Mittuniversitetet
MID SWEDEN UNIVERSITY

Performance and Cyclomatic Complexity Correlation in Java Reactive Frameworks

Anton Cervin & Christian Trenter

Final Project
Main field of study: Computer Engineering BA (C)
Credits: 15
Semester/year: Spring 2022
Supervisor: Ali Hassan Sodhro
Examiner: Felix Dobslaw
Course code/registration number: DT133G
Programme: Software Engineering

Performance and Cyclomatic Complexity Correlation in Java Reactive Frameworks

Anton Cervin and Christian Trenter
Department of Computers and Systems Sciences
Mid Sweden University
Östersund, Sweden
{ance1901, cher1903}@student.miun.se

***Abstract*—Software performance is of interest to all who want to lower their hardware costs and provide services with speedy responses to users, to this end reactive programming can be employed. Therefore it is important to measure the performance of tools such as reactive frameworks and to compare and contrast them with each other in order to improve the development of these tools, and help provide developers with the information they need when searching for a reactive framework that suits their project. To achieve this the study will aim to indicate the reproducibility of the research on reactive framework performance conducted by Ponge et al. on three commonly used reactive frameworks. Further, a root cause analysis to identify the sources of the identified bottlenecks will be carried out and complemented by suggestions for improving the performance of those parts in the reactive frameworks that are causing performance issues. An analysis of how performance correlates with the cyclomatic complexity of the frameworks will also be presented. We find, that in some test cases, the performance of the frameworks differs depending on the machine used, and that the latest framework versions do not show a marked performance increase. Further, we discover hotspots in the Mutiny framework and reason about potential synchronization bottlenecks. We attempt to find a novel use case for cyclomatic complexity as an indicator for performance but find that no correlation exists between cyclomatic complexity and performance for reactive frameworks.**

***Index Terms*—Benchmarking, Java, RxJava, Reactor, Mutiny, Reactive frameworks, Reactive streams, Reactive programming**

I. Introduction

Since the turn of the last decade the software industry has been moving away from monolithic systems toward microservices, a type of distributed system where the application as a whole is composed of several smaller applications with a delimited functional scope and an API for allowing other services in the architecture to utilise its function. These services can be scaled up or down depending on demand, which allows for more efficient utilisation of hardware and thereby lowers the costs of cloud services [1].

Imperative style blocking I/O is not resource efficient for services that need to scale on demand, as each new connection will instantiate a new thread which at some point might block for I/O, each new thread requires memory and puts additional strain on the kernel scheduler. The ability to scale services according to demand is important for modern cloud services and to enable this services must be able to send requests to each other without blocking the thread making the request whilst waiting for a response as each created thread requires additional memory and context switches [2]. Asynchronous programming allows for a "fire and forget" model to be achieved, a request can be sent by a thread to another service and immediately be reused for another purpose. The response is then handled by another thread with no active waiting having been required.

Reactive programming is an inherently asynchronous programming paradigm, it is in essence a publish-subscribe pattern as the data is pushed through a pipeline of operators that transform the data before consuming or sending it down the pipe to a proceeding operator. Reactive Streams is a protocol accompanied with dependencies for implementing reactive extensions [3], in a previous study by Ponge et al. [2] three *reactive extensions* or *reactive frameworks* were benchmarked against each other, namely RxJava, Reactor and Mutiny. This study showed that RxJava performs best for single operation reactive types whilst Reactor proved to be the slowest. For multiple operations RxJava and Reactor showed similar performance whilst Mutiny, which includes less optimization, performed worse than the other two. However, the study did not include a root cause analysis as to why these frameworks manifest different performance characteristics.

A core tenet of science is that published research should be reproducible, C. Collberg and T. Proebsting [4] examine the repeatability of 601 papers from ACM journals and conferences and find that 33% of them have weak repeatability due to a number of different factors such as source code not being provided and build issues. The authors go on to say: "Repeatability and reproducibility are cornerstones of the scientific process, necessary for avoiding dissemination of flawed results". In accordance with this statement our study will examine the reproducibility of the research conducted by Ponge et al. [2].

Good performance makes for happy customers and lower operating costs and is therefore an important part of software development and maintenance [5]. A crucial part of this process is the identification of software components that causes performance drag [6]. Bottlenecks are one such common issue, these are commonly found when synchronizing access to shared data as it creates a funnel for threads leading to decreased concurrency and thereby performance. Locating

bottlenecks however is often a labour intensive, so research is being done to construct smart profiler tools that help alleviate the task for developers [7], [8], [9], [10]. A profiler is a tool that can obtain frequency, memory usage and execution time [5]. For this study, we used a profiler in conjunction with the benchmarks [11] constructed in the study conducted by Ponge et al. [2] to try and identify ”hotspots” in the reactive frameworks and suggest possible solutions.

Cyclomatic complexity (CC) is a metric measuring the complexity of a piece of software. The value of CC represents the number of linearly independent paths in the software, i.e a program with one if statement has a CC of 2. It is most associated with predicting software maintainability but has also shown a correlation with the number of defects found in a code base [12]. In this paper we explore the possibility of CC being correlated with performance in the three reactive frameworks mentioned above.

II. Purpose and Contributions

Reactive programming is becoming increasingly popular [13]. There are multiple frameworks available and it is hard to know which one to pick based on required needs. A study conducted by Ponge et al. [2] compares three different reactive java frameworks, namely RxJava [14], Reactor [15] and Mutiny [16] where equivalent operators and operator chains are benchmarked against each other.

A reproducibility study will be conducted to ensure the results can be replicated. What the previous research did not take into account was the potential performance differences arising from running the tests on different hardware. Therefore, the study will include an investigation to establish if the results of *“Analysing the performance and costs of reactive programming libraries in Java”* will differ when the benchmarks are executed on different machines with other hardware specifications, as well as how the latest versions of the frameworks perform under the same benchmarks.

Finally, the study will include an evaluation of the code to check for potential bottlenecks and if cyclomatic complexity and performance correlate in the evaluated frameworks.
To the best of our knowledge we are the first to investigate if cyclomatic complexity correlates to performance in reactive frameworks and the first to document an investigation into the reasons as to why the frameworks mentioned above showcase different performance characteristics.

RQ1 *Can the study “Analysing the performance and costs of reactive programming libraries in Java” be successfully reproduced?*
RQ2 *Do the findings in “Analysing the performance and costs of reactive programming libraries in Java” generalise to different hardware and framework versions?*
RQ3 *Identify bottlenecks in each framework and suggest adjustments.*
RQ4 *How does cyclomatic complexity and performance correlate in the evaluated frameworks?*

The contributions of this study includes:

- Determining the reproducibility of the study *“Analysing the performance and costs of reactive programming libraries in Java”* [2].
- Show if and how the performance differs depending on hardware and version.
- Present identified bottlenecks in the code.
- An investigation on the possible correlation between cyclomatic complexity and performance.

III. Background

Reactive programming is a programming paradigm that follows the principles of the *Reactive Manifesto* [17], which can be summarized by the following keywords:

- **Responsive**: The system provides rapid and consistent response times.
- **Resilient**: If a system fails for any reason, it should still be responsive. This is achieved by “replication, containment, isolation and delegation”. For example, containing the failures within a component and isolating all components from each other will make a failure in one component not affect others.
- **Elastic**: Should react to changes and stay responsive under dynamic workloads.
- **Message driven**: In order to adhere to the resiliency principle, reactive systems need to establish a boundary between components by relying on asynchronous message passing. This ensures loose coupling, isolation and location transparency.

This paradigm also utilizes asynchronous data streams and thus effectively implements the observer pattern [18]. The observer pattern is one out of many *design patterns*. A design pattern can be described as a reusable solution to common problematic design situations in software development [19]. The ground principle of the observer pattern is to allow modelling a one-to-many relationship between objects. When an observable changes state any observer that is subscribed to said observable will get notified and updated accordingly [19].

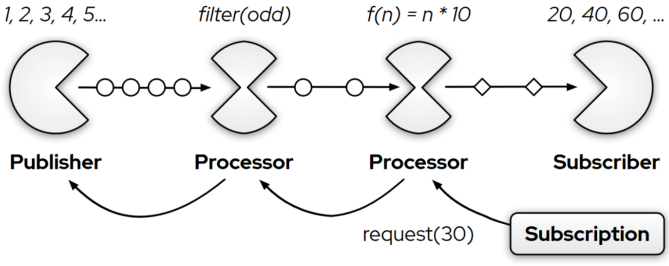


Fig. 1: *Example of a reactive stream pipeline [2].*

So, what is a *stream*? It is an orderly sequence of events that take place over time. It includes values, errors, and completion signals. These are emitted by a publisher (Observable)

and consumed by a subscriber (Observer). An example of a reactive stream pipeline can be seen in *Fig. 1* where a publisher is emitting a stream of numbers *(1, 2, 3, 4, 5...)*, the stream then continues through two processors which filters out all odd numbers and multiplies all even numbers *(2, 4, 6, 8...)* with 10. In the end the subscriber then receives *20, 40, 60, 80...* .

A. *RxJava & Reactor*

RxJava and Reactor share many similarities. They both implement optimizations such as operator fusing and pre-fetching, as well as naming operators according to functional programming terminology. RxJava and Reactor are based on the joint research done in reactive-streams-commons [15], [20] and thus share common characteristics. The biggest difference between these frameworks is the reactive APIs. Reactor provides two base classes, Mono and Flux, for single and multi-event streams whilst RxJava has five. Developers thereby have more options when developing their solutions with RxJava, e.g., backpressure support is optional with the Observable and Flowable classes (the latter implementing backpressure whilst the other does not) [14].

Cyclomatic complexity indicates the complexity of a code base [21], RxJava and Reactor have considerably greater cyclomatic complexity than Mutiny as can be seen in *Table I*, this can be attributed to the optimizations employed by these frameworks such as operator fusion which describes merging operators in order to reduce the number of round trips made to memory thereby, in theory, increasing performance.

B. *Mutiny*

Unlike RxJava and Reactor, Mutiny [16] does not include operator fusing or pre-fetching and thus has a much smaller code base. When comparing the lines of code for each framework Reactor has more than three times the lines of code as Mutiny and RxJava almost five times more, as shown in *Table I*.

The above discussion sums up the idea of Mutiny, keeping it small and simple whilst still adhering to the reactive streams specification. An example of this is that instead of being confronted with hundreds of possible methods when using IDE completions Mutiny instead shows about ten. This kind of design will often lead to more verbosity but improve readability and navigability [2]. As stated above the cyclomatic complexity (CC) of Mutiny is far lower than that of RxJava and Reactor, metrics gathered by running *Sloc, Cloc and Code* [22] show that RxJava has a CC score of *11 690* and Reactor *13 533*, in contrast to Mutiny's score of *3 220* for the latest versions. More information on cyclomatic complexity can be found below *(D. Cyclomatic Complexity)*.

C. *Bottlenecks*

A bottleneck occurs when the performance of a system is dragged down by a single component, often this stems from inefficiently synchronizing access to shared data in a multi threaded application, the threads heap up as they try to acquire the lock causing them to become blocked and unable to continue execution. The performance of the system as a whole is now dependant on this one block of code, just as the speed at which water can be poured from a bottle is dependant on the width of its neck. Bottlenecks are a common problem in real-world software and while some are easily fixed [7] most are generally difficult to detect and address [8]. Application profiling is one way to expose bottlenecks, using a profiling tool to obtain frequency, memory usage and elapsed execution time of method calls [5] helps developers identify hotspots in the code that might be holding back the performance of the software and requires attention.

D. *Cyclomatic Complexity*

As mentioned in the introduction cyclomatic complexity (CC) is a measure used to evaluate the complexity of a software program [23]. It was developed in 1976 by Thomas J. McCabe and is therefore also referred to as *"the McCabe metric"* [24]. Historically, the actual usefulness of CC has been questioned and the metric was, according to M. Shepperd & D.C. Ince, accepted by researchers relatively unquestioned [24]. Cyclomatic complexity is in essence an estimate of decision making nodes and paths in a program.

IV. Related Work

The work by Ponge et al. [2] investigates the performance of three reactive libraries that have been introduced in the background. This was done by creating a suite of tests for CPU-bound *single- and multi/-operator pipelines* and *single and stream publishers* as well as I/O bound *multi-operator pipelines for streams in file system operations* and *network requests*.

The results showed that for all CPU-bound single operations (*uni individual operations* and *single direct transformations*), both with single and multiple operators, RxJava had the best performance, thereafter Mutiny and lastly Reactor.

For all CPU-bound stream pipelines (*multi individual operations* and *stream direct transformations*), both with single and multiple operators, RxJava and Reactor showed near equal performance whilst Mutiny lagged.

For handling file operations and network requests (I/O-bound benchmarks), the results showed no statistical advantage in either of the libraries.

Comparisons of frameworks have been done before. An example of this is the research conducted by Alkatheri et al. [25] where four different frameworks were compared: Apache Hadoop, Apache Spark, Apache Storm, and Apache Flink. They concluded that Apache Flink was the best performing framework followed by Spark but these frameworks are not related to reactive programming and thus differ from our research.

However, there is an example of research aimed at investigating the usage of reactive frameworks and the blocking violations that occur when calling blocking system calls from non-blocking threads in reactive projects. F. Dobslaw et al. [18] investigate the occurrences of blocking violations in projects implemented with RxJava and Reactor frameworks.

Library	*Java Lines of Code (LOC)*	*Number of Files*	*Cyclomatic Complexity (CC)*
RxJava 3.0.13	100313	907	11750
RxJava 3.1.4	182016	855	11690
Reactor 3.4.8	72858	444	13358
Reactor 3.4.18	116442	418	13533
Mutiny 1.0.0	21177	300	2840
Mutiny 1.4.0	40056	325	3220

TABLE I: *Code metrics for framework versions used by Ponge et al. [2] and the latest framework versions used in our measurements.*

A profiling tool, BlockHound, is used in conjunction with a custom made tool to automatically append BlockHound to GitHub projects targeted for investigation. The projects included for testing had large code coverage to provide ample data for the profiler as well as being developed in Java with the use of RxJava and/or Reactor. The study found that from the 29 projects that were successfully evaluated 24.1% had one or more blocking violations. Through stack tracing the blocking violations it was found that the main causes of this were five system calls used incorrectly by calling them in non-blocking threads.

The study by F. Dobslaw et al. [18] differs from our intended research as we will investigate the performance of the reactive frameworks and attempt to identify if and where bottlenecks occur rather than investigating incorrect usage of these frameworks.

Bottlenecks are a common occurrence in software [26], [27], [28] but locating them is often a labour intensive undertaking. To find bottlenecks software has to be executed in such a way as to expose them. Such performance tests are not always readily available as most tests are often written to check for correctness rather than performance. Toffoloa et al. [7] developed PerfSyn to automatically synthesize test programs that expose bottlenecks. The tool gathers execution feedback and employs graph search algorithms to mutate inputs in such a way as to reveal performance issues in the tested software. However, the tool is not yet publicly available and could therefore not be used in this study. Similar to the previous study is the work of T. Yu and M. Pradel in [8]. The authors developed the tool SyncProf which detects and suggests improvements to remove synchronization bottlenecks by analysing execution time and CPU usage under varying workloads. However, the tool is made for analyising C/C++ applications and as such was not employed in our study. T. Field and R. Nair developed a lightweight runtime profiler GAPP (Generic Automatic Parallel Profiler) to identify serialization bottlenecks [9] by tracing context switches inside the kernel to keep track of all active threads, allowing the tool to measure the degree of parallelism which is an indicator for the existence of synchronization bottlenecks.

A number of studies have been done on, or related to, cyclomatic complexity [29], [30], [31]. Gill, G.K. and Kemerer, C.F. investigated the relationship between cyclomatic complexity and software maintenance with the assumption being that systems with high cyclomatic complexity would be harder to maintain [29], Farooq et al. simply proposed a model to decrease cyclomatic complexity [30] while Hutajulu et al. measured a software engineer's programming skills by comparing it to the cyclomatic complexity of the code produced [31]. However, to our knowledge, few have compared actual performance in correlation to the stated metric.

V. Research Methodology

This study aimed to investigate the reproducibility of the study *Analysing the Performance and Costs of Reactive Programming Libraries in Java* [2]. The benchmarks were run on different hardware with the same framework versions used by Ponge et al. [2], as well as the latest versions. We investigated the performance of the frameworks with a profiler tool and analysed the results. Finally, we compared the performance results to the cyclomatic complexity score gathered by using *Sloc, Cloc and Code* [22]. This section is comprised of the following three subsections: Preparation, Data Collection and Analysis.

A. Preparation

The preparation is divided into two subsections: *Hardware* and *Software*.

1) Hardware: Three different machines were used for our measurements, one for the reproducibility part and two for finding performance variation due to hardware and/or framework versions. The specifications of all machines can be seen in *Table II* where we have listed all hardware and software specifications of the machines used.

Hardware limitations - RQ1

- *Intel Core i7-8565U* as opposed to *Intel Xeon CPU* in the study reproduced.
- *2.80GHz* clock speed as opposed to *3.50GHz* in the study reproduced.
- *16GB RAM* as opposed to *252GB RAM* in the study reproduced.
- *4 cores* as opposed to *16 cores* in the study reproduced.

Hardware	*Machine Reproduced*	*Machine* 1	*Machine* 2	*Machine* 3
CPU	Intel Xeon CPU	Intel Core i7-10870H	Intel Core i7-1165G7	Intel Core i7-8565U
Clock Speed	3.50GHz	2.21GHz	2.50GHz	2.80GHz
Number of Cores	16	8	2	4
RAM	252GB	16GB	16GB	16GB
Software	*Machine Reproduced*	*Machine* 1	*Machine* 2	*Machine* 3
Operating System	Linux	Windows 10	Windows 10	Ubuntu 16.04
JDK Version	11.0.11+9	16.0.2+7	16.0.2+7	11.0.14+9
Garbage Collector	Shenandoah GC	Shenandoah GC	Shenandoah GC	Shenandoah GC
Linux Kernel	4.4.0	-	-	4.4.0

TABLE II: *Machine Specifications.*

2) Software: To assess the reproducibility of the study (*RQ1*), the benchmarks were run with the same, or as close as possible to, software used by Ponge et al. [2]

We used *Machine 3* for this part with Ubuntu installed [32]. Ubuntu 16.04 was chosen due to its compatibility and availability with kernel 4.4.0. However, we were unable to source the same OpenJDK used by Ponge et al. [2] (OpenJDK 11.0.11+9) so instead used OpenJDK 11.0.14+9.

To determine performance differences across hardware and versions (*RQ2*) we configured two machines (*Machine 1* and *Machine 2*) with the same software specifications as presented in *Table II*. The benchmarks were run for the following framework versions:

- *RxJava 3.0.13 & RxJava 3.1.4.*
- *Reactor 3.4.8 & Reactor 3.4.18.*
- *Mutiny 1.0.0 & Mutiny 1.4.0.*

To verify if the results generalised to different hardware we used the same framework versions as in the previous study. For this research to be relevant we also tested the latest versions to see if and where performance increases have been achieved since the previous research was published.

To gather the data needed for locating bottlenecks (*RQ3*) we needed a profiler tool to acquire a performance analysis of each tested framework. The test suite allowed for a profiler called *Java Flight Recorder* to be used with the tests through the JVM arguments. This generated a JFR-file for each benchmark which in turn could be examined with a tool called *VisualVM*. Hence, the following additional software was required for *RQ3*:

- *Java Flight Recorder - A JVM integrated profiler with low overhead [33].*
- *VisualVM - A visual interface for viewing profile files, such as those gathered by JFR [34].*

Finally, we needed a tool to be able to calculate the cyclomatic complexity of any given code base. Many tools are available to gather this metric such as *Cyclo* [35] and *JArchitect* [36] but we opted to use *Sloc, Cloc and Code [22]* as it is free to use and claims fast and accurate calculations.

Software limitations - RQ1

- *OpenJDK 11.0.14+9* as opposed to *OpenJDK 11.0.11+9* in the study reproduced.

B. Data Collection

To collect data, the same test suite as Ponge et al. [2] was used. However, different JVM arguments and measurement iterations were used to collect the data needed to answer each research question. A short explanation of all benchmarks follows below:

1) Uni individual operators: A comparison of the performance of transforming an event value (*map*) and chaining this value with another operation (*chain*) on the single operation types *Single* for RxJava, *Mono* for Reactor and *Uni* for Mutiny [2].

2) Single direct transformations: Compares performance for stream publishers i.e *Single*, *Mono* and *Uni*, with multi-operator pipelines. This is done by randomizing a number from a publisher, converting it to an absolute value using *Math::abs* and then converting it to a hexadecimal string by using *Long::toHexString*. These results are then concatenated to enclose the string in brackets [2].

3) Multi individual operators: Comparing performance of transforming values (*map*), and then selecting values on the basis of a predicate (*filter*), chaining with a single-valued operation (*mapToOne*) and finally chaining with a stream-returning operation (*mapToMany*) on backpressured stream pipelines (*Multi* for Mutiny, *Flux* for Reactor and *Flowable* for RxJava. [2].

4) Stream direct transformations: Compares performance for stream publishers i.e *Flowable* for RxJava, *Multi* for Reactor and *Flux* for Mutiny [2], with multi-operator pipelines. Each pipeline starts with a random number which is then transformed into an absolute value with *Math::abs*. For each item a stream of strings is produced with the corresponding hexadecimal number using *Long::toHexString* followed by concatenating the "!"-character three times [2].

5) Text Processing: The publisher reads lines from the book *Les Misérables* by Victor Hugo. First, the operator immediately after the publisher discards all blank lines followed by an

operator that calculates the number of characters in the line. After that these numbers are written to a file on a dispatched worker thread pool. Finally, the last operator transforms the character count to a string prefixed with an arrow [2].

6) Network Requests: The publisher issues six concurrent HTTP requests to collect the content of the same book as in the text processing benchmark, but collected from an HTTP server. When all responses have been triggered and collected they become a stream of all six HTTP responses. For each item the book text is extracted and finally all characters in the collected book are calculated.

RQ1: Can the study "Analysing the performance and costs of reactive programming libraries in Java" be successfully reproduced?

For the first part, we simply used the test suite without any extra arguments on *Machine 3* with the following command: *java -XX:+UseShenandoahGC -jar target/benchmarks.jar*. Out of the box, the tests are run with 5 forks, 5 warm-up iterations and 5 measurement iterations, with each benchmark being tested for 10 seconds per iteration. An average score is presented at the end of the tests.

When collecting data for Network Requests and Text Processing, Ponge et al. [2] claim to have used 1000 samples, however when analysing the histogram showing the results (see *Fig. 2 (b)*) it becomes clear that this number is incorrect. Therefore we opted to run the tests to acquire 5000 samples. This was done by adding the following arguments to the line above: *java -XX:+UseShenandoahGC -jar target/benchmarks.jar -f 5 -wi 10 -i 1000 .*TextProcessing.** and *java -XX:+UseShenandoahGC -jar target/benchmarks.jar -f 5 -wi 10 -i 1000 .*NetworkRequests.**.

Thereby running 5 forks, 10 warm-up iterations and 1000 measurement iterations per benchmark. The warm-up iterations are required for the compiler to reach a stable state.

RQ2: Do the findings in "Analysing the performance and costs of reactive programming libraries in Java" generalise to different hardware and framework versions?

The same procedure was followed to collect data for RQ2 but done twice on each machine. First the tests were conducted with the same framework versions used by Ponge et al. [2], followed by the latest stable version of each framework. The reasoning for doing the benchmarks both with the same framework as well as the latest was to conclude if any differences found was due to hardware or different versions.

RQ3: Identify bottlenecks in each framework and suggest adjustments/recommendations

The test suite was run multiple times on *Machine 1* and *Machine 2* to eliminate any outlier runs caused by the garbage collector. The command used for this collection included a profiler argument: *java -XX:+UseShenandoahGC-jar target/benchmarks.jar -f 5 -wi 5 -i 100 -prof jfr*.

This generated a JFR (Java Flight Recorder) file for each benchmark that could be viewed and analysed with the help of VisualVM. This allows the user to order the method calls according to execution time and filter them according to library for easier identification of potential trouble spots in the framework.

RQ4: How does cyclomatic complexity and performance correlate in the evaluated frameworks?

We gathered code metrics of the latest versions of the reactive frameworks by using *Sloc, Cloc and Code* [22]. The core java code was analysed by ignoring tests and other miscellaneous modules for all frameworks.

C. Analysis

The results gathered from the test suite were sufficient to answer RQ1 and RQ2. By running the Java Flight Recorder together with the test suite sufficient data was collected and used to investigate potential bottlenecks in each framework to address RQ3 and finally, the cyclomatic complexity gathered for each framework was used to find any correlation between CC and performance.

1) RQ1 - Analysis: To determine if the results from the benchmark runs conducted for this study were consistent with the results gathered by Ponge et al. [2] we created diagrams that could be compared to the diagrams in the original study. If these diagrams correlated we could safely assume that the particular benchmark was indeed reproducible.

2) RQ2 - Analysis: For *RQ2* it was important to run all tests with two different framework versions so we could see if any potential differences were due to hardware or if these potential differences were due to newer versions. Our reasoning when analysing the data was as follows:

Old framework version

- If the results from the benchmarking tests were the same on all machines the results indeed generalise to different hardware for that particular version.
- If the results would differ on one or both of *Machine 1* or *Machine 2* the results indicate that it does not generalise to different hardware.

New framework version

- If the results from the benchmarking tests were the same on all machines and these correlated to the results gathered by the original study we can assume that the results indeed generalise to different hardware and the newer versions as well.
- If the results would differ on *Machine 1* or *Machine 2* but not the other this would indicate that it is due to different hardware.

- If the results would differ on both *Machine 1* and *Machine 2* and these two correlate that would indicate that this is due to the newer version.

3) RQ3 - Analysis: To identify the bottlenecks Java Flight Recorder [33] was used to generate profile files of the executed benchmarks. VisualVM [34], a tool that provides the ability to view the generated profile file as a list of executed methods and their subsequent execution times, was used to analyse these files. By using Java Flight Recorder and VisualVM we could identify the hotspots in the reactive frameworks while running the benchmark suite. The methods identified were then further investigated to identify the causes as to why they appeared as hotspots to establish whether or not they might be causing a bottleneck. The method in question might take up a lot of CPU time, but this does not in itself indicate a bottleneck. If the method is repeatedly called, such as *onNext()* which is called for every regular event, it makes sense that this method will be highlighted by the profiler. For example, in the Mutiny benchmark *(multi_mapToOne in MultiIndividualOperators)* the *MultiFlatMapOp.onItem()* method, which is called by *MultiSubsciber.onNext()* and thereby called for each item published takes up a near quarter of the total CPU-time. The self-time of this method, i.e. the time spent executing, excluding the time taken for calls to other methods, was 23.4% of the total execution time of the benchmark for *Machine 1* and 23.5% for *Machine 2*, but this does not indicate a bottleneck.

However, as the benchmark suite tests equivalent operators and operator chains amongst the different frameworks, we can compare the CPU time of equivalent methods across the frameworks in the hope of isolating methods that are taking an undue amount of processing time. These methods can then be investigated further to identify the code snippets that might be causing the performance drag. We can further limit the search scope by identifying those methods that have the most severe performance differences.

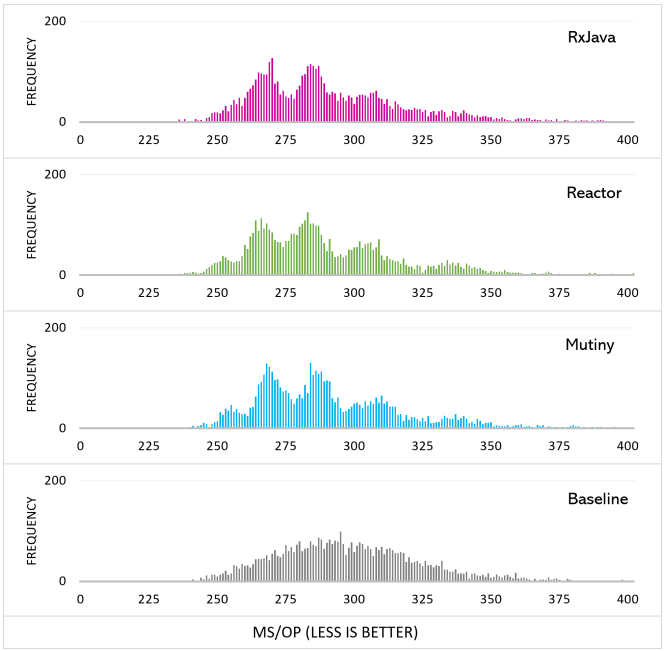
4) RQ4 - Analysis: For *RQ4* we ran *Sloc, Cloc and Code* [22] to verify the changes in the results obtained by Ponge et al. [2]. This data was then compared to the performance of each framework to see if there is any correlation between code complexity and stated performance.

VI. Results

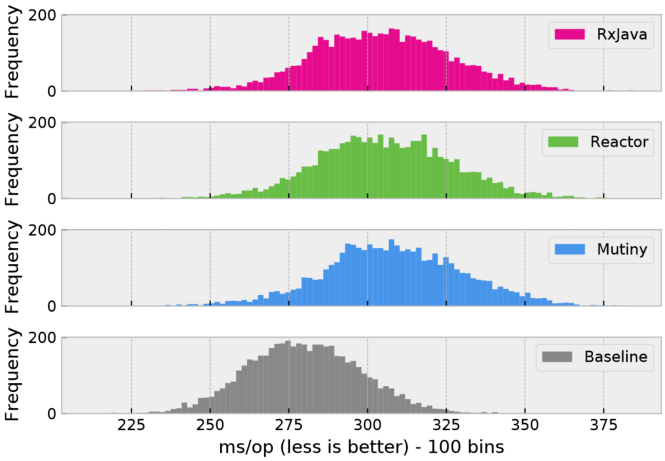
This section presents the results with answers to the research questions in section II.

A. ***RQ1*** - *Can the study "Analysing the performance and costs of reactive programming libraries in Java" be successfully reproduced?*

For most benchmarks, yes. After configuring *Machine 3* and running the tests our results showed that each CPU-bound benchmark[1] conducted on *Machine 3* corresponds almost exactly to the results gathered by Ponge et al. [2].



(a) Text processing - *Machine 3.*

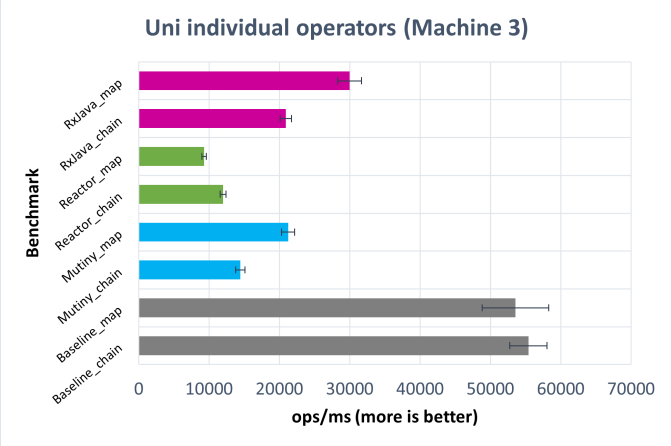


(b) Text processing - Ponge et al. [2].

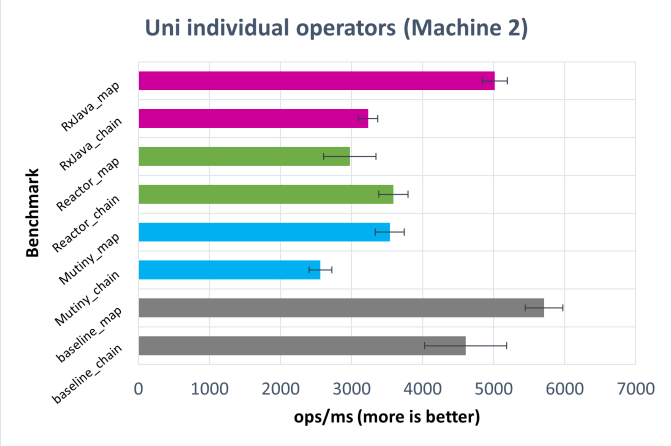
Fig. 2: *Comparison of text processing between reproduced data (a) and data from the original study (b).*

However, the results for the Text processing benchmark gave unexpected results. *Fig. 2* shows a comparison of the reproduced results from the text processing benchmarks and the results acquired in the original study by Ponge et al. [2]. The reproduced results shows that all reactive frameworks outperformed the imperative code baseline which is the opposite of the results in the original study. What should be noted is the apparent overloading sub-distributions evident in each of the frameworks for Machine 3 as compared to the single distribution present in the original study, however this does not affect the conclusions. The results for network requests showed negligible performance differences compared to the results obtained by Ponge et al. [2].

[1]Worth noticing is that the *mapToOne* results for RxJava in the Multi individual operators benchmark has been excluded, both in this paper and in the study conducted by Ponge et al. [2]. This is due to a bug with the *flatMapSingle* operator causing a memory exhaustion.



(a) Uni individual operators - *Machine 3.*



(b) Uni individual operators - *Machine 2.*

Fig. 3: *Comparison of uni individual operators - (a) shows the successfully reproduced results and (b) is the results from Machine 2.*

B. ***RQ2*** *- Do the findings in "Analysing the performance and costs of reactive programming libraries in Java" generalise to different hardware and framework versions?*

When running the benchmarks on *Machine 1* and *Machine 2* all benchmarks produced the expected results when running on *Machine 1*, both with the early and the latest version[2] of frameworks. However, for *Machine 2* things were a bit different in two cases: *Uni Individual Operators* and *Single Direct Transformations*.

1) Uni Individual Operators: In the original study, as well as for *Machine 1* and *Machine 3*, Reactor performed worse than its counterparts when running the *uni individual operators* benchmark. In *Fig. 3*, which shows a comparison between the successfully reproduced results and the results from *Machine 2*, we can see both RxJava and Mutiny ahead in (a) whilst this is not the case in (b). This is especially true when comparing *Reactor_chain* with the corresponding benchmarks *RxJava_chain* and *Mutiny_chain* where Reactor is the best performing framework from being the worst when running on *Machine 2*. This applies to both the early versions of the frameworks as well as the latest ones.

2) Single Direct Transformations: The same pattern appears when running the *single direct transformation* benchmark. Again Reactor is catching up to the other frameworks on *Machine 2* whilst falling behind on *Machine 1* and *Machine 3*.

By using the analysis method mentioned in *Section V-C2* we came to the conclusion that not all findings generalise to different hardware and when comparing the results between versions the correlation between all frameworks stay the same. Some performance gains have been achieved by all frameworks in the later versions which can be expected from later releases.

C. ***RQ3*** *Identify bottlenecks in each framework and suggest adjustments/recommendations*

The Mutiny *mapToOne* benchmark in the test suite for multi individual operators was found to perform significantly worse than its Reactor counterpart in every test conducted across all machines and all framework versions. As can be seen in *Fig. 4* Reactor actually performs **264%** better on *Machine 1* and thus is the biggest difference across all benchmarks. This can of course be due to the optimization implemented in Reactor that is not present in the Mutiny framework but is still a red flag for potential bottlenecks. As stated in the footnote, RxJava's equivalent to *mapToOne* is currently causing a bug and thus could not be included.

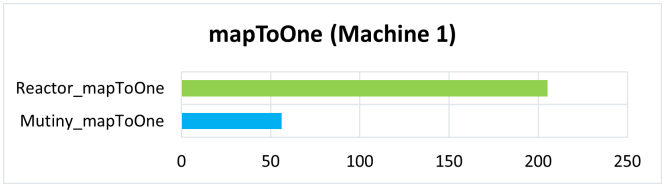


Fig. 4: *mapToOne comparison between Reactor and Mutiny.*

A potential performance bottleneck was exposed when analysing the multi individual operators results. This was the FlatMapManager.remove() method which took up 1% of the total execution time. The entire method can be seen in *Fig. 5*. As the entire method is wrapped in a synchronization block, meaning a lock is held that stops the execution of other threads to that particular code, it might be causing a bottleneck as synchronized regions of code may introduce software hangs [37].

[2]When the data was collected for the latest frameworks we investigated if the *mapToOne* bug in RxJava was fixed in the latest version. However, the bug is still present in *RxJava 3.1.4* since it still causes memory exhaustion.

```
final void remove(int index) {
    synchronized (this) {
        T[] a = inners.get();
        if (a != terminated()) {
            a[index] = null;
            offerFree(index);
            size.decrementAndGet();
        }
    }
}
```

Fig. 5: *Potential bottleneck in Mutiny.*

Refactoring this method to avoid synchronizing the whole code block could possibly increase performance.

D. ***RQ4*** *How does cyclomatic complexity and performance correlate in the evaluated frameworks?*

For the CPU-bound benchmark results one can observe that the Mutiny framework is, although lagging in a few benchmarks, performing on a similar level to the other frameworks. According to our measurements displayed in *Table I* these have considerably greater cyclomatic complexity than Mutiny. It can therefore be determined that in this case, cyclomatic complexity and performance have no correlation with each other for CPU-bound operations.

For I/O-bound operations Mutiny performs just as well as its counterparts, no correlation can therefore be determined to exist between performance and cyclomatic complexity for these benchmarks either.

VII. Discussion

The results of the reproducibility aspect of the paper indicate that the results of Ponge et al. [2] could indeed be reproduced with *Machine 3*, both for the CPU- and I/O-bound benchmarks, excluding text processing. However, the same conclusions can be drawn from the results of text processing as was made in the original study; no reactive framework outperforms another in I/O-bound benchmarks.

Looking at the results from the tests on different hardware we can conclude that the results actually differ depending on the machine. Our results show that Reactor performs better with fewer cores, however this result would have to be substantiated with more machines before any conclusion can be made, and a further investigation would have to be conducted to narrow down the reasons for why this is. The CPU-bound benchmark diagrams from *Machine 2* can be found on *https://github.com/anjohcer/benchmark/Diagrams* and can be compared to the diagrams found in the study conducted by Ponge et al. [2]. What can be concluded from our findings is that hardware seems to impact the performance of Reactor the most.

These results are interesting because Reactor had the worst performance of all frameworks on both *uni individual operators* and *single direct transformations* but performed better and even outperformed both RxJava and Mutiny in some benchmarks.

A lack of documentation in open source projects prevents developers from contributing and helping projects grow, when the time came to correct the potential bottlenecks discovered in Mutiny we were faced with poor documentation and an abundance of single-letter variable names. Research shows that professional developers with years of experience have trouble understanding methods with poor variable naming [38] and for us the experience was no different. Furthermore, having words as variable names as opposed to letters or abbreviations leads to faster comprehension of the code [39], this is something that we believe should be implemented in the Mutiny code base to encourage more contributions.

Including comments for methods has been proven to increase the change ratios and bug fixes for open source projects [40], and including documentation of the code greatly increases program comprehension [41]. What needs to be mentioned here is that not all single-letter variables are bad, and that sometimes documentation might be superfluous. In a study conducted by Beniamini et al. [42] a conclusion was made that some single-letter variables are common and okay, for example *i* as a loop index.

However, when looking in the class where our exposed bottleneck resides, namely *MultiFlatMapOp.java*, we were met by only single-letter variable names, and no documentation. We spent a lot of effort trying to decipher the code but to no avail and had to abort our goal of correcting the bottleneck. There are of course many reasons as to why one might opt to not include any documentation and if you are single-handedly taking care of a project, descriptive variable naming might not be necessary. However, this might scare away potential contributors looking for a project to work on.

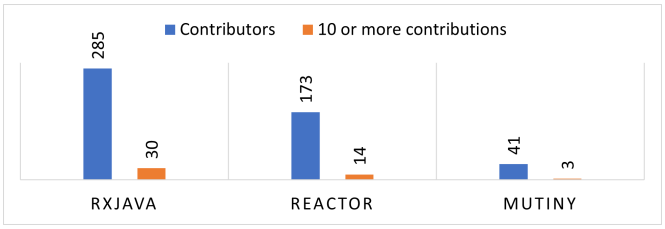


Fig. 6: *Amount of unique contributors to each framework and the amount of contributors with 10 or more contributions.*

Looking at both RxJava and Reactor we found descriptive and comprehensive documentation as well as method comments together with informative variable names. We could also see that said frameworks have far more contributors than Mutiny as can be seen in *Fig. 6*[3] which shows the amount of unique contributors of each framework as well as contributors that have contributed 10 times or

[3]Mutiny has 6 contributors with 10 commits or more. However, 3 of these are bots and thus are not included in the diagram.

more. Since we have stated that bugs have a higher fix ratio when documented one can understand why far fewer hotspots where discovered in RxJava and Reactor as opposed to the hotspots found in Mutiny as these are likely found and accounted for in a faster pace.

To understand possible reasons as to why cyclomatic complexity does not show a correlation with performance in this instance we need to consider what cyclomatic complexity is actually telling us about an application. Two algorithms that perform the same task can have different CC, this could be due to the fact that the developer did not design the most optimal algorithm, or the algorithm contains optimizations that may or may not create more overhead than the performance gains make up for. Therefore we can not make any assumptions about performance by only considering cyclomatic complexity.

Cyclomatic complexity, as previously stated, expresses the number of linearly independent paths a program can make, but this does not tell us anything about the time complexity. Consider two sorting algorithms, selection sort[4] with an average time complexity of $O(n^2)$ and CC of 5 and merge sort[5] $O(nlog(n))$ with CC of 9. The execution time of selection sort increases exponentially whilst merge sort has a close to linear increase. However, selection sort performs better than merge sort for smaller data sets, and thus makes it impossible to make any assumptions about performance without first determining the input. Therefore, knowing the CC score does not help us make an informed assumption about the algorithms performance.

A. Limitations

We acknowledge some limitations in our work. The test suite developed in the previous study could be updated to give greater code coverage. We were unable to source a tool for cyclomatic complexity calculations that measures CC on a method level. Instead, the cyclomatic complexity score applies to the entirety of the framework. We were unable to comprehend the code where the potential bottleneck resided due to the lack of any documentation and meaningful variable names, which is a barrier for developers to contribute in open source software [40], thus we were unable to make any successful corrective adjustments.

B. Ethical/Societal Considerations

There are some possible conflicts of interests that should be stated. Julien Ponge and Clement Escoffier, two of the authors of *"Analysing the Performance and Costs of Reactive Programming Libraries in Java"* which is the study being reproduced, are the main contributors to Mutiny with 402 and 651 contributions respectively[6] [16]. Since Mutiny is one out of three frameworks being investigated this should be known. However, the benchmarks used are publicly available and we found no evidence of bias in our research.

[4]https://gist.github.com/codelance/4186240
[5]https://gist.github.com/codelance/4186238
[6]as of 2022-06-05.

VIII. Conclusions

We can conclude that the results of *"Analysing the performance and costs of reactive programming libraries in Java"* [2] were indeed reproducible with the sole exception of one benchmark, the I/O-bound *Text processing* benchmark, which showed different results. Instead of all reactive frameworks being outperformed by the imperative baseline our findings showed the opposite. However, no reactive framework had any performance advantage in these tests which is the same conclusion made by Ponge et al. [2].

Furthermore, our results from *RQ2* indicate that these results may differ between hardware. *Machine 1* and *Machine 3* had similar specifications - as can be seen in *Table II* with the major difference being the amount of cores where *Machine 1* had 8 and *Machine 3* had 4 - and also had very similar results. However, *Machine 2* showed different correlation between frameworks in two out of six benchmarks: *uni individual operators* and *single direct transformations* where Reactor went from the worst performing framework in the original study to being equal or better in the results from *Machine 2*.

From the generated JFR-files it could be determined that bottlenecks seem to be present in Mutiny but not as obvious in RxJava or Reactor. One of the reasons for this could be the lack of documentation in Mutiny which potentially hinders contributors from getting involved, thus less resources are available to tackle bugs and performance issues. We were unable to correct the potential bottleneck we discovered due as we were unable to comprehend the code due to poor variable-naming and documentation.

Finally, we could find no correlation between cyclomatic complexity and performance. Mutiny, which has the lowest cyclomatic complexity score of all frameworks investigated, outperformed the other frameworks for some benchmarks whilst performing worse for others, without a pattern it is not possible to deduce a correlation.

A. Future work

For future research a new test suite that covers more of the frameworks could be developed. Furthermore, an attempt to reproduce the text processing benchmarks to conclude if they indeed are outperforming the imperative baseline as well as a more thorough investigation into Reactor's performance on machines with fewer cores. A more in-depth investigation into potential bottlenecks would also be beneficial, but would require more experience with reactive frameworks.

As stated in the limitations of our work no cyclomatic complexity tool that measure on a method level exists as far as we know. A different approach for investigating the correlation between cyclomatic complexity and performance could be to acquire the CC score for each and every method used in each benchmark. That way a comparison could be made based on the CC score for each benchmark instead of the CC score for the entire framework.

REFERENCES

[1] J. Ponge and M. Little, "Scalability and resilience in practice: Current trends and opportunities," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019, pp. 267–2670.

[2] J. Ponge, A. Navarro, C. Escoffier, and F. Le Mouël, "Analysing the performance and costs of reactive programming libraries in java," in *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, ser. REBLS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 51–60. [Online]. Available: https://doi.org/10.1145/3486605.3486788

[3] reactive-streams-jvm. [Online]. Available: https://github.com/reactive-streams/reactive-streams-jvm

[4] C. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *Communications of the ACM*, vol. 59, no. 3, p. 62–69, Feb 2016.

[5] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating performance bottleneck detection using search-based application profiling," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 270–281. [Online]. Available: https://doi.org/10.1145/2771783.2771816

[6] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Comput. Surv.*, vol. 48, no. 1, jul 2015. [Online]. Available: https://doi.org/10.1145/2791120

[7] L. D. Toffola, M. Pradel, and T. R. Gross, "Synthesizing programs that expose performance bottlenecks," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. Vienna Austria: ACM, Feb 2018, p. 314–326. [Online]. Available: https://dl.acm.org/doi/10.1145/3168830

[8] T. Yu and M. Pradel, "Syncprof: detecting, localizing, and optimizing synchronization bottlenecks," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, Jul 2016, p. 389–400. [Online]. Available: http://doi.org/10.1145/2931037.2931070

[9] R. Nair and T. Field, "Gapp: A fast profiler for detecting serialization bottlenecks in parallel linux applications," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, Apr 2020, p. 257–264. [Online]. Available: http://doi.org/10.1145/3358960.3379136

[10] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating performance bottleneck detection using search-based application profiling," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, Jul 2015, p. 270–281. [Online]. Available: http://doi.org/10.1145/2771783.2771816

[11] rebls21-paper-benchmarks. [Online]. Available: https://github.com/jponge/rebls21-paper-benchmarks

[12] N. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, p. 675–689, Oct 1999.

[13] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, aug 2013. [Online]. Available: https://doi.org/10.1145/2501654.2501666

[14] Rxjava. [Online]. Available: https://github.com/ReactiveX/RxJavas

[15] Reactor. [Online]. Available: https://projectreactor.io/

[16] Mutiny. [Online]. Available: https://smallrye.io/smallrye-mutiny/

[17] reactive-manifesto. [Online]. Available: https://www.reactivemanifesto.org/

[18] F. Dobslaw, M. Vallin, and R. Sundström, "Free the bugs: Disclosing blocking violations in reactive programming," 2020, pp. 177–186, ISSN: 2470-6892.

[19] A. Benítez-Hidalgo, A. J. Nebro, J. J. Durillo, J. García-Nieto, E. López-Camacho, C. Barba-González, and J. F. Aldana-Montes, "About designing an observer pattern-based architecture for a multi-objective metaheuristic optimization framework," in *Intelligent Distributed Computing XII*, J. Del Ser, E. Osaba, M. N. Bilbao, J. J. Sanchez-Medina, M. Vecchio, and X.-S. Yang, Eds. Springer International Publishing, 2018, vol. 798, pp. 50–60, series Title: Studies in Computational Intelligence. [Online]. Available: http://link.springer.com/10.1007/978-3-319-99626-4_5

[20] reactive-streams-commons. [Online]. Available: https://github.com/spring-attic/reactive-streams-commons

[21] T. J. McCabe, "A complexity measure," in *Proceedings of the 2nd international conference on Software engineering*, ser. ICSE '76. Washington, DC, USA: IEEE Computer Society Press, Oct 1976, p. 407.

[22] Sloc, cloc and code. [Online]. Available: https://github.com/boyter/scc

[23] S. K. S. Kumar, S. P. Kulyadi, P. Mohandas, M. J. S. Raman, and V. S. Vasan, "Computation of cyclomatic complexity and detection of malware executable files," pp. 1–5, 2021.

[24] M. J. Shepperd and D. C. Ince, "A critique of three metrics," *J. Syst. Softw.*, vol. 26, pp. 197–210, 1994.

[25] S. Alkatheri, S. Abbas, and M. Siddiqui, "A comparative study of big data frameworks," *International Journal of Computer Science and Information Security*, p. 8, 01 2019.

[26] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: an empirical study," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, May 2016, p. 61–72. [Online]. Available: http://doi.org/10.1145/2884781.2884829

[27] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, May 2014, p. 1013–1024. [Online]. Available: http://doi.org/10.1145/2568225.2568229

[28] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," ser. PLDI '12, New York, NY, USA, Jun 2012, p. 77–88. [Online]. Available: http://doi.org/10.1145/2254064.2254075

[29] G. Gill and C. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1284–1288, 1991.

[30] U. Farooq, Abubakar, and A. B. Aqeel, "A meta-model for test case reduction by reducing cyclomatic complexity in regression testing."

[31] D. D. Hutajulu, M. E. S. Simaremare, Y. S. Pangaribuan, and A. R. Ginting, "Measuring programmer quality from complexity point of view," in *2021 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS*, 2021, pp. 262–266.

[32] Ubunutu. [Online]. Available: https://ubuntu.com/

[33] Java flight recorder. [Online]. Available: https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htmJFRUH172

[34] visualvm. [Online]. Available: https://visualvm.github.io/

[35] Cyclo. [Online]. Available: https://github.com/sarnold/cyclo

[36] Jarchitect. [Online]. Available: https://www.jarchitect.com/

[37] M. D. Shah and S. Z. Guyer, "Iceberg: Dynamic analysis of java synchronized methods for investigating runtime performance variability," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ser. ISSTA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 119–124. [Online]. Available: https://doi.org/10.1145/3236454.3236505

[38] E. Avidan and D. G. Feitelson, "Effects of variable names on comprehension: An empirical study," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 55–65. [Online]. Available: http://ieeexplore.ieee.org/document/7961504/

[39] J. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter identifier names take longer to comprehend," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 217–227.

[40] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Empirical analysis of change-proneness in methods having local variables with long names and comments," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2015, pp. 1–4, ISSN: 1949-3789.

[41] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," vol. 44, no. 10, pp. 951–976, 2018, conference Name: IEEE Transactions on Software Engineering.

[42] G. Beniamini, S. Gingichashvili, A. K. Orbach, and D. G. Feitelson, "Meaningful identifier names: The case of single-letter variables," in *Proceedings of the 25th International Conference on Program*

Comprehension, ser. ICPC '17. IEEE Press, 2017, p. 45–54. [Online]. Available: https://doi.org/10.1109/ICPC.2017.18

APPENDIX 1: TIME PLAN

B. Planned timeplan

Week 11: Find research question and read up on subject.
Week 12 - 14: Continue getting familiar with the subject. Start writing introductory sections; abstract, introduction, background, purpose & contributions, related works.
Week 15: Setting up testing environments, developing and writing methodology and running benchmarks.
Week 16 - 17: Running benchmarks with profiler, analysing results.
Week 18 - 19: Analysing results for reproducibility and identifying bottlenecks.
Week 20: Write results, discussion, and conclusions.
Week 21: Prepare for presentation, edit text.
Week 22: Final adjustments.

C. Actual timeplan

Week 11-12: Find research question and read up on subject.
Week 13-15: Continue getting familiar with the subject. Start writing introductory sections; abstract, introduction, background, purpose & contributions, related works.
Week 16: Setting up testing environments, developing and writing methodology and running benchmarks.
Week 17-18: Running benchmarks with profiler, analysing results.
Week 19-20: Analysing results for reproducibility and identifying bottlenecks. Writing results, discussion, and conclusions.
Week 21: Prepare for presentation, edit text.
Week 22: Final adjustments.

APPENDIX 2: CONTRIBUTIONS

Most work was done in close collaboration, and so divvying the contributions of each team member is a difficult exercise, but for these areas however we had roughly the following division of contributions.

- Setting up testing environments: Christian 80% Anton 20%.
- Analysing results of benchmarks, RQ1-2: Christian 60% Anton 40%.
- Diagrams/Tables: Christian 100%.
- Analysing profiler files RQ3: Christian 20% Anton 80%.