. Furthermore, the experiment is reproducible.

## 2.4 Chosen Data Collection Method

Quantitative data is observed from the experiment.

Not relying on external data sources is a great benefit of the experiment research strategy. The experiment is feasible to perform with a rather low resource investment.

## 2.5 Alternative Data Collection Methods

The grounded theory researcher could conduct unstructured interviews and take field notes. The case study researcher could collect documents such as technical documentation and application log files.

## 2.6 Chosen Data Analysis Method

We profile the observations using descriptive statistics. The measures we present is measures of central tendency (arithmetic mean) and variability (standard deviation, range).

The measurement that leads our analytical work is the mean operational throughput. Each operation is either a write or a read + immediate completion of a message.

Mean, or average, is sometimes prefixed with “arithmetic” as to distinguish it from other means, such as the geometric mean [15].

Denscombe writes that the mean is “affected by extreme values [..] and can shift the figure for the mean towards any such ‘outliers’” [12, p. 248].

Standard deviation describes “how far, on average, the [observations] varied from the mean” [12, p. 252]. Many standard deviations can be compared to see how stable the results are over time.

If extreme values or incohesive standard deviations are discovered, then the number of iterations (observations) can easily be increased.

Denscombe has two things to say about calculating these numbers.

[12, p. 252]:

> computer software will generally take care of the mathematics involved.

[12, p. 43]:

> There is a formula [..] but the maths do not need to concern the project researcher because there are plenty of software utilities [..] that do the calculations at the press of a button.

This is certainly the case for us. The benchmark runner JMH produces all descriptive statistics. Which formula JMH used is not a concern for this study.

## 2.7 Alternative Data Analysis Methods

The grounded theory researcher could code and categorize his data [12, p. 115]. Then apply a constant comparative method to analyze the data [16]. The goal is to generate theories.

Depending on the nature of the data, the case study researcher could use one of, or combine, a qualitative and quantitative analysis. One approach for doing a qualitative analysis is thematic analysis which set out to discover patterns in the data.

# 3 Application of methods

## 3.1 Strategy

### 3.1.2 Introduction to JMH

Addressing the concerns outlined in the section "2.2 The pitfalls of benchmarking", dead code elimination can be avoided by passing output values to JMH for correct disposal through a `Blackhole` parameter or as a benchmark method return value [17]. Constant folding is eliminated by reading input values from `State` objects that are managed by JMH [18]. Loops are not necessary to write as JMH take care of looping through measuring time-based benchmark runs or batch size controlled one-time iterations (JMH call this a "single shot" measurement mode) [19].

Some optimizations are taken care of by JMH and need little next to no inception by the benchmark author. For example, JMH tries to avoid false sharing by padding `State` objects [20]. JMH also minimize the risk of profile sharing between benchmarks [21] and eliminate memory garbage leaked from a previously executed benchmark by running each benchmark in a new JVM process – referred to as forking. Many forks (runs) is used to better measure run-to-run variance (not to be confused with many *iterations* which occur within the same JVM) [22].

### 3.1.3 Queue service being utilized

A `QueueService` (application-defined type) interface provide an API over many different implementations. This interface is what unit tests and performance benchmarks use.

Many concrete `QueueService` implementations are provided that are different only in how thread-safety is accomplished. All measured performance differences between the implementations can safely be attributed to changes in how thread-safety was accomplished.

A `QueueService` client pushes a message given a specified queue name:

```
void push(String queue, String message);
```

A client polls a message given a specified queue:

```
Message poll(String queue);
```

Once a consumer has finished processing the message, it must be marked as completed:

```
void complete(Message message);
```

If a client fails to mark a message as completed within a certain time period, the message will automagically be put back as the head of the queue and hopefully picked up by another consumer.

Repeated calls to `push()` with the same `String` instance will put the same message over and over again on the same queue, growing the queue size with one element each time. This is primarily the reason why `poll()` return a complex type `Message` (application defined type) in order for the service implementation to know exactly which message is being marked as completed.

The service implementations must support *at-least-once* delivery but may offer stronger *exactly-once* delivery semantics (assuming clients do not time out).

#### 3.1.3.1 The framework: `AbstractQS`

An almost complete queue service implementation is provided by `AbstractQS`. This abstract class model a queue service by *mapping* queue names to actual *queues* of messages.

`AbstractQS` will use one global `Map` instance whose life-cycle is dependent/bound to the queue service instance. Each queue name map to exactly one `Queue` instance. These queue instances may come and go. `AbstractQS` will attempt to remove empty queues during message polling.

Exactly which `Map`, `Queue` and `Message` implementations to use, and how to access them in a thread-safe manner is not defined by `AbstractQS`.

The missing details are provided by the concrete implementation through passing a `Configuration` (application defined type) object up the inheritance chain. Most importantly, the `Configuration` object encapsulates `Map` and `Queue` as `Lockable` (application defined type) entities that enable the framework to describe if it wants to perform a *read*, *write* or *unsafe* (bypasses locking/synchronization) operation on said entity. The framework has no dependency on actual implementation types of the collaborators.

Another part of the framework is `AbstractMessage` (application defined type), an implementation of `Message`. `AbstractMessage` encapsulates a queue and message content and provide behavior specific for `AbstractQS` as described in section "3.1.3.5 PojoMessage and AtomicMessage".

#### 3.1.3.2 Concurrent access

Each push and poll of a message have to go through one single `Map` instance. During contention, this will put a lot of stress on the map. The map implementation and how access to the map is made thread-safe will have a great impact on the performance of the queue service.

Our benchmark setup will use exactly 8 queues (see section "3.2.3 QueueServiceBenchmark") but a varying number of threads that push and poll messages.

The queue contention will always be smaller than the contention over the `Map` since the number of queues is more than just one. The contention over individual queue instances will increase with an increased number of threads.

The queue service framework will benchmark the performance of many building blocks working together. Key performance drivers are (in no particular order): `Map` implementation, `Queue` implementation, synchronization mechanism (if needed) and the number of threads.

#### 3.1.3.3 Pushing a message

`AbstractQS.push(String)` will first create a new instance of the concrete message implementation.

Next, `push()` will use a map write access to call `Map.compute(Function)`, providing the map with a remapping function that push the message instance to a preexisting queue or if the queue does not exist, a new queue that is created inside the remapping function.

If the remapping function pushes to a preexisting queue, then this queue will be write-accessed. Access to a queue that the remapping function created will bypass any locking in place as the queue instance is not yet visible to other threads.

#### 3.1.3.4 Polling a message

Messages that are being processed by another consumer are collocated in the same queue instance as messages that have not yet been polled. Polling a message simply mark it as grabbed but its place in the queue remain. Each read attempt has to iterate through already polled messages, checking for their expiration and grab the first message that expired or has not yet been consumed.

In the first step of the implementation of `AbstractQS.pull()`, the map is read-accessed to get hold of the queue. If the queue exists, then the queue is read-accessed to iterate through messages starting with the head. Each message is attempted to be consumed by calling `AbstractMessage.tryGrab()`. First such call that succeeds break the iteration.

In the second step, if the implementation has a strong reason to believe the queue is empty, then an attempt to remove the queue from the map will be performed. This entails one write-access of the map and if the queue is still present, it will be confirmed empty by using a read-access.

#### 3.1.3.5 `PojoMessage` and `AtomicMessage`

`AbstractMessage.tryGrab()` must return a response which indicates success or failure. If failure, the response will also describe why the request was declined. This can either be because the message is completed or because the message is being actively processed by another consumer.

`tryGrab()` is invoked concurrently and the implementation will affect performance and may also affect delivery semantics. Two concrete message implementations are provided: `PojoMessage` and `AtomicMessage`.

`PojoMessage` has no thread synchronization mechanism in place. This is a performant implementation which suits queue service implementations that lock all access to the queue itself. Concurrent access to `PojoMessage` may cause duplicate deliveries.

`AtomicMessage` guards his state by compare- and set operations using one `AtomicReference`. `AtomicMessage` is thread-safe.

#### 3.1.3.6 Completing a message

The only thing `AbstractQS.complete()` does is to call `AbstractMessage.complete()`.

#### 3.1.3.7 Implementation: `SynchronizedQS`

`SynchronizedQS` is a queue service implementation that uses `HashMap`, `ArrayDeque` and `PojoMessage` as building blocks.

Read and write access to the map is serialized using Java's `synchronized` keyword. Since pushing and reading messages are serialized, individual queues need no synchronization.

This queue service implementation provides exactly-once delivery.

#### 3.1.3.8 Implementation: `ReadWriteLockedQS`

This implementation use `HashMap`, `ArrayDeque` and `PojoMessage`.

Concurrent readers are allowed, but only one writer may access the map and individual queues at any given moment in time. The lock implementation is `ReentrantReadWriteLock`.

This queue service delivers messages exactly-once.

#### 3.1.3.9 Implementation: `ConcurrentQSWithPojoMessage`

This implementation uses no application-controlled synchronization mechanism. Instead, the map and queues are thread-safe data structures provided by `java.util.concurrent.ConcurrentHashMap` and `ConcurrentLinkedDeque`.

The message implementation used is `PojoMessage` which during contention can lead to multiple deliveries. Hence, this setup only guarantees at-least-once delivery.

#### 3.1.3.10 Implementation: `ConcurrentQSWithAtomicMessage`

This implementation is almost the same as the one described previously, except that the message implementation is `AtomicMessage`. `AtomicMessage` is thread-safe and enables this queue service implementation to guarantee exactly-once delivery semantics.

# 3.2 Data Collection

## 3.2.1 Our setup

The `QueueServiceBenchmark` (application defined type) class file is a *harness* of `@Benchmark` annotated methods. Each benchmark method represents a workload that JMH benchmark/performance profile. Usually, these methods are very short and can be described as nano- or micro benchmarks.

The benchmark entry point is `StartJmh` (application defined type) and is launched using the Gradle task `bench` (see the "README.md" file).

## 3.2.2 JMH basics

By default, JMH starts a *benchmark*, or *trial*, in a new JVM.

When the benchmark runs, JMH will call the benchmark method over and over again during a fixed time-based duration. Each call to the method is called an *invocation*.

Run time expiration completes an *iteration*/observation and JMH proceeds to start a new iteration. Exactly how many iterations to run is configurable, by default it is 20.

All of these measurement iterations are preceded by *warmup* iterations whose data observations are ignored. By default, the number of warmup iterations are also 20 and these iterations are executed in the same JVM as the ensuing measurement iterations.

Optionally, many benchmark runs can be configured to execute in new JVM:s (*forks*) as described in section "3.1.2 Introduction to JMH".

Normally, one `@Benchmark` annotated method define one benchmark. Two or more such methods can be grouped into one benchmark using the `@Group` annotation. Grouped methods enable multithreaded benchmark runs to use a fixed schema for how to distribute execution threads amongst the methods.

Non-grouped benchmark methods can also execute in parallel using many threads, but there is only one method that all those threads are dispatched to. Non-grouped methods are many times referred to as *symmetric* tests and grouped methods are referred to as *asymmetric* tests.

JMH uses `State` objects which are useful to encapsulate input values to the benchmark code. For example, a queue service object reference.

The *scope* of a state object does not define the object's lifetime, but only how the state object is shared/accessed by many threads during the benchmark run. *Benchmark* scope make the state object be shared by all worker threads – only one instance of the state object is created. *Group* scope makes the state object be shared and unique per thread group – as many instances are created as there are thread groups. *Thread* scope makes the state object non-shared and unique per thread – as many instances of this object will be created as there are threads.

No scope survive the benchmark run. The instance is created on demand even during warmup iterations and survive throughout all warmup and measurement iterations. The instance is destroyed after the trial and will not be reused by other benchmarks running afterward. In fact, that is not technically possible if forking is enabled.

The persistence of the state object does not hinder the benchmark author from carefully setting up and using many invariants of the state object. *Fixture* methods annotated `@Setup` and `@TearDown` may be declared on the state object which will be called by JMH at specific points on the timeline, as defined by an annotation member value of type `Level`. There are three levels: Trial, iteration, and invocation. Time spent in these fixture methods does not participate in the profiled time cost of the benchmark.

### 3.2.3 `QueueServiceBenchmark`

`QueueServiceBenchmark` uses two grouped benchmark methods to separate between threads that *write* messages to the queue service and threads that *read* messages from the queue service. One message pushed or one message polled – one benchmark method invocation – equals one *operation*. It is the number of operations per microsecond that define the queue service throughput and is presented in chapter "4 Result".

Each thread will be strictly assigned to one of the two tasks. A writer thread will push messages as fast as he can for as long as the iteration runs. A reader thread works much the same way, except that the reader polls messages.

As many benchmarks will execute as the number of queue service implementations there are described in section "3.1.3 Queue service being utilized". Although technically speaking, the implementation type is a JMH parameter, the effect is one unique benchmark per implementation.

For each measurement iteration, a new instance of the queue service will be used.

Queue names are fixed and shared between writer and reader threads. This study uses 8 queues and therefore 8 queue names. Each thread will construct their queue access order out from a shared pool of names before each benchmark run. The order is randomized only once before the benchmark run and remains static throughout.

Each writer thread constructs exactly one message that is unique for the thread. Writer threads reuse the same message on each push.

The overhead of setting up the queue service instance, queue- names and messages are done outside of the measurement run and are not counted in the results. Time spent outside the queue service is kept to an absolute minimum.

### 3.2.4 Benchmark configurations

#### 3.2.4.1 Static parameters

We use 20 warmup iterations and 20 measurement iterations (JMH defaults). Each throughput benchmark iteration runs in 1 second (JMH default). We run each benchmark 10 times/forks (JMH default).

#### 3.2.4.2 Dynamic parameters

The one dynamic parameter that impact the benchmark is the number of concurrent readers and writers. This study will conduct the following benchmark setups:

| Name | Readers | Writers |
|---|---|---|
| Uncontended | 1 | 1 |
| Contended readers | 4 | 1 |
| Contended writers | 1 | 4 |
| Contended readers + writers | 4 | 4 |

Readers and writers run in parallel as it was described in the section "3.2.2 JMH basics". This makes the "uncontended" test not necessarily uncontended per se. The queue service is shared and will most likely lead to resource contention, just not as much compared to the other setups.

At most 8 threads execute. This number was chosen because the machine used to execute the benchmarks can only run 8 threads in parallel and the tasks are strictly CPU-bound (see chapter "1 Concurrency and parallelism" in "Extended background.pdf").

## 3.3 Data Analysis

`QueueServiceBenchmark` is abstract and declares all benchmark methods in concrete nested types. Each nested type declares two grouped benchmark methods, i.e., one benchmark. The only difference between these benchmarks is what measurement mode they are annotated to use. Other than the measurement mode, they perform the exact same workload as described in section "3.2.3 QueueServiceBenchmark".

The three benchmarks represent all of the supported modes; Throughput (operations per microsecond), average time (microseconds per operation) and single shot (microseconds per operation).

The difference between average time and single shot is that average measurement mode execute a benchmark given a fixed run duration versus single shot uses a fixed invocation count ("batch size").

This study will only execute and analyze throughput benchmarks (`QueueServiceBenchmark.Thrpt`).

# 4 Result

The specifics of the machine used to produce the results presented in this chapter is provided in "Appendix B: Machine specification". Most importantly, the CPU is a 4-core Intel® Core™ i7-4720HQ @ 2.60 Ghz using hyper-threading (8 logical processing units).

The tables presented in this chapter are ordered by benchmark first and descending score second.

Two tables for each benchmark setup is provided. The first table will give an overview of the scores and the second table will – by referencing back to the first table's row index (*Idx*) – look at the deviations.

The commands used to produce these results are provided in "Appendix C: Used commands".

In order to save space in the remainder of this document, the implementations have been aliased:

| Short name | Implementation |
|---|---|
| Synchronized | `SynchronizedQS` |
| ReadWriteLock | `ReadWriteLockedQS` |
| ConcurrentPojo | `ConcurrentQSWithPojoMessage` |
| ConcurrentAtomic | `ConcurrentQSWithAtomicMessage` |

## 4.1 Score legend

The *score* is the observed average throughput per benchmark (*BM*), measured in operations per microsecond. The score captures an average across all measurement runs (iterations) and JVM forks.

The *total* is an aggregation of reader and writer throughput. If one of the implementations score highest as both a reader and writer, then that implementation will win on total throughput as well. But the other way around is not necessarily true; just because someone is a clear winner looking at the total score doesn't necessarily mean he has outperformed all other implementations in both aspects.

The gain columns show the relative and accumulative score gains as percentages. The relative gain (*Rel. G.*) is the gain one implementation had over the next ranked implementation. The last implementation listed has no relative gain because there is no one else beneath him. The accumulative gain (*Acc. G.*) show each implementation's gain over the worst performing implementation.

For the *error* column, please see the next section where it is put in context.

Remarks that drive the analytical work in chapter "5 Concluding Remarks" are put beneath the score diagrams.

## 4.2 Deviation legend

Assuming normal distribution, the "deviations" table list a score range within a 99.9% confidence interval (*CI min* and *CI max*). The difference between the observed average and the lowest/highest score in the confidence interval is the *error* column in the first "score overview" table.

The deviations' table also lists the observed range (*Obs. min* and *Obs. max*) and standard deviation (*SD*).

The standard deviation has been visualized in box and whisker diagrams. Only the reader and writer standard deviations are included in the datasets. The 'R' and 'W' indicate standard deviation type for the lowest and highest values; reader or writer. The implementations are ordered from lowest average of the two to the highest average.

# 4.3 Test: Uncontended

## 4.3.1 Score overview

| Idx | BM | Implementation | Score (ops/us) | ± Error (CI) | Rel. G. | Acc. G. |
|---|---|---|---|---|---|---|
| T1 | Total | ConcurrentPojo | 12.671192 | 0.169216 | 15.35% | 192.07% |
| T2 | Total | ConcurrentAtomic | 10.984582 | 0.071411 | 36.64% | 153.20% |
| T3 | Total | Synchronized | 8.039285 | 0.245325 | 85.31% | 85.31% |
| T4 | Total | ReadWriteLock | 4.338382 | 0.065876 | | |
| R1 | Reader | Synchronized | 5.386474 | 0.374479 | 10.18% | 1337.31% |
| R2 | Reader | ConcurrentPojo | 4.888705 | 0.09864 | 30.02% | 1204.48% |
| R3 | Reader | ConcurrentAtomic | 3.760031 | 0.038093 | 903.31% | 903.31% |
| R4 | Reader | ReadWriteLock | 0.374762 | 0.010142 | | |
| W1 | Writer | ConcurrentPojo | 7.782487 | 0.080454 | 7.72% | 193.37% |
| W2 | Writer | ConcurrentAtomic | 7.224551 | 0.048484 | 82.27% | 172.34% |
| W3 | Writer | ReadWriteLock | 3.96362 | 0.069194 | 49.41% | 49.41% |
| W4 | Writer | Synchronized | 2.652811 | 0.172249 | | |

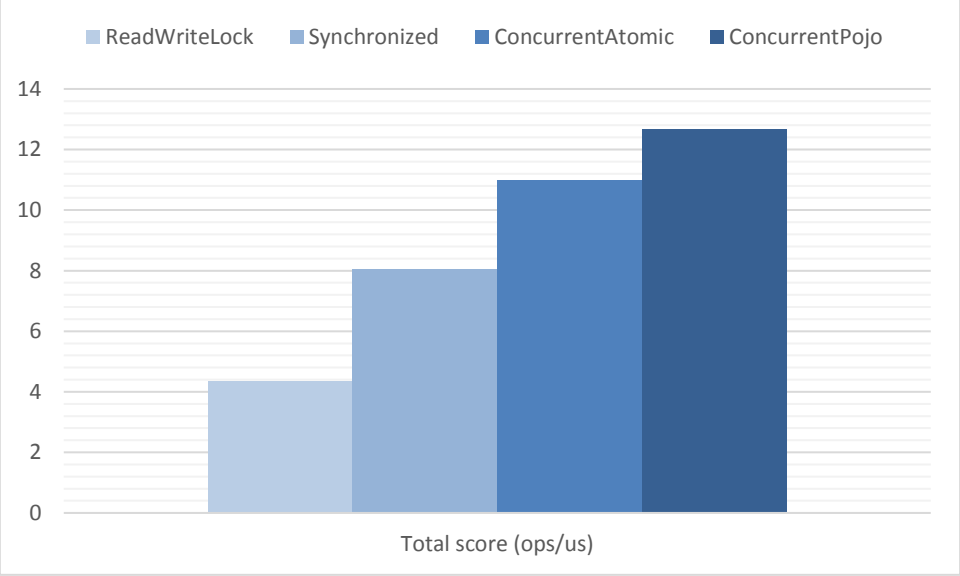


*Figure 1 Uncontended total score*

Looking at total throughput, ConcurrentAtomic outperformed ReadWriteLock with 153.2% (≈6.65 million of operations per second).

### 4.3.2 Deviations

| Idx | CI min | CI max | Obs. min | Obs. max | SD |
|---|---|---|---|---|---|
| T1 | 12.502 | 12.84 | 10.672 | 14.296 | 0.716 |
| T2 | 10.913 | 11.056 | 9.416 | 11.628 | 0.302 |
| T3 | 7.794 | 8.285 | 4.694 | 11.389 | 1.039 |
| T4 | 4.273 | 4.404 | 3.67 | 5.305 | 0.279 |
| R1 | 5.012 | 5.761 | 1.161 | 10.205 | 1.586 |
| R2 | 4.79 | 4.987 | 3.858 | 5.933 | 0.418 |
| R3 | 3.722 | 3.798 | 3.04 | 4.132 | 0.161 |
| R4 | 0.365 | 0.385 | 0.273 | 0.504 | 0.043 |
| W1 | 7.702 | 7.863 | 6.814 | 8.643 | 0.341 |
| W2 | 7.176 | 7.273 | 6.276 | 7.693 | 0.205 |
| W3 | 3.894 | 4.033 | 3.325 | 4.974 | 0.293 |
| W4 | 2.481 | 2.825 | 1.151 | 6.636 | 0.729 |

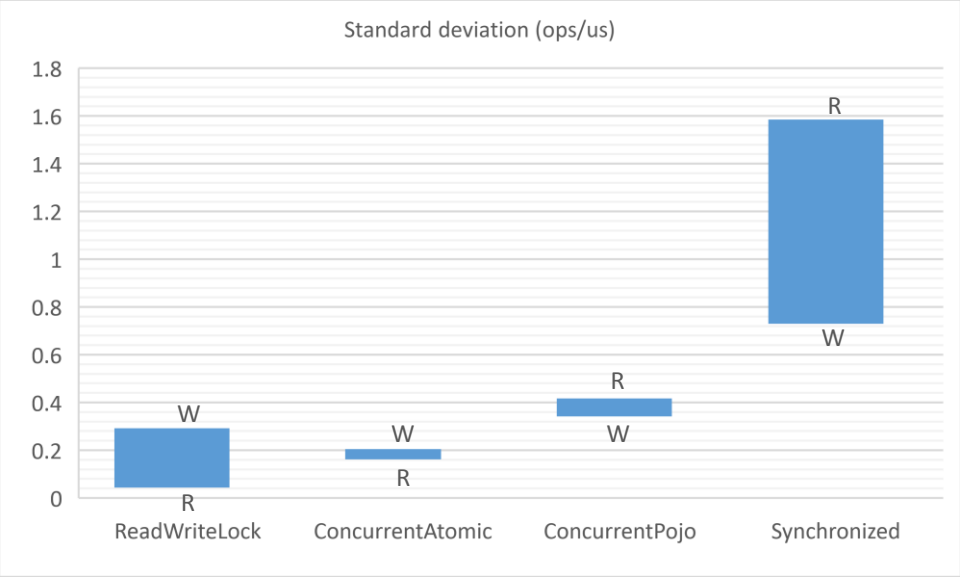


*Figure 2 Uncontended deviation*

Synchronized has the highest average standard deviation (1.1575 ops/us) and also the largest gap between reader- (1.586 ops/us) and writer (0.729 ops/us) deviations.

# 4.4 Test: Contended readers

## 4.4.1 Score overview

| Idx | BM | Implementation | Score (ops/us) | ± Error (CI) | Rel. G. | Acc. G. |
|---|---|---|---|---|---|---|
| T1 | Total | ConcurrentPojo | 37.924188 | 0.273196 | 33.04% | 850.84% |
| T2 | Total | ConcurrentAtomic | 28.50618 | 0.36902 | 106.24% | 614.71% |
| T3 | Total | Synchronized | 13.821948 | 0.200336 | 246.55% | 246.55% |
| T4 | Total | ReadWriteLock | 3.988484 | 0.056077 | | |
| R1 | Reader | ConcurrentPojo | 35.580107 | 0.275146 | 35.44% | 1585.02% |
| R2 | Reader | ConcurrentAtomic | 26.269988 | 0.372668 | 100.34% | 1144.11% |
| R3 | Reader | Synchronized | 13.112556 | 0.24413 | 520.99% | 520.99% |
| R4 | Reader | ReadWriteLock | 2.111552 | 0.022675 | | |
| W1 | Writer | ConcurrentPojo | 2.344081 | 0.009334 | 4.82% | 230.44% |
| W2 | Writer | ConcurrentAtomic | 2.236193 | 0.01605 | 19.14% | 215.23% |
| W3 | Writer | ReadWriteLock | 1.876932 | 0.034107 | 164.58% | 164.58% |
| W4 | Writer | Synchronized | 0.709392 | 0.078467 | | |

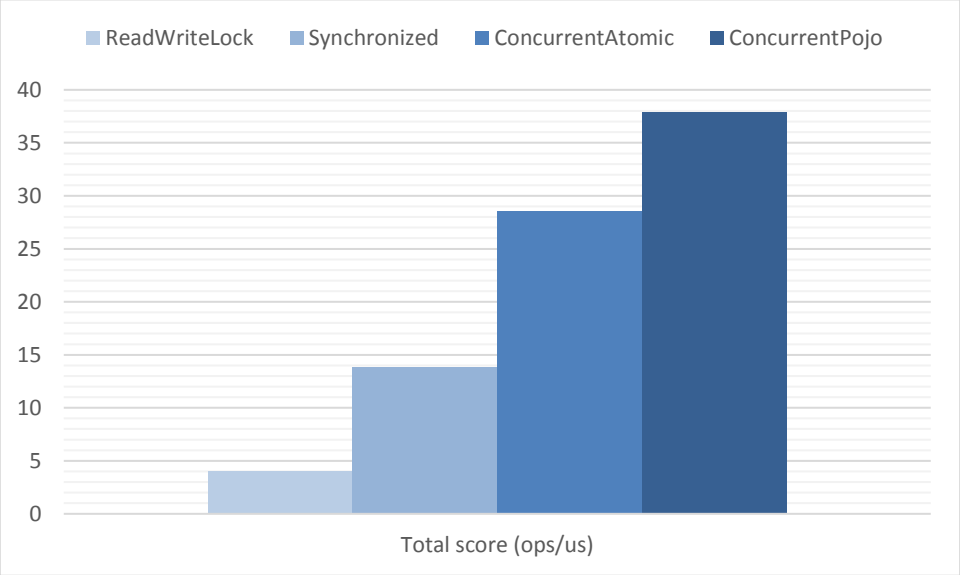


*Figure 3 Contended readers total score*

Of all the tests, "contended readers" show the largest total throughput gains ConcurrentAtomic had over Synchronized and ReadWriteLock. The relative gain over Synchronized is 106.24% (≈14.68M ops/s) and the accumulated gain over ReadWriteLock is a whopping 614.71% (≈24.52M ops/s).

Synchronized outperformed ReadWriteLock on total throughput in all tests. This test shows the largest relative gain of 246.55% (≈9.83M ops/s).

Synchronized also outperformed ReadWriteLock as a *reader* in all tests. This test shows the smallest relative gain of 520.99% (≈11M ops/s).

### 4.4.2 Deviations

| Idx | CI min | CI max | Obs. min | Obs. max | SD |
|---|---|---|---|---|---|
| T1 | 37.651 | 38.197 | 35.22 | 41.197 | 1.157 |
| T2 | 28.137 | 28.875 | 24.17 | 31.2 | 1.562 |
| T3 | 13.622 | 14.022 | 11.234 | 16.516 | 0.848 |
| T4 | 3.932 | 4.045 | 3.192 | 4.308 | 0.237 |
| R1 | 35.305 | 35.855 | 32.868 | 38.868 | 1.165 |
| R2 | 25.897 | 26.643 | 21.904 | 28.92 | 1.578 |
| R3 | 12.868 | 13.357 | 9.386 | 16.022 | 1.034 |
| R4 | 2.089 | 2.134 | 1.821 | 2.292 | 0.096 |
| W1 | 2.335 | 2.353 | 2.211 | 2.441 | 0.04 |
| W2 | 2.22 | 2.252 | 2.009 | 2.352 | 0.068 |
| W3 | 1.843 | 1.911 | 1.37 | 2.057 | 0.144 |
| W4 | 0.631 | 0.788 | 0.177 | 2.286 | 0.332 |

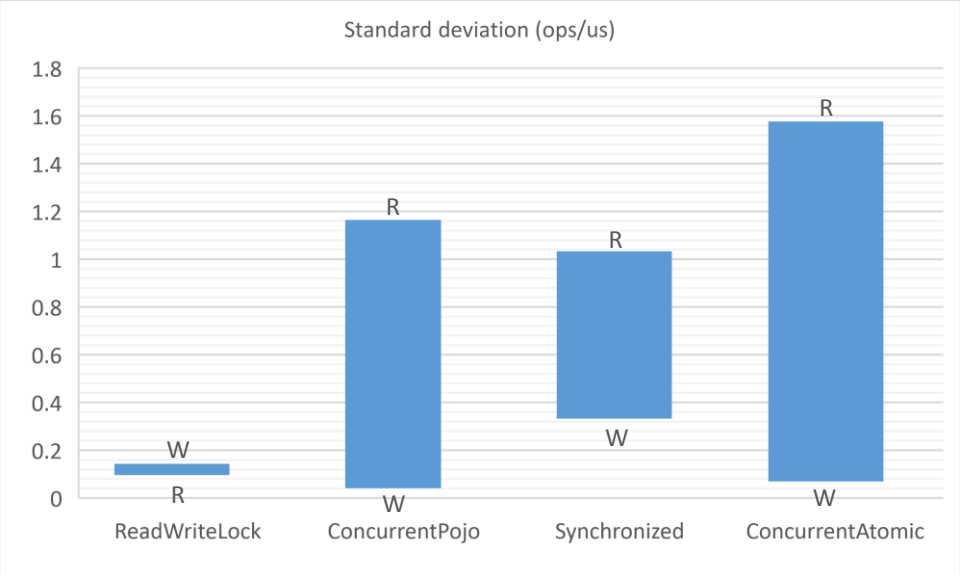


*Figure 4 Contended readers deviation*

ConcurrentAtomic has the highest average standard deviation (0.823 ops/us) and also the largest gap between reader- (1.578 ops/us) and writer (0.068 ops/us) deviations.

# 4.5 Test: Contended writers

## 4.5.1 Score overview

| Idx | BM | Implementation | Score (ops/us) | ± Error (CI) | Rel. G. | Acc. G. |
|---|---|---|---|---|---|---|
| T1 | Total | ConcurrentPojo | 11.90557 | 0.141463 | 13.24% | 96.66% |
| T2 | Total | ConcurrentAtomic | 10.513829 | 0.081731 | 26.82% | 73.67% |
| T3 | Total | Synchronized | 8.290536 | 0.28363 | 36.95% | 36.95% |
| T4 | Total | ReadWriteLock | 6.053754 | 0.124646 | | |
| R1 | Reader | Synchronized | 5.234507 | 0.420744 | 38.11% | 11924.78% |
| R2 | Reader | ConcurrentPojo | 3.790232 | 0.071606 | 27.68% | 8606.97% |
| R3 | Reader | ConcurrentAtomic | 2.968489 | 0.028723 | 6719.25% | 6719.25% |
| R4 | Reader | ReadWriteLock | 0.043531 | 0.00024 | | |
| W1 | Writer | ConcurrentPojo | 8.115338 | 0.078732 | 7.55% | 165.55% |
| W2 | Writer | ConcurrentAtomic | 7.54534 | 0.058251 | 25.54% | 146.90% |
| W3 | Writer | ReadWriteLock | 6.010223 | 0.124658 | 96.67% | 96.67% |
| W4 | Writer | Synchronized | 3.056029 | 0.187069 | | |

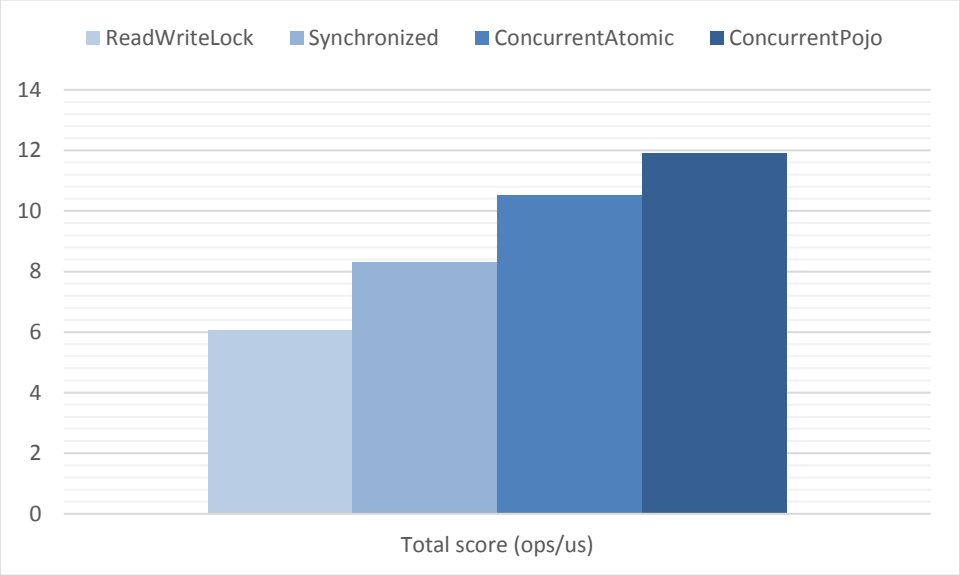


*Figure 5 Contended writers total score*

Of all the test, “contended writers” show the smallest total throughput gains ConcurrentAtomic had over Synchronized and ReadWriteLock. The relative gain over Synchronized was 26.82% (≈2.22M ops/s) and the accumulated gain over ReadWriteLock was 73.67% (≈4.46M ops/s).

This test shows the smallest relative gain Synchronized had over ReadWriteLock on total throughput: 36.95% (≈2.24M ops/s). It also shows the largest gain Synchronized had as a *reader* over ReadWriteLock with a massive 11 924.78% (≈5.19M ops/s).

### 4.5.2 Deviations

| Idx | CI min | CI max | Obs. min | Obs. max | SD |
|---|---|---|---|---|---|
| T1 | 11.764 | 12.047 | 8.828 | 12.972 | 0.599 |
| T2 | 10.432 | 10.596 | 9.027 | 11.236 | 0.346 |
| T3 | 8.007 | 8.574 | 6.066 | 12.094 | 1.201 |
| T4 | 5.929 | 6.178 | 4.483 | 6.924 | 0.528 |
| R1 | 4.814 | 5.655 | 1.222 | 10.96 | 1.781 |
| R2 | 3.719 | 3.862 | 2.61 | 4.372 | 0.303 |
| R3 | 2.94 | 2.997 | 2.312 | 3.279 | 0.122 |
| R4 | 0.043 | 0.044 | 0.039 | 0.047 | 0.001 |
| W1 | 8.037 | 8.194 | 6.218 | 8.702 | 0.333 |
| W2 | 7.487 | 7.604 | 6.638 | 8.103 | 0.247 |
| W3 | 5.886 | 6.135 | 4.438 | 6.88 | 0.528 |
| W4 | 2.869 | 3.243 | 1.134 | 6.517 | 0.792 |

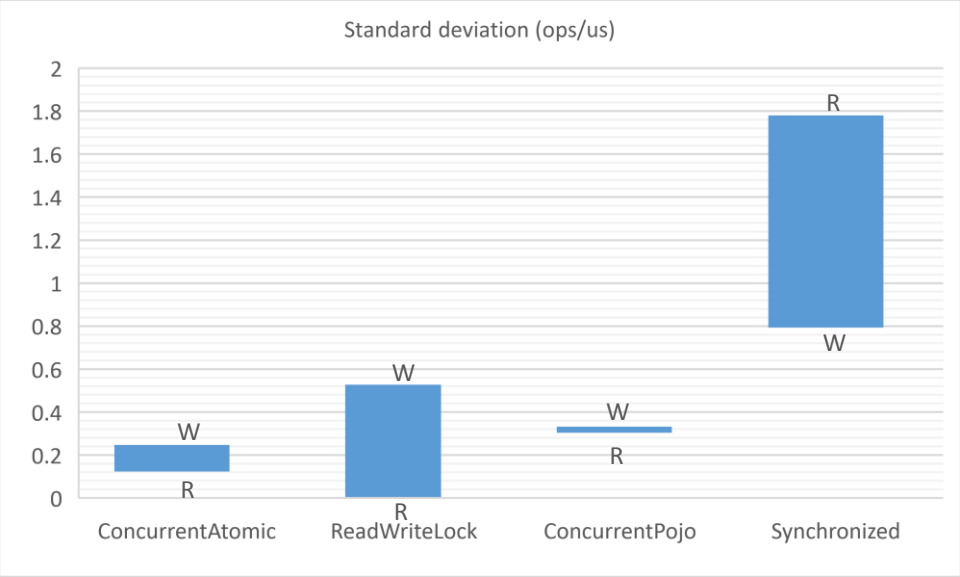


*Figure 6 Contended writers deviation*

Synchronized has the highest average standard deviation (1.2865 ops/us) and also the largest gap between reader- (1.781 ops/us) and writer (0.792 ops/us) deviations.

# 4.6 Test: Contended readers + writers

## 4.6.1 Score overview

| Idx | BM | Implementation | Score (ops/us) | ± Error (CI) | Rel. G. | Acc. G. |
|---|---|---|---|---|---|---|
| T1 | Total | ConcurrentPojo | 17.880986 | 0.075485 | 20.69% | 240.27% |
| T2 | Total | ConcurrentAtomic | 14.815837 | 0.076166 | 38.10% | 181.95% |
| T3 | Total | Synchronized | 10.728233 | 0.23574 | 104.16% | 104.16% |
| T4 | Total | ReadWriteLock | 5.254866 | 0.112334 | | |
| R1 | Reader | ConcurrentPojo | 11.100944 | 0.112325 | 27.86% | 873.76% |
| R2 | Reader | Synchronized | 8.681852 | 0.356268 | 26.34% | 661.56% |
| R3 | Reader | ConcurrentAtomic | 6.872022 | 0.042126 | 502.80% | 502.80% |
| R4 | Reader | ReadWriteLock | 1.140012 | 0.030581 | | |
| W1 | Writer | ConcurrentAtomic | 7.943815 | 0.044591 | 17.16% | 288.19% |
| W2 | Writer | ConcurrentPojo | 6.780042 | 0.066621 | 64.77% | 231.32% |
| W3 | Writer | ReadWriteLock | 4.114854 | 0.101712 | 101.08% | 101.08% |
| W4 | Writer | Synchronized | 2.046381 | 0.12907 | | |

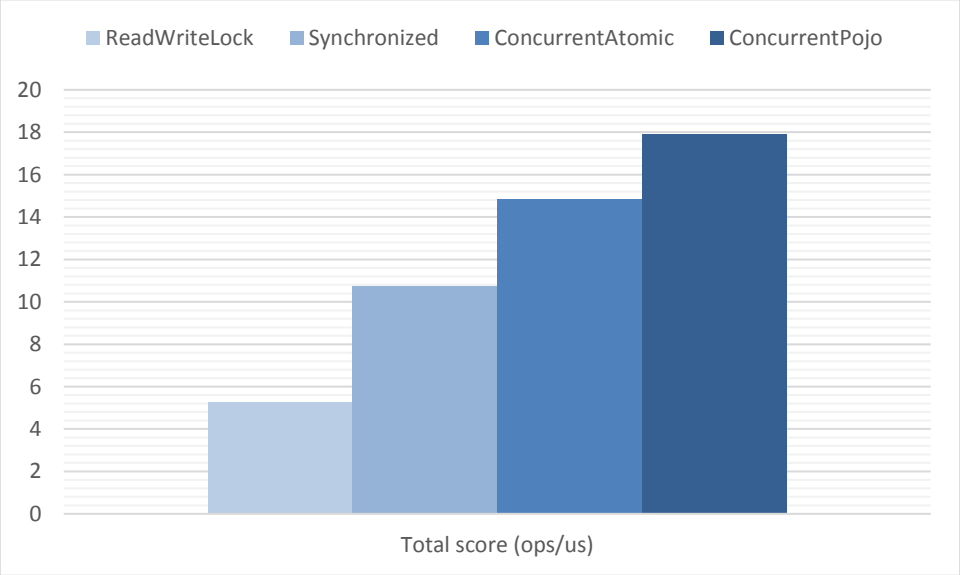


*Figure 7 Contended readers + writers total score*

ConcurrentAtomic outperformed ReadWriteLock on total throughput with 181.95% (≈9.56M ops/s).

ConcurrentAtomic also beat ConcurrentPojo as a *writer* with 17.16% (≈1.16M ops/s) relative gain. In all other tests and aspects, ConcurrentPojo had relative gains over ConcurrentAtomic ranging from 4.82% (≈0.11M ops/s – as a writer in contended readers) to 35.44% (≈9.31M ops/s – as a reader in contended readers).

### 4.6.2 Deviations

| Idx | CI min | CI max | Obs. min | Obs. max | SD |
|---|---|---|---|---|---|
| T1 | 17.806 | 17.956 | 16.933 | 18.57 | 0.32 |
| T2 | 14.74 | 14.892 | 13.176 | 15.32 | 0.322 |
| T3 | 10.492 | 10.964 | 8.395 | 13.858 | 0.998 |
| T4 | 5.143 | 5.367 | 4.089 | 6.115 | 0.476 |
| R1 | 10.989 | 11.213 | 8.662 | 11.823 | 0.476 |
| R2 | 8.326 | 9.038 | 4.867 | 12.392 | 1.508 |
| R3 | 6.83 | 6.914 | 6.052 | 7.262 | 0.178 |
| R4 | 1.109 | 1.171 | 0.631 | 1.332 | 0.129 |
| W1 | 7.899 | 7.988 | 7.11 | 8.303 | 0.189 |
| W2 | 6.713 | 6.847 | 6.105 | 8.507 | 0.282 |
| W3 | 4.013 | 4.217 | 3.16 | 4.875 | 0.431 |
| W4 | 1.917 | 2.175 | 0.744 | 3.528 | 0.546 |

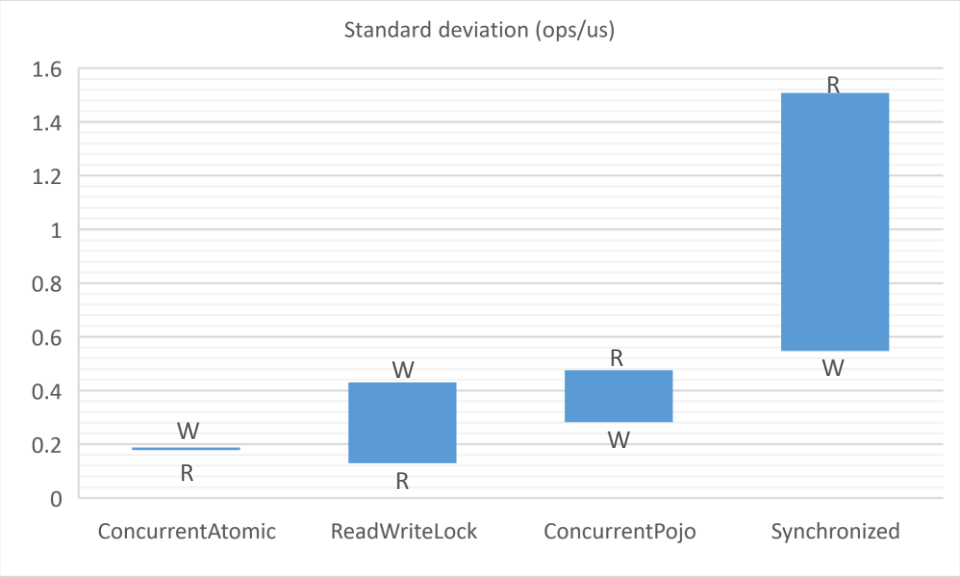


*Figure 8 Contended readers + writers deviation*

For the contended readers and writers test, Synchronized has the highest average standard deviation (1.027 ops/us) and also the largest gap between reader- (1.508 ops/us) and writer (0.546 ops/us) deviations.

# 5 Concluding Remarks

*This chapter uses implementation aliases as listed in the opening of chapter ”4 Result”.*

## 5.1 Conclusion

### 5.1.1 Ranking

Looking at total throughput, the ranking does not change no matter how we changed the thread count. From the best performer to the worst performer, the order stays the same:

1. ConcurrentPojo
2. ConcurrentAtomic
3. Synchronized
4. ReadWriteLock

It was expected that concurrent data structures provided by the JDK outperform explicit locking and thread management. What was not expected is that `synchronized` greatly outperform `ReentrantReadWriteLock`.

Choosing the right implementation is certainly important. ConcurrentAtomic and ReadWriteLock which both yields the same delivery semantics for the queue service – exactly-once – differed greatly.

Synchronized was always the worst performing writer, but many times one of the best performing readers. So much so that it consistently beat ReadWriteLock on total throughput. Synchronized as a reader even beat the concurrent data structures in the “uncontended” and “contended writers” test, or to put it differently; whenever reader contention was low.

Since we know that synchronized does not make any difference between reader and writer threads, this finding seems to imply that the difference in the outcome should be analyzed with the other implementations in mind. Their possible lack of read or write performance will make Synchronized look comparably better.

None of the implementations specifies if writers are prioritized before readers. `ReentrantReadWriteLock` could have been constructed using a fair policy which does specify a preference for writer threads. But this study did not use a fair policy.

Since we found that Synchronized only fell behind as a reader when reader contention was increased, it does seem to imply that the other implementations, even though it is unspecified, has a preference for writer threads making their readers suffer. They did catch up when readers were increased since Synchronized continued to serialize all reader access and is unbenefited from the increase of reader contention.

### 5.1.2 Concurrent data structures

Concurrent data structures compared to ReadWriteLock and Synchronized, consistently won the race on total and writer throughput. Only in two tests, as listed in the previous section, did Synchronized yield a better reader throughput.

The only difference between ConcurrentPojo and ConcurrentAtomic is which message implementation they use. `PojoMessage` is not thread-safe, `AtomicMessage` is thread-safe by encapsulating all state mutations as compare- and set operations on an `AtomicReference`.

Synchronization will always bring with it an overhead cost, and it was therefore expected that ConcurrentPojo would beat ConcurrentAtomic in all tests. This was almost the case; the one exception being the contended readers and writers test.

Why did ConcurrentAtomic beat ConcurrentPojo as a writer during high contention? It is hard to reason about this. The only difference between these two implementations during the pushing of a new message is the time it takes to construct a new message instance. `AtomicMessage` adds on top of `AbstractMessage` only one instance field that is eagerly initialized to a new instance of `AtomicReference`. `PojoMessage` add two instance fields, one `Instant` and one `boolean`. The `Instant` field is initialized to `null` and the `boolean` is initialized to `false`. This study will leave the question open to the reader.

ConcurrentPojo can not promise exactly-once delivery semantics. ConcurrentAtomic can. If the consumers are idempotent (can process the same message many times without changing the result), then ConcurrentPojo may be an attractive choice. This author would be hesitant to add complexity and confidence in consumer compliance for a smallish two digit gain; 20.69% (≈3.07M ops/s) for contended readers and writers. One would also have to estimate, or even better – benchmark – and factor in the potential performance cost that all consumers might have to pay in order to become idempotent.

### 5.1.3 `ReentrantReadWriteLock` versus `synchronized`

ReadWriteLock beat Synchronized in all tests judging by the write throughput. But never enough to win on total throughput. Synchronized outperformed ReadWriteLock consistently on both total- and reader throughput.

This is quite a remarkable finding. ReadWriteLock has the ability to allow concurrent readers. Synchronized lack this ability. Yet, Synchronized had the most gains over ReadWriteLock as a reader. One explanation could be that the read operations was not long enough to warrant the separation of readers and writer. JavaDoc of `ReadWriteLock` writes [23]:

> Further, if the read operations are too short the overhead of the read-write lock implementation (which is inherently more complex than a mutual exclusion lock) can dominate the execution cost, particularly as many read-write lock implementations still serialize all threads through a small section of code. Ultimately, only profiling and measurement will establish whether the use of a read-write lock is suitable for your application.

### 5.1.4 Variability

A small standard deviation increases the reliability of the benchmark and may help queue service providers to predict and estimate service level agreements.

Unlike score rating, there is no consistent ranking to be found for variability.

Synchronized is overall a pretty lousy pick in terms of small deviations. ReadWriteLock manages to stay in the top two throughout all tests. ConcurrentPojo is mostly a worse off pick than ConcurrentAtomic. For contended readers and writers, ConcurrentAtomic is a clear winner.

It is true for all tests that ReadWriteLock had a smaller average deviation as well as a smaller range of deviations than Synchronized. Also true for all tests is that the reader threads of ReadWriteLock had

the smallest deviations compared with the writer threads of ReadWriteLock. For Synchronized, the inverse is true.

# 5.2 Discussion and Future Research

## 5.2.1 Benchmark parameters

There are a lot of parameters, or factors, that could change and very likely affect performance. More research could be put in to see which factor has the greatest effect. For example, the number of queues, cores, and threads.

The benchmarks executed in this study uses a message timeout of 999 days (see `QSImpl`). This effectively takes timeouts out of the picture. Reducing this timeout would barely make much of a difference since the reader threads in our benchmarks immediately mark the message completed as soon as it has been polled. Realistic message processing and timeouts are something that could be added.

## 5.2.2 Modifying the queue service framework

### 5.2.2.1 Using a fair policy for `ReentrantReadWriteLock`

`ReentrantReadWriteLock` could be setup to use a fair policy/mode. This will affect how `ReentrantReadWriteLock` prioritizes threads. This change would require modification of only a few lines of code in the constructor of `ReadWriteLockedQS`. From this:

```
public ReadWriteLockedQS(Duration timeout) {
    super(message(PojoMessage::new).
          timeout(timeout).
          map(readWrite(new HashMap<>(), new ReentrantReadWriteLock())).
          queue(readWrite(ArrayDeque::new, ReentrantReadWriteLock::new)));
}
```

To this:

```
public ReadWriteLockedQS(Duration timeout) {
 super(message(PojoMessage::new).
       timeout(timeout).
       map(readWrite(new HashMap<>(), new ReentrantReadWriteLock(true))).
       queue(readWrite(ArrayDeque::new, () -> new ReentrantReadWriteLock(true))));
}
```

### 5.2.2.2 Using the same data structures

This study did not use the same map and queue implementations across all queue services. To some degree, measured performance differences between the queue service implementations that use concurrent data structures (ConcurrentQS***) versus the queue services that use explicit locking (`SynchronizedQS`, `ReadWriteLockedQS`) must be attributed to performance differences between the map and queue implementations used.

The motivation behind the decision of using separate backing data structures is that we seek to benchmark a real-world component. In the real world, a queue service wouldn't need explicit locking if the backing data structures already were thread-safe. The reader of this study will get a better picture

of the performance difference going from explicit locking to concurrent data structures provided by the JDK.

It is easy to configure all queue services to use the same data structures, and if so, the only difference left between the queue services would strictly be how thread-safety is accomplished.

As with the reconfiguration of `ReadWriteLockedQS` described in the previous section, changing the implementations are very easy. Here is an example of a reconfigured constructor of `SynchronizedQS`:

```
public SynchronizedQS(Duration timeout) {
    super(message(PojoMessage::new).
          timeout(timeout).
          map(mutex(new ConcurrentHashMap<>())).
          queue(noLock(ConcurrentLinkedDeque::new)));
}
```

## 5.2.3 Alternative frameworks

### 5.2.3.1 Reduce the number of iterations

An easy way to estimate performance differences between two algorithms that accomplish the same result is to study how many elements they need to look at. For example, *linear* search has an average performance cost of O(n) where n equals the number of elements in the data structure to be searched [24]. *Binary* search assumes that the data structure is sorted and can eliminate half of the entire data structure on each failed attempt, reducing the time cost to O(log n) [25].

`AbstractQS` collocate messages grabbed for consumption, but not yet completed or purged, in the same queue as messages that has not been consumed. When a message is taken by a consumer, the message is only marked as being processing but does not get removed from the queue. This makes each new poll of a message potentially iterate through a bunch of messages for no real benefit. An alternative framework could move messages under processing to a separate queue. New polls that want to check for a processing expiration would only need to examine the head of the separate queue.

### 5.2.3.2 Eager message eviction

`AbstractQS` uses lazy eviction. With lazy eviction enabled, a message marked completed has no effect on the map and queue. Instead, reader threads during message iteration will delete completed messages.

With eager eviction, each completion of a message will eagerly get removed from the queue and also attempt to remove empty queues.

Which mode to use is controlled by a static boolean flag: `AbstractQS#LAZY_EVICTION`. The benchmarks have been executed with this flag set to `true`.

Please note that turning off lazy eviction will change the delivery semantics of `ReadWriteLockedQS` from exactly-once to at-least-once since `ReadWriteLockedQS` currently use `PojoMessage` and this message class will be accessed concurrently if eviction is eager (see the implementation of `AbstractQS.poll()`).

### 5.2.3.3 Assume that two reads are faster than one write

Each call to `AbstractQS.push()` asks for a write-access of the underlying map regardless if a new queue had to be put into the map or not.

*If* it is true that most calls get routed to a queue that already exists and *if* it is true that two read operations on the map plus the occasional read and write – is faster than always performing one write, then one could optimize the `push()` implementation by using a write access to the map only if the queue didn't exist.

In code, we could accomplish this in two steps. 1) Take the current implementation of `push0()` and move it to another method, say: `writePush()`. 2) Rewrite `push0()` to something like the following:

```
Lockable<Queue<M>> lq = c.map().readGet(map -> map.get(message.queue()));

if (lq == null) {
    writePush(message);
    return;
}

lq.write(q -> q.add(message));

if (lq != c.map().readGet(map -> map.get(message.queue()))) {
    // Queue instance is no longer in the map, someone removed or replaced him.
    writePush(message);
}
```

This author tried this hack. However, even before reaching benchmarking, `AbstractQSTest.test_at_least_once()` showed that `ReadWriteLockedQS` suffered greatly. This concurrency test went from taking about 5-6 seconds on the author's machine (which is about the same for all implementations, although sometimes `ReadWriteLockedQS` take much longer) to several minutes. Turning off lazy eviction reduced this time cost to about 20-22 seconds. But still, the punishment was severe and it is a hint that the read locks of `ReentrantReadWriteLock` are underperforming significantly. This might have an explanation as described in section "5.1.3 ReentrantReadWriteLock versus synchronized".

These assumptions about `ReentrantReadWriteLock` for this particular test method and queue service framework is not scientifically backed nor do we desire to utilize a framework that is unfairly biased against only one of the implementations. Hence, this hack was not investigated further and certainly did not make it into production. But it does serve as a reminder that 1) the framework (or queue service solution) does have a big impact. 2) The queue service solution might need to be tailored for the thread-safety mechanism in place – or, the other way around. 3) These mechanisms are not as exchangeable with each other as we might first have thought! Just one minor tweak and all of a sudden just one of them drop faster than a speeding bullet.

# 5.3 Relation to Other Studies

No research, academic/scientific or otherwise, has been found that compares to this study.

Some research implies that `synchronized` is a better choice than `Lock` for uncontended code blocks.

[26]:

> The synchronized mechanism is optimized for the uncontended case […].

[9, p. 295]:

> […] the primary performance goal [for java.util.concurrent] is scalability: to predictably maintain efficiency even, or especially, when synchronizers are contended.

[9, p. 306] did find that on x86-based machines, `synchronized` was faster than `ReentrantReadWriteLock` but fell behind as contention grew. [27] notes that it "appears that Lock performs best with high numbers of threads".

Our study had a much wider angle and measured two aspects; reading and writing. As such, it is not comparable to the aforementioned studies. However, in line with these other research results, our study did find that Synchronized beat all other implementations whenever reader contention was low. In opposition to the other studies, our benchmarking showed that `ReadWriteLock` had a gain over Synchronized only as a writer, but not enough to win on *total* throughput – no matter the level of contention.

If anything, the results of this study have stressed the importance of benchmarking real-world application code.

# 5.4 Research Quality

## 5.4.1 Reproducibility

All queue service implementations and benchmarks are provided in a source code repository [14].

The details of the machine used to run the benchmarks and produce the results presented in this report is provided in "Appendix B: Machine specification".

All running applications were killed before running the benchmark. The network interface card and OS timers such as turning off disks and monitors after a certain amount of time were disabled.

## 5.4.2 Validity

Despite the promise of a real-world application and/or component, the benchmark setup is not "real-world". Only in a real-world application would the validity be complete. For example, no time at all was spent by the worker threads to produce or consume messages. The absolute majority of the time was spent in the queue service.

We did not look at CPU or memory consumption which most certainly are very important aspects of "performance". A comparatively high resource consumption may render one implementation less favorable despite having a higher throughput.

## 5.4.3 Reliability

### 5.4.3.1 Statistically rigorous evaluation

[28, p. 58] writes:

> In this paper, we argue that […] not using a statistically rigorous data analysis can be misleading and can even lead to incorrect conclusions.

[28] make no reference to JMH – which was first tagged in the source code repository 6 years after [29] the publication of [28]. Still, in order to produce statistically rigorous data, [28, pp. 58-59] advocates a number of methods, all of which, JMH complies with. For example, make many measurement iterations across many forks to account for run-to-run variance. A key point is:

> [...] each of these measurements, collect performance numbers for different iterations once performance reaches steady-state, i.e., after the start-up phase[.]

What these authors describe as the start-up phase is a phase during which the JVM has not completed a bunch of optimizations, i.e., before the runtime environment has reached a steady-state.

[28] does not mention "warmup iterations" and it appears that the authors – without giving alternatives – simply measure all iterations and then apply post-production math in an honest attempt at rolling back the damage done.

This paper favors the JMH style of running an equally large number of warmup iterations as measurement iterations and does not account the observations done during warmup.

[28, p. 62] consider 30 or more observations to be a large number of measurements. The queue service benchmarks gave rise to 200 observations (10 forks times 20 measurement iterations).

#### 5.4.3.2 Benchmarking non-steady state

The queue service implementations that are being benchmarked has non-steady states; run time length may impact the performance. Arguments can be made that the single shot measurement mode is a better fit than the used throughput mode. This author concludes that for the benchmarks this study run, throughput produce more reliable results. It is discussed further in "Appendix D: Benchmarking non-steady state".

#### 5.4.3.3 Correctness of the queue service

The correctness of the source code produced for this study is assumed to be correct.

There are tools out there that are used to test concurrent code – none of which was employed by this study. CovCon was cited in chapter "5 Testing concurrent code" in "Extended background.pdf". This is not to say that the author of this study find CovCon to be reliable. A quick look at CovCon's source code reveal many troubling details; for example, using a `StringBuffer` visible to only 1 thread instead of a `StringBuilder` [30]. No one can make a benighted mistake like this and at the same time claim to be a concurrency expert.

The only point this author wish to make is that anyone can make and publicize a "tool". Just because there is a tool out there, does not guarantee using it have beneficial results for your application. This author has a personal history to read and assert the reliability of a third-party tool's source code before embedding it.

### 5.4.4 Generalizability

The generalizability is rather low. Of all the factors that play a role, we have only changed one: number of threads. Furthermore, the results are specific to the machine used to execute the tests.

### 5.4.5 Extensibility

The design of the framework support extensibility quite well. Please see sections "3.1.3 Queue service being utilized" and "5.2.2 Modifying the queue service framework".

### 5.4.6 Credibility

No other party than the author of this study testifies that the results presented were the actual results achieved.

## 5.5 Ethical and Social Consequences

No animals were harmed in the making of this study.

# Scientific References


[2] Niraj, T., Andersen, D., and Satyanarayanan, M. “Quantifying Interactive User Experience on Thin Clients”, IEEE Comput., Comput., Washington, D.C., vol. 39, issue 3, p. 47, March, 2006

[5] Bergman, P., and Nilsson, F. “Software Transactional Memory”, bachelor's thesis, Dept. of Comput. and Syst. Sci., Stockholm University, Stockholm, Sweden, June, 2015

[6] Sutter, H. "The free lunch is over: A fundamental turn toward concurrency in software", Dr. Dobb's Journal, M & T Pub., vol. 30, no. 3, pp. 202-210, March 22, 2005

[7] Pinto, G., Torres, W., Fernandes, B., Castor, F., and Barros, R. “A Large-Scale Study on the Usage of Java's Concurrent Programming Constructs”, Journal of Syst. and Software, Amsterdam, Netherlands, vol. 106, pp. 59-81, August, 2015

[9] Lea, D. “The java.util.concurrent synchronizer framework”, Sci. of Comput. Programming, Amsterdam, Netherlands, vol. 58, issue 3, pp. 293-309, December, 2005

[11] Leitner P., Bezemer C. "An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects", PeerJ Preprints 4:e2496v2, October 6, 2016

[12] Denscombe, M. “The Good Research Guide”, 4th ed. Glasgow, Great Britain: Open University Press, 2010.

[26] Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. "Java Concurrency in Practice", 1st ed. Addison-Wesley Professional, p. 230, May 19, 2006

[28] Georges, A., Buytaert, D. and Eeckhout, L. "Statistically Rigorous Java Performance Evaluation", ACM SIGPLAN Notices - Proceedings of the 2007 OOPSLA, ACM, New York, NY, USA, vol. 42, issue 10, pp. 57-76, October, 2007

[33] Lea, D. “A Java Fork/Join Framework”, Proceedings of the ACM 2000 conference on Java Grande, ACM, New York, NY, USA, pp. 36-43, June 3, 2000

[34] Choudhary, A., Lu S., and Pradel M. "Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests", ICSE 2017, 2017

# Web References


[1] Wikipedia (2017, April 5). Java version history [online] Available: https://en.wikipedia.org/wiki/Java_version_history#J2SE_5.0 [2017, April 10]

[3] Linden, G. (2016, November 9). Marissa Mayer at Web 2.0 [online] Available: http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html [2017, May 13]

[4] Garrett, O. (2015, June). Inside NGINX: How We Designed for Performance & Scale [online] Available: https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/ [2017, April 10]

[8] Oracle. Frequently Asked Questions About the Java HotSpot VM [online] Available: http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#benchmarking_recommend [2017, April 10]

[10] Oracle. The Java HotSpot Performance Engine Architecture [online] Available: http://www.oracle.com/technetwork/java/whitepaper-135217.html#ultra [2017, April 9]

[13] Code Tools: jmh [online] Available: http://openjdk.java.net/projects/code-tools/jmh/ [2017, April 10]

[14] Andersson, M. (2017, March). Exploring Java concurrency performance [online] Available: https://github.com/martinanderssondotcom/queue-service-benchmark [2017, April 12]

[15] Wikipedia (2017, March 5). Arithmetic mean [online] Available: https://en.wikipedia.org/wiki/Arithmetic_mean [2017, April 10]

[16] Robert Johnson Foundation. Constant Comparative Method - Grounded Theory [online] Available: http://www.qualres.org/HomeCons-3824.html [2017, April 10]

[17] Shade (2017, March 28). code-tools/jmh: 36a2ee9a075e jmh-samples […] [online] Available: http://hg.openjdk.java.net/code-tools/jmh/file/36a2ee9a075e/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_08_DeadCode.java [2017, May 14]

[18] Shade (2017, March 28). code-tools/jmh: 36a2ee9a075e jmh-samples […] [online] Available: http://hg.openjdk.java.net/code-tools/jmh/file/36a2ee9a075e/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_10_ConstantFold.java [2017, May 14]

[19] Shade (2017, March 28). code-tools/jmh: 36a2ee9a075e jmh-samples […] [online] Available: http://hg.openjdk.java.net/code-tools/jmh/file/36a2ee9a075e/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_11_Loops.java [2017, May 14]

[20] Shade (2017, March 28). code-tools/jmh: 36a2ee9a075e jmh-samples […] [online] Available: http://hg.openjdk.java.net/code-tools/jmh/file/36a2ee9a075e/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_22_FalseSharing.java [2017, May 14]

[21] Shade (2017, March 28). code-tools/jmh: 36a2ee9a075e jmh-samples […] [online] Available: http://hg.openjdk.java.net/code-tools/jmh/file/36a2ee9a075e/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_12_Forking.java [2017, May 14]

[22] Shade (2017, March 28). code-tools/jmh: 36a2ee9a075e jmh-samples […] [online] Available: http://hg.openjdk.java.net/code-tools/jmh/file/36a2ee9a075e/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_13_RunToRun.java [2017, May 14]

[23] Lea, D. (2016, January 12). ReadWriteLock (Java Platform SE 8) [online] Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReadWriteLock.html [2017, May 21]

[24] Wikipedia (2017, May 19). Linear search [online] Available: https://en.wikipedia.org/wiki/Linear_search [2017, May 21]

[25] Wikipedia (2017, May 18). Binary search algorithm [online] Available: https://en.wikipedia.org/wiki/Binary_search_algorithm [2017, May 21]

[27] Lawrey P. (2011, July 30). Synchronized vs Lock performance [online] Available: http://vanillajava.blogspot.com/2011/07/synchronized-vs-lock-performance.html [2017, May 21]

[29] Shade (2013, November 20). code-tools/jmh: 1e59f0d249dc [online] Available: http://hg.openjdk.java.net/code-tools/jmh/rev/1e59f0d249dc [2017, May 14]

[30] Pradel, M. (2017, February 9). ConTeGe/CoverageMeasurement.java at CovCon · michaelpradel/ConTeGe [online] Available: https://github.com/michaelpradel/ConTeGe/blob/CovCon/CFPDetection/src/cfp/CoverageMeasurement.java#L56 [2017, May 14]

[31] Wikipedia (2017, May 6). Computer multitasking [online] Available: https://en.wikipedia.org/wiki/Computer_multitasking [2017, May 14]

[32] OpenJDK (2014, February 28) jdk8/jdk8/jdk: 43cb25339b55 src/share/classes/java/util/concurrent/ForkJoinPool.java [online] Available: http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/43cb25339b55/src/share/classes/java/util/concurrent/ForkJoinPool.java#l3328 [2017, May 14]

[35] Oracle (2017, May 18) Flow (Java Platform SE 9 [build 170]) [online] Available: http://download.java.net/java/jdk9/docs/api/java/util/concurrent/Flow.html [2017, May 20]

[36] Shade (2017, March 28). code-tools/jmh: 36a2ee9a075e jmh-samples […] [online] Available: http://hg.openjdk.java.net/code-tools/jmh/file/36a2ee9a075e/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_26_BatchSize.java#l59 [2017, May 21]

# Appendix A: Extended background

## 1 Concurrency and parallelism

For simplicity, this section assumes that one CPU core only has one execution unit.

Single-core processors can not run multiple *concurrent* units of a program, or threads, at the same time. They may, however, provide "the illusion of parallelism" [31] through one of the various strategies an Operating System utilize to quickly switch back and forth between different threads. The user is happy thinking his single-core machine execute tasks simultaneously.

Multi-core processors can provide true *parallelism*, limited to the number of cores, or rather threads, that the CPU can execute at any given moment in time.

The prudent developer of fully CPU-bound tasks takes into account that there is a fairly small gain to be made from running more threads than the number of cores the machine has to offer. Adding more threads will increase memory consumption and too many of them will reduce throughout as CPU cycles are wasted on administration and context switching instead of actual application runtime.

As a prudent example, `ForkJoinPool.commonPool()`, by default, uses 1 thread less than the number of available processing units [32].

Only if the threads spend idle time such as waiting on IO-devices is it expected that increasing the number of threads to a number greater than what can be executed in parallel will also yield better performance.

Threads do wait a lot and it is common to use more threads than what the hardware can execute in parallel. Each benchmark test of queue service implementations performed in this study is – from the application's viewpoint – strictly CPU-bound and no more threads will be utilized than the number of threads that can run in parallel.

## 2 Thread-safe code

A queue service implementation exposed to concurrently running threads has to be thread-safe.

Threads that share memory may run into any number of issues. The most predominate one is *race conditions*. Non-atomic operations performed by different threads may interleave each other's work rendering the final result non-deterministic.

Even thread-safe code may face perils lurking around the corner. A common methodology to synchronize access to critical sections is by using locks. A lock is a mechanism that will only let the owner of a lock proceed while all the rest of the threads kindly wait for their turn. If thread A is waiting on a lock that is held by thread B, and thread B is waiting on a lock that is held by thread A, then these two threads are *deadlocked*, unable to make progress. If left unsupervised, these threads will remain in a deadlock until the application process terminate. One way to avoid deadlocks is to make sure that threads acquire locks in the same order.

# 3 Scheduling and orchestrating threads

In order to benchmark a queue service implementation concurrently, there exists a need to orchestrate when worker threads run. This study will use a software library (see chapter “3 Research Strategy and Methods”) that assign worker threads to access each queue service implementation.

Even without such a library or framework, managing threads and asynchronous work [performed in the future] is well-supported in Java.

Since Java 1.5, there exist a set of types that provide the developer with a high-level API for thread management and task execution: `Executor`, `ExecutorService`, `ThreadFactory` and `Executors`. Most of these also come as a “managed” version in Java EE (JSR-236) to be utilized by applications that run in a managed environment.

One `ExecutorService` is `ForkJoinPool` that together with `ForkJoinTask` provide a framework for execution of tasks that can recursively split themselves into smaller pieces. This semantic make it easier to implement “divide and conquer” algorithms at the same time the `ForkJoinPool` can make sure all his worker threads are busy by stealing work from each other [33, p. 38].

`ForkJoinPool` belongs in a package called `java.util.concurrent` (JUC). This package offers many types – sometimes referred to by the community as *primitives* – that can be used to orchestrate threads (for example, `CountDownLatch`, `CyclicBarrier`, `Phaser`) and accomplish more direct thread-to-thread communication (for example, `BlockingQueue`, `BlockingDeque`, `Exchanger`).

Java 8 has taken the abstractions one step further with a parallelizable Streams API and `CompletableFuture`.

# 4 Mechanisms to achieve thread-safe code

Any software component that wishes to be thread-safe has a number of options to choose from.

The most straightforward solution is to get rid of the need for having thread-safe code. Do not share state; use [method-]local variables or share only immutable data structures.

Make only one thread responsible for updates. Such as the application thread in JavaFX that is the only thread allowed to update GUI components (many other GUI frameworks use the same concept).

If mutable data structures are shared amongst threads, then there is a need to synchronize them.

Java’s keyword `synchronized` make threads execute serially. When using `synchronized`, threads compete to take ownership of the same object’s monitor. The object serves as a mutually exclusive *lock* and is many times referred to as a “mutex”.

`Lock` offer a more sophisticated use of locks compared to `synchronized`. For example, one can `tryLock()`. `ReentrantReadLock` guarantees reentrancy just like `synchronized`. `ReentrantReadWriteLock` offers a separation between reader threads (allowed to read at the same time) and writer threads (the bad guys that need an exclusive lock). `StampedLock` may further improve read performance using `tryOptimisticRead()`.

A Java developer may choose to make his class thread-safe by using thread-safe building blocks, or data structures. Some of these building blocks which are not thread-safe can programmatically be made `synchronized` (thread-safety through serialization) by passing a `Collection` or a `Map` to one of many `synchronizedXXX()` methods in the `Collections` utility class. But, this is usually not a good

idea since Java provide non-blocking data structures out of the box. For example, `ConcurrentMap` and `ConcurrentLinkedQueue`. Java does not provide a "ConcurrentSet", but a non-blocking thread-safe `Set` can be derived from a concurrent map either explicitly (`Collections.newSetFromMap()`, `ConcurrentHashMap.newKeySet()`) or implicitly (`ConcurrentSkipListSet`).

Library-provided types that start with "Concurrent" is often misconceived as being "lock free". But these types provide no such guarantees. What is provided, which is most likely the source of confusion, is fast and often *non-blocking* algorithms. But these types oblige to no specification that requires a completely lock-free implementation.

The Java developer is not free from responsibility simply by using thread-safe building blocks. These types provide atomicity only at best on a per-method basis.

Java provides thread-safe data structures for specific fine-grained problems. For example, compare-and-swap an integer (`AtomicInteger`), compare-and-swap a reference (`AtomicReference`). Sharded counter (`LongAdder`).

The modifier `volatile` is a much more low-level construct that makes variable reads and writes atomic and visible to other threads. Moreover, the read and write actions of a volatile field share a happens-before relationship with each other. `volatile` is a very fine-grained tool used in almost peculiar situations to fix specific problems. For example, eternal thread sleep on a cached value, double-checked locking and word tearing of 64-bit data types.

Finally, it should be noted that some unsafe state can simply be moved to a `ThreadLocal`, making it safe. This is a popular way to cache objects that are expensive to create and initialize.

# 5 Testing concurrent code

Writing thread-safe code can be a challenge. Testing thread-safe code even more so.

[34, p. 8] make the claim that the CovCon tool found "thread safety bugs" in 17 out of 18 tested and publicly available Java classes.

Testing concurrent code is hard because of the non-deterministic nature of concurrent code. Often the developer is left to reason about his code as a means to validate the correctness of a program. This is the case for the queue service framework built for this study (see section "6.4.3.3 Correctness of the queue service" for a discussion on this topic).

# 6 Message Queues

There exists a lot of message queues of varying semantics and offerings.

Most notably, a message queue has two different modes or patterns of communication. One is where a message is only consumed by one recipient; *queue* (or *channel*). Another model is where the message may be consumed by none, one or many recipients; *topic*. These two models are also referred to as point-to-point and publish-subscribe.

Message queues can be used to solve different problems.

One use-case for distributed message queues is to integrate decoupled producers and consumers. This usually makes the individual nodes' language- and time-independent.

A distributed message queue can also be used as a horizontal scaling mechanism. In this design, a message represents a piece of work that needs to be executed and more consumers can be spawned as workload increases.

Instead of integrating or facilitating communication between different processes, a non-distributed message queue can be used for communication between modules or threads within a process. A non-distributed message queue is also favorable to use as a mock in a testing environment instead of using a real and sometimes costly third party service.

The use-case for non-distributed message queues within one single runtime is limited. The whole concept of such a thing borderlines with event-driven programming.

*Events* are historically something that has only one source but many subscribers, triggered by an object and consumed by callbacks. With this historical perspective in mind, an in-process publish-subscribe message queue could be referred to as an *event bus*. The difference here is purely semantics.

It’s also worth noting that Java 9 will come with a publish/subscribe framework; the Flow API [35].

# Appendix B: Machine specification

| | |
|---|---|
| Make and model: | Gigabyte P35X v3 |
| CPU: | 4-core Intel® Core™ i7-4720HQ @ 2.60 Ghz using hyper-threading (8 logical processing units) |
| Memory: | 16gb DDR3L 1600 MT/s |
| OS disk: | Liteon IT LMT-256L9M |
| Benchmark/code disk: | Samsung SSD 850 EVO 1TB |
| OS: | Windows 10 Pro, version 1607, build 14393.1198 |
| Java runtime: | Oracle's Java HotSpot™ 64-bit Server VM, build 1.8.0_131-b11 |
| Source code repository tag: | 1.0.0 |

# Appendix C: Used commands

Details of the commands used to run the benchmarks are described in the "README.md" file provided in the source code repository [14]. If the command is too long to fit one line, it is split into two using the '\' character.

`bench` is a Gradle task that will execute benchmarks. `reorganize` will reorganize the contents of the JMH results file and compute score gains; all of which is put into a new file with suffix "_reorg". The reorganize task does not change any of the values JMH produced. It was put together as a useful tool only to simplify the author's task of constructing the tables in chapter "4 Result".

Command-line argument `-Pr=T` specifies which measurement mode to use. "T" is a regex and will cause the runner to execute benchmarks found in `QueueServiceBenchmark.Thrpt`.

`-Pq=8` specifies how many queues the benchmark should use (8).

`-Ptg` specifies the number of threads we want to use. The default is 1 reader and 1 writer thread.

`-Plf` and `-Prf` specifies the name of the log- and results file respectively.

## 1 Uncontended

```
gradlew bench -Pr=T -Pq=8 -Plf=uncontended_log.txt -Prf=uncontended_res.txt
gradlew reorganize -Prf=uncontended_res.txt
```

## 2 Contended readers

```
gradlew bench -Pr=T -Pq=8 -Ptg=4-1 Plf=contended_r_log.txt \
  -Prf=contended_r_res.txt
gradlew reorganize -Prf=contended_r_res.txt
```

## 3 Contended writers

```
gradlew bench -Pr=T -Pq=8 -Ptg=1-4 Plf=contended_w_log.txt \
  -Prf=contended_w_res.txt
gradlew reorganize -Prf=contended_w_res.txt
```

## 4 Contended readers + writers

```
gradlew bench -Pr=T -Pq=8 -Ptg=4-4 Plf=contended_rw_log.txt
  -Prf=contended_rw_res.txt
gradlew reorganize -Prf=contended_rw_res.txt
```

# Appendix D: Benchmarking non-steady state

See section “3.3 Data Analysis” for an introduction to the different measurement modes that JMH support. This appendix will dig deeper into the difference between throughput and single shot, as it applies to this study.

Ideally, the code executed under a benchmark has a steady-state. This makes the performance independent of the run time. If on the other hand, the benchmark performance is impacted by the run time, then, in theory, the run time must not be fixed anymore – the number of invocations has to be fixed and conclusions about performance pertain only to the given condition set by the number of invocations.

As an example, `ArrayList` has to grow an internal array [every now and then,] as elements are added to the list. This growth is obviously not free of cost and will be recurring over time.

Assume that we write a benchmark method that inserts one element into the list. Also assume that the `ArrayList` instance remains the same throughout each measurement run (*iteration*, not *invocation*).

If *throughput* measurement mode is used, then we will measure how many invocations were made to our benchmark method – how many elements did we insert – for a defined time period (by default 1 second).

It makes sense to think that increasing the time period will raise our confidence in the throughput numbers, but not affect the results too much! However, in this example, if we add enough of more seconds to the iteration time, then the `ArrayList` will suffer from additional growth cost because, with more execution time, more elements are added. This is a trend, although probably a diminishing one because of the way `ArrayList` grows will make the actual growth happen less frequently over time. But, the important point to be made here is that we will find that our measured performance *decreases* as we add more time. We say that in this case, our benchmarking is profiling *non-steady* state which is not the textbook ideal.

For the `ArrayList` example, a better measurement mode may be *single shot*. Single shot measures only one invocation or, the insertion of one element. By doing so, we remove the time dependency.

Unless we increase our forks and iteration count tremendously, then one single invocation will yield far too few observations. To counter this, a *batch size* parameter can be provided to the single shot measurement mode. This batch size will increase how many invocations JMH considers to be one observation. In doing so, we have also changed the semantics of the conclusions drawn from the test. We do no longer measure isolated element insertions. With a batch size of say 1 000, we measure and draw conclusions about how long time it took to insert exactly 1 000 elements.

The queue implementation used for `SynchronizedQS` and `ReadWriteLockedQS` was `ArrayDeque`. This implementation too has to manage an ever growing internal capacity. The argument can therefore be made, and it is a justified one, that at least half of our queue service implementations had no steady state. So one may think that single shot would have been a better measurement mode to use. In fact, JMH has written that the benchmark mode “single shot” is “the only acceptable benchmark mode” for benchmark code that doesn't have a steady state [36].

First, it must be said that this is one of the reasons why this study made the reader threads mark messages as completed as fast as possible so that they could be purged from the queues as fast as possible. This is at least an honest attempt to keep the queue sizes down and hopefully keep them there.

Another possible counteraction would be to construct each new `ArrayDeque` with a very high internal capacity to start with. However, this would only move a *potential* performance penalty from the future into an *actual* performance penalty in the very present when new queues need to be instantiated and the first element added.

This study did not create `ArrayDeque` with an application configured size of the internal capacity. Nor was single shot mode chosen.

The chief reason behind this decision comes down to this: We can remove the time parameter but only to add another one: batch size. Instead of limiting the study to a fixed number of invocations (batch size), it was more preferred to limit the study to a time-based run time; 1 second. That is, any conclusions drawn from the results in chapter 4 does scientifically only apply to the queue service framework utilized under 1 second. Limit by a fixed number of reads and writes, or limit by a fixed number of run time seconds? Pick your poison.

Flavor is not the only reason behind the choice. Throughput was chosen because this author deduces it will yield the most reliable results for the research question asked.

A queue service is by no means expected to have a steady-state and there is no way for us to magically give the queue service a steady state. In particular so when readers and writers execute in parallel. There is no guarantee that readers will move messages out from the queues as fast as writers can put new messages in. If writers outperform readers, then the queues will have to accommodate growth.

This study is all about the non-steady state. This study wishes to benchmark the performance over time and is most willing to incorporate *all* factors a real-world component has to endure; including growth and unbeknown red midget devils lurking around inside the code universe. To this end, the throughput measurement mode is spot on.

Even for the previous example using an `ArrayList` benchmark, single shot measurement mode is debatable or the benchmark setup is convoluted. Either the benchmark ought to compare performance over time, simply because it matters. Or the benchmark has to define important performance-impeding aspects as benchmark parameters.

Furthermore, single shot is absolutely no guarantee that the benchmark measures a steady-state. For example, an `ArrayList` with a default initial capacity will allocate the array storage container *lazily* on the first element insertion. That is to say, the benchmark method would not only test insertion of one element but also the time it takes to create a new array. Comparing the results of this test with say a `LinkedList` who does not have this one-time initialization cost would completely annihilate comparative conclusions drawn.

For your convenience, `QueueServiceBenchmark` actually defines benchmarks using all supported measurement modes and these benchmarks use the same workload. Here is an example command you can use to run the uncontended test using single shot (only difference being is that '`-Pr=T`' is replaced with '`-Pr=SS`'):

```
gradlew bench -Pr=SS -Pq=8 -Plf=uncontended_log.txt -Prf=uncontended_res.txt
```

Department of Computer and Systems Sciences
Stockholm University
Forum 100
SE-164 40 Kista
Phone: 08 – 16 20 00
www.su.se