Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Detecting Software Performance Anti-patterns from Profiler Data

Alexander Bran

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Dr.-Ing. André van Hoorn Catia Trubiani, Ph.D. (GSSI, Italy) Alberto Avritzer, Ph.D. (Sonatype, USA)
Commenced:	January 1, 2017
Completed:	July 17, 2017
CR-Classification:	I.7.2

Abstract

Nowadays performance is very important in the software business. For example, if the search of an online shopping website takes too long, the customers won't buy and the web site loses money. For measuring and optimizing performance, there are various solutions available. In this thesis, the focus is set on so-called profilers, more precisely on profiler data from YourKit, which is one of the leading tools in this segment. Profilers are used in development and can monitor all runtime data during the execution of a program. It measures for example the response time and saves the exact CPU and memory usage at any given time.

The main aspect of this thesis is to analyze and detect different performance anti-patterns in the profiler's data export. Anti-patterns are the opposite of programming patterns, which are capturing expert knowledge of ”best practices“ in software design. Anti-patterns, on the other hand, document common mistakes made during software development. The goal is to automatically detect performance anti-patterns in the profiler data and show what the problem is and where it occurs. Therefore, this research is conducted with a company operating in the open-source domain. Together with them, we made a case study about the manual detection of anti-patterns in profiler data from load tests. This data is used in order to develop analysis strategies for the detection of anti-patterns with the help of a program called PADprof, which is also been developed in this thesis.

The results show that most of the selected performance anti-patterns can be automatically detected in the available data. Nevertheless, more tests need to be conducted in order to evaluate if the anti-patterns can be detected in different data from other systems.

Kurzfassung

Performance ist heutzutage ein sehr wichtiges Thema bei der der Softwareentwicklung. Niemand kauft etwas in einem Onlineshop ein, wenn das Suchen eines Artikels mehrere Sekunden benötigt. Deswegen wird bei der Entwicklung immer mehr auf diesen Aspekt geachtet. Um die Performance sicherzustellen, gibt es viele Wege. In dieser Arbeit liegt der Fokus auf dem Instrumentieren des Systems mit einem so genannten „Profiler“, in unserem Fall YourKit — einen der führenden in diesem Segment. Profiler werden während der Entwicklung eingesetzt und loggen performancerelevante Daten, wie zum Beispiel Antwortzeit und Prozessor-/Speicheraustung im Zusammenhang mit der Programmausführung mit. So ist es möglich, detaillierte Informationen über alle Performance-Aspekte zu erlangen. Mit Hilfe dieser Daten kann die Geschwindigkeit des Systems nach und nach verbessert werden. Jedoch geschieht das Analysieren dieser Profilerdaten noch vollkommen händisch und ist deswegen ziemlich aufwändig.

In dieser Arbeit soll dieser Prozess automatisiert werden. Dabei liegt der Fokus auf dem Erkennen und Analysieren von sogenannten „Performance-Anti-Patterns“ aus den Profiler-Daten. Anti-patterns sind das Gegenteil von Programmier-Patterns, welche gute Wege Software zu entwickeln darstellen. Anti-Patterns dagegen beschreiben häufig gemachte Fehler während der Softwareentwicklung.

Das Ziel dieser Arbeit ist es, Performance-Anti-Patterns automatisch in Profiler-Daten zu erkennen und darzulegen wo und was das Problem ist. Dafür werden Profiler-Daten aus einer Fallstudie einer im Open-Source-Bereich arbeitenden Firma analysiert. Mit Hilfe der Daten, wird außerdem in dieser Arbeit ein Programm namens PADprof entwickelt, welches die Daten aus YourKit analysisert, um die Performance-Anti-Patterns zu erkennen. Die Ergebnisse der Arbeit zeigen, dass die meisten der untersuchten Performance-Anti-Patterns automtatisch in den uns vorliegenden Daten erkannt werden können. Trotzdem müssen noch mehr Tests durchgeführt werden, um sicherzustellen, dass dies auch auch mit Daten aus anderen Systemen funktioniert.

Contents

List of Figures

List of Tables

List of Acronyms

CTH	Circuitous Treasure Hunt
EP	Extensive Processing
GUI	graphical user interface
PADprof	Performance Anti-pattern Detection from profiling data
SPA	Software Performance Anti-Pattern
UP	Unnecessary Processing
WCS	Wrong Cache

List of Algorithms

Chapter 1

Introduction

This thesis presents an approach for automatic detection of anti-patterns from profiler data.

The intention of this chapter is to explain why such an approach is required. To do this, the chapter starts with a short motivation in Section 1.1. Afterwards the goals of this research are presented in Section 1.2. The developed approach is explained in Section 1.3. Lastly the structure of this document is outlined in Section 1.4.

1.1. Motivation

Nowadays, it is natural that every company monitors its software to check whether the performance is good and if everything runs as expected. But what happens when a problem occurs, like the response time of a website is higher than usual?

Today, this can be seen in peaks of a graph in a monitoring software, like an Application Performance Monitoring tool [HHOM17]. After a company noticed that something is wrong with their system, the developers need to find the problem. This is usually very difficult and can only be achieved with the help of software tools. Aside from the Application Performance Monitoring tools, there are also so-called profilers which instrument the execution of an application by recording performance data like CPU-utilization, memory usage, response times or a call tree of methods from the application. Normally these tools are used in combination with load tests.

With the help of the data from a profiler, software architects can look for the problem. Today, this is done mainly manually by having a look at a big chunk of gathered data in the profiler which can cost a lot of time. In order to speed up and simplify this process, this research presents an approach of automatically detecting Software Performance

Anti-Patterns (SPAs) in profiler data. An anti-pattern describes a poor recurrent approach to design problems which may have a negative impact on particular software quality attributes like the performance [Wer13]. For SPAs, there are also common solution strategies documented, which can help the developer fix the specific problem.

With the help of such a tool, the software architect would be relieved, because the problem is found automatically. This is particularly good, because in general a developer team consists of just one architect and many developers. Also, the automatic analysis is much faster than the process of doing it manually. In order to develop such an analysis tool, we work together with an innovative company operating in the open-source ecosystem domain. Together with them we did a case study about SPAs in profiler data and providing data and results of the tests from the case study.

1.2. Goals

The goal of this thesis is to develop an approach for automatic detection of SPA, which should then, in the end, lead into a software that can read the different profiler data files and later, after some analyzing, shall output which methods can cause a performance problem and what SPA is occurring there.

In order to develop a software that can automatically detect SPAs, the following sub goals need to be fulfilled:

Evaluation of YourKit: The contributing company in this thesis is using the profiler YourKit for their analysis. Therefore, it is important to research what data YourKit [profiler16] can provide. How and in which formats it can be exported is also important. For the future it is also important to know which anti-patterns can be detected and which cannot, because there can be either not enough or not the proper data for detecting the anti-pattern. After that, it is a logical step to analyze the different data formats to know how to implement an importer for those later on.

Evaluation of the case study: The second goal is to evaluate the case study from the company to get insights which anti-patterns were investigated and how they were detected manually.

Anti-pattern research: Another goal is to research which SPAs exist and if they can be detected with the available data. Also, there is the possibility that different analysis processes are needed for detecting them. Certain ones could need more than just one measurement file for the detection, because they need historical data.

Developing concepts for detecting anti-patterns: The next goal is to develop a way of how the specific anti-patterns can be detected with the available data. For that it is necessary to think of boundaries which need to be exceeded in order to assign the data to a specific anti-pattern.

Implementation: The next goal is the implementation of the software, which needs to read the exported files from YourKit and then check the boundaries made in the previous goal in order to identify whether there is an SPA located. In the end of the analysis process there should be an output with the problem method name and which anti-patterns causes the performance decrease.

Evaluation of the implementation: After the implementation of the concepts, the system needs to be tested with different kinds of data to evaluate the quality of the detection. For this purpose, it is very helpful that we can access real data from a running development. Also, we can run tests on the system to generate data we need to evaluate the software properly.

1.3. Overview of the Approach

The base of this research is a case study which is presented in Section 3.2. In this study, the company investigated profiler data in order to find performance problems. After one problem was found, a Software Performance Anti-Pattern was assigned and the problem was fixed. Afterwards, they searched for other performance flaws in the new iteration. Overall, four iterations were made in order to improve the performance of the system. From every iteration, the profiler data was saved for this research. The approach for the automatic detection was to look at the given data with the help of the profiler YourKit and understand how the software architect diagnosed the problem manually in the study. This was the first step to elaborate how an automatic detection of anti-patterns with the data recorded by the profiler could work. The detection has two stages: First, identify that there is a problem present in the data (for example method times are higher than usual), second assign a Software Performance Anti-Pattern to it.

To achieve this, our approach was to develop a software which can read the different files that can be exported with YourKit (Section 3.3.2) and then analyze them. In order to detect the SPAs, it was necessary to develop rules for each of the anti-patterns. For this purpose, we worked together with a performance engineer from the company that made the case study in order to get insights of the system and the data. Also, he supported us with new data we needed to test our approach. Also, Mrs Trubiani helped us a lot by examining the developed analysis methods and contributing her expertise to our ideas of how to detect the different SPAs.

To control, if our detection works as we wanted, we compared the saved profiler data of the different iterations from the case study to each other. Additionally, data that has no problems or anti-patterns in it, called baseline data here, was used for the comparison. With the help of this baseline data it was possible to compare the "anti-pattern infested“ data with normal “good” data. We noticed that comparing different snapshots to each other, either with baseline data or just historical data is the only way to detect if there is a performance problem, because just from one file it is not possible to say if some behavior is abnormal. The reason is that it is not clear how the “normal” state of the investigated system looks like.

Another approach was to compare data from different loads, but with the same software version. This could be an analysis variation where no historical or baseline data would be needed to detect a certain anti-pattern, because a comparison between the different loads could be made. But for this approach, very limited data was available, wherefore the focus was set on the comparison between data of different software revisions.

1.4. Document Organization

This thesis is structured as follows:

Chapter 2 – Foundations and State of the Art: In this chapter, all the investigated Software Performance Anti-Patterns as well as the related work is presented.

Chapter 3 – Case Study and Research Design: Here, the case study on which this research bases is explained. Furthermore, the profiler YourKit is presented with details about the graphical user interface and the exportable data.

Chapter 4 – Automated Anti-Pattern Detection: This chapter is the main chapter, because most of the work is noted here. The analysis process of every anti-pattern is explained as well as their limitations. Also, the developed tool of this research is presented in this chapter.

Chapter 5 – Evaluation: In chapter 5 the evaluation of the developed approach is conducted. Research questions are presented as well as the results and a discussion of them.

Chapter 6 – Conclusion: In the last section, a summary of the complete thesis can be found. Furthermore, possible future work is presented.

Appendix A – Example data: The appendix holds exported example data from YourKit.

Tool available at github [TBH+17].

Chapter 2

Foundations and State of the Art

In this chapter, the Software Performance Anti-Patterns that are investigated in this research are introduced in Section 2.1. Furthermore, the releated work is presented in Section 2.2

2.1. Software Performance Anti-Patterns

An anti-pattern describes a poor recurrent approach to design problems which may have a negative impact on particular software quality attributes [Wer13]. In terms of performance anti-patterns, the affected quality attribute is the performance of the software.

Circuitous Treasure Hunt (CTH)

This anti-pattern is originally focused on database statements, where data is retrieved from a first table to get data from a second table, to then receive data from a third table [SW00a]. This can be arbitrarily continued until the “ultimate results” are obtained, which leads into long processing times to get the requested data. To solve this problem the data organization needs to be refactored. In our case, the CTH is used for describing a problem within the code itself. It is not related to database requests, but for calling other methods. This means that the Circuitous Treasure Hunt exists when a method calls too many other methods before it is completely executed.

Extensive Processing (EP)

In this anti-pattern, one processor or thread is blocked because of a long running process which monopolizes it [SW03b]. The processor is removed from the pool, this is particularly problematic if the extensive processing is on the processing path that is executed for the most frequent workload. For solving this problem, identifying the processing steps that may cause the slow downs is required. Then these steps can be delegated to processes, that do not restrain the fast path.

Wrong Cache Strategy (WCS)

Using cache is not always beneficial in terms of performance [Wer13]. If a cache is used in inappropriate situations, it may lead to an increased pollution of the memory and, thus, may result in hiccups which are caused by garbage collections. The WCS can be found in any abstraction layer (architecture, design and implementation). To solve this problem, the caching strategy needs to be rethinked.

Unnecessary Processing (UP)

Unnecessary processing is processing that is executed, but is either not needed, or not needed at that time [SW03b]. This is particularly problematic if this is done on the processing path that is executed most of the times. There are several solutions to this anti-pattern. If the processing is completely nonsensical, it can be deleted. Otherwise a rearrangement of the processing steps or a restructuring of the processes, where for example a background task can do the unnecessary processing, could help.

2.2. Related Work

Software Performance Anti-Patterns as well as strategies to avoid or temper performance problems were firstly described by Smith and Williams in [SW00b] and were later refined in [SW02; SW03a]. There are also some anti-patterns that are directly related to Java which is the development platform of our case study. Additionally, some anti-patterns are related to specific Java technologies, such as Enterprise Java Beans [TCL03], Java EE [TCL03], and Java multithreading [HAT+04].

There are different ways of detecting performance problems in a system. Very popular are static analysis tools like PMD [RAF04] or FindBugs [APM+07; HP07]. They can find potential root causes for performance anti-patterns like resource leak which can be the cause for the “Ramp”. However, just a few of the found issues really caused a serious problem [AP10]. The problems are often found during tests in the development process. Also, these tools have a high number of false positive rate. Therefore the developers do not trust those tools anymore. Additionally, the real problems can get lost in the noise [BBC+10].

However, most of the performance issues cannot be detected statically, because they can be only observed at runtime. There is an automatic anti-pattern detection bachelor thesis from Hidiroglu [HG16] for Application Performance Management (APM) traces. With the same data Heger et al. [HHO+16] are discovering recurring performance issues with diagnoseIT. APM tools are used during production and are not designed for load testing or profiling. Also, there are other papers that tackle this problem in different ways. Like Wert et al.’s [WOHF14] elaboration of automatically detecting so-called “communication performance anti-pattern” such as “Empty Semi-Trucks”. They also provide heuristics for the Circuitous Treasure Hunt anti-pattern. Furthermore, there is a framework for detecting design and deployment anti-patterns [Par05]. The performance anti-patterns are detected there from summarized data, using a rule-engine approach. There is also an approach of modeling and detecting performance anti-patterns has been done in the past [Tru11] [CDT14]. We are in touch with Ms Trubiani and we moreover flew to Italy to work together for this research. At the time this paper was written, nobody has done automatic detection of anti-patterns in profiler data yet.

Chapter 3

Case Study and Research Design

In this chapter the fundament of this research is presented. Furthermore, an overview of the available data and the profiler YourKit is given. Lastly the current method of finding performance problems is presented. Section 3.1 explains how the research was driven. In Section 3.2 the case study from where this research started is presented. YourKit is described in Section 3.3, where the profilers' interface is shown with screenshots and the different data sets are investigated. Last but not least, the current analysis process is explained in Section 3.4.

3.1. Methodology

The base of this research is the case study which was conducted together with a company. For every profiled load test experiment (see the next section for more information), we had access to the recorded profiler data. The first step was then to reproduce which values the software architect investigated in order to assign the specific anti-patterns manually. This was done by looking at every available value in the profiler YourKit which is presented in Section 3.3. Because in every experiment, a certain anti-pattern was fixed, we also compared the different snapshots to each other in order to identify deviations in the data. A snapshot is the data format from YourKit where all the recorded data is saved. After some time, we also had access to "baseline data", which represents the profiled system without any problems, respectively without anti-patterns inside. We were able to compare every "problem data" to it in order to detect differences. With this insights, we searched closer for deviations that can be detected automatically. After that we developed rules for detecting the anti-pattern for which we used different values from the data of YourKit. In order to test the detection, many tests were conducted. The tests, thoughts, and data sets that are used for the analysis of the three anti-patterns can be found in Section 4.3.

3.2. Case Study

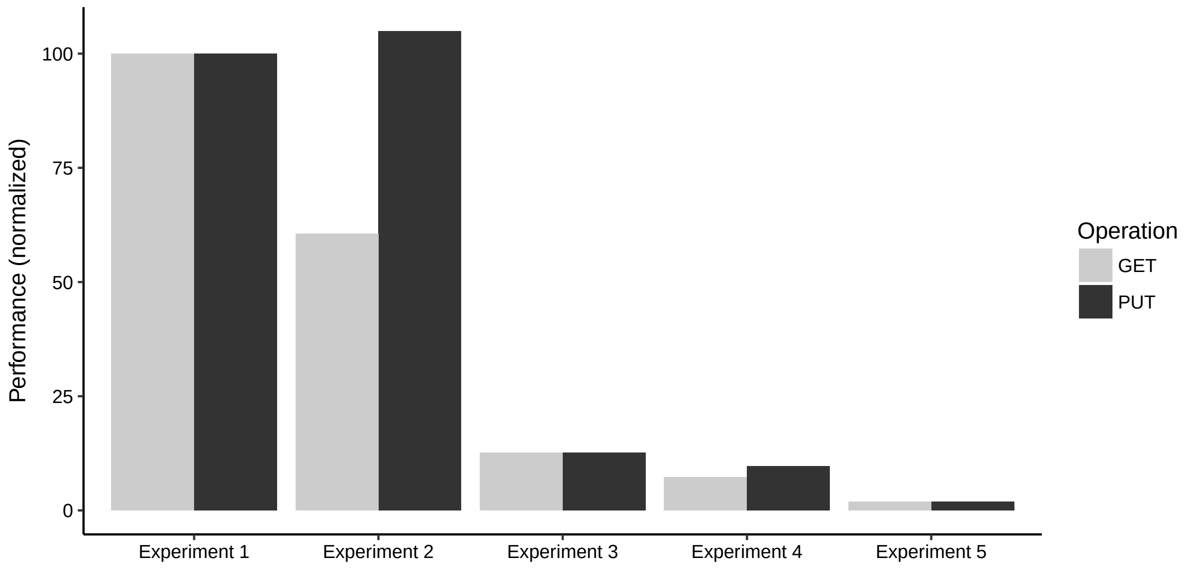


Figure 3.1.: Bar chart with the normalized performance for Http Gets and Puts

Figure 3.1 shows the results of the case study from a software component repository system provided by contributing company. The system provides services to upload and download software artifacts via an HTTP-based API. In the study, we distinguish only between GET and PUT operations. The performance requirement to be analyzed during the load tests is that requests must not exceed a response time of 100 milliseconds on average. The system is deployed to an environment for executing performance tests, in this case to Amazon Web Services. Apache JMeter is used as the load testing tool, and YourKit is used as the profiler tool.

- **Experiment 1:** The first experiment consisted of running a baseline system. In the bar chart in Figure 3.1 the normalized performance was rated to 100 for the two analyzed operations. The delay in this test of the GETs and PUTs was significantly larger than the allowed average of 100 ms. The output of this experiment was a YourKit snapshot that was manually analyzed in order to assign an anti-pattern to it. This manual analysis process is described in Section 3.4. The refactoring that was done in order to improve the performance of this experiment consisted of inverting the order of parsing expressions and checking for permission per user. This problem was mapped to the Circuitous Treasure Hunt (CTH) anti-pattern.
- **Experiment 2:** After the first refactoring, the load test execution produced an average normalized performance that was rated as 60.60 for GETs and 104.93

for PUTs. To boost performance, the refactoring here was done by correcting the permission resolver cache method. The manual investigation of the profiler data from this snapshot mapped the Extensive Processing (EP) anti-pattern.

- **Experiment 3:** The fixed software version of Experiment 2 produced an average normalized performance of 12.70 for GETs and 12.63 for PUTs. The PUT delay decreased in comparison to the previous experiment by 92.3 %, respectively 47.90 % for the GETs. The action implemented for refactoring was a change in a method specification from pass by value to pass by reference, which prevented the repeated execution of serialization and de-serialization. The experimental data was manually mapped to the Wrong Cache (WCS) anti-pattern.
- **Experiment 4:** Here, the normalized performance for GETs is 7.28 and for PUTs 9.72. In this Experiment, the Unnecessary Processing (UP) anti-pattern was mapped. The solution to this performance problem was to remove a method that was marked as unnecessary.
- **Experiment 5:** In the last experiment the normalized performance was rated as 1.96 for GETs and 1.89 for PUTs. Throughout the experiments the normalized response time was 50 times faster. The performance of experiment 5 was significantly better than the objective performance requirements.

The analysis of the different anti-patterns data can be found in Section 4.3. Except for the UP SPA, because it requires that the analysis tool has knowledge about the system in order to distinguish between necessary and unnecessary processing/methods. This is because of the definition of the anti-pattern itself which was declared in Section 2.1. This knowledge could not be provided with our approach of the automatic detection. For this purpose, an artificial intelligence or something similar would be necessary.

3.3. YourKit

This chapter is about the profiler YourKit. Here it is described what profiling is and why YourKit is used exclusively in this research. Afterwards it is explained which data can be used for the automatic anti-pattern detection and how it should be exported. Lastly, some analysis is presented which can be made inside the profiler itself.

In Section 3.3.1, a short overview is given about what a profiler is and why YourKit is used in this research. Also in this part, it is explained how the analysis data is gathered. In Section 3.3.2 a summary of the available data is given, which is then described with the help of multiple screenshots from Section 3.3.3 to Section 3.3.7. These chapters present what YourKit can do and what data it provides. Additionally, it is explained which data could be useful for the analysis. In Section 3.3.9, the export process of the different data sets is explained in order to use the automatic detection. There an automatic, as well as a manual way, of doing the export process is presented. Section 3.3.10 deals with the question which kind of analysis is already built into YourKit and where the limitations from it are in comparison to the automatic anti-pattern detection.

3.3.1. General information

YourKit is a so-called "profiler" which is available for Java and .NET [profiler16]. Profilers are used for dynamic program analysis by measuring, for example, the space or the time complexity of a program. YourKit supports a variety of monitoring options, like memory, CPU and database monitoring. In this thesis, YourKit is the only profiler which will be investigated, because this it is used in the case study by the company. All the profiler data that will be used for this research is thankfully provided by this company. So there is no need to investigate other profilers, because the available data is limited to YourKit only. Gathering data is achieved by profiling the application. There are different types of profiling:

- So-called telemetry data like CPU usage or memory consumption is available as soon as the profiler is connected to the application. The standard buffer for this kind of data is one hour, but can be customized by changing the startup option [YourKit Documentation, Section 5.1]. This data gives an overview of for example CPU and memory utilization.
- The next type is CPU profiling, with which most of the data we use is collected. There are several settings for this recording type [YourKit Documentation, Section 5.2]. All of our data is recorded with the "sampling" option. Here, the profiler periodically queries stacks of running threads in order to estimate the slowest part of the code. Therefore it measures the CPU time spent with the different methods. This is also the best option in order to discover performance bottlenecks, because it adds virtually no overhead to the profiled application. There is also "Tracing" which gathers more data like the method invocation count. But, also noticeably slows down the profiled application, because YourKit needs to execute special code on each enter to and exit from the methods. Additionally, the CPU times are affected too because of this circumstance. "Call counting" is the opposite of that, because it has almost zero overhead. But it just provides a plain method list with method invocation counts. All the profiling options, including CPU profiling, need to be manually activated by the user. We cannot communicate the exact settings of the CPU sampling due to confidential reasons

- The last profiling method that is used in this research, is the “Monitor profiling” in which the blocked and waiting threads are recorded [YourKit Documentation, Section 11]. This profiling needs to be started manually, too.
- Additionally, there is the possibility to record so called “Memory snapshots” by doing “Memory profiling” [YourKit Documentation, Section 8]. This kind of snapshot includes more memory-related data such as loaded classes, existing objects and references between objects.

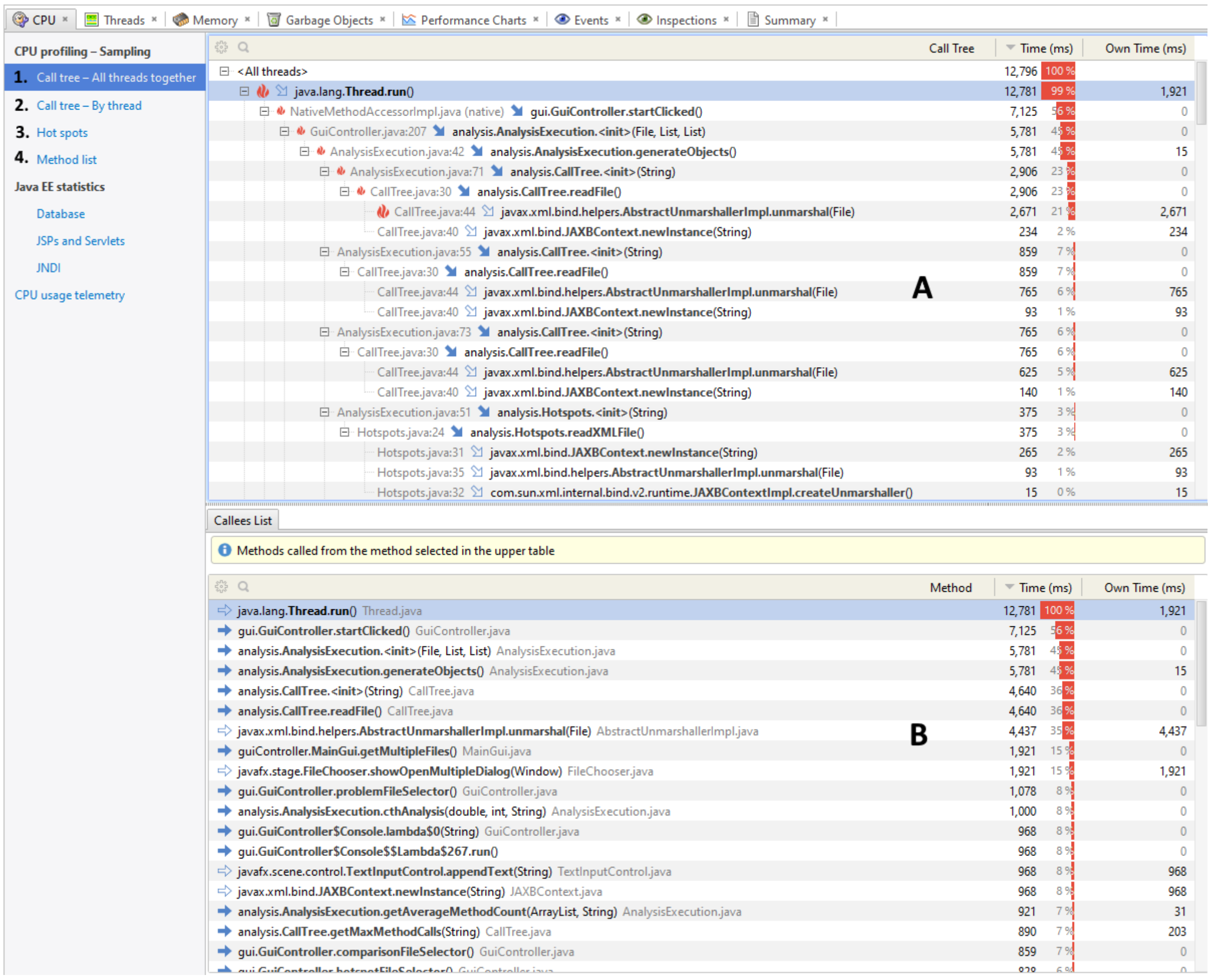
3.3.2. Available data

Data can be exported from YourKit in XML, CSV, HTML, plain text, and zip formats. In this research, the focus is set on the XML and CSV data outputs, because these are the easiest to read automatically. Also, YourKit saves its data in so-called “snapshots” which we are using to look at the data, measured at the collaboration company, with YourKit directly. This file format is proprietary to YourKit and gives, by reading it, access to all data. For future processing, the snapshot is loaded into YourKit and then XML or CSV data is exported. In this research we only use “Performance Snapshots”, but there are also "Memory Snapshots“ as described above. Also, there are multiple options to customize the snapshot within the profiler.

Using this profiler, not everything is exported at once in a big XML file. YourKit has at least one separate export file for all of their views within the profiler (overall 10 views). Here the most important exportable data of the different views is summarized. Important to note is, that the exported data looks exactly like the view in the graphical user interface. For example: If the call tree is not fully extended, then also the hidden methods are not included in the XML file. This is particularly problematic if the “Call tree - By thread” view shall be exported. To get all the data, it is necessary to do a right click on every displayed thread and select “Expand Node Fully”, which can be exhausting when many threads are measured. But there are ways around this problem, which we will present in Section 3.3.9. In the next sections, all the views of YourKit from which we use the data are presented.

3.3.3. CPU View

This CPU view is divided in four subviews, marked with the number 1 to 4 in Figure 3.2, from which data can be exported:



Call Tree	Time (ms)		Own Time (ms)
<All threads>	12,796	100 %	
java.lang.Thread.run()	12,781	99 %	1,921
NativeMethodAccessorImpl.java (native) gui.GuiController.startClicked()	7,125	56 %	0
GuiController.java:207 analysis.AnalysisExecution.<init>(File, List, List)	5,781	45 %	0
AnalysisExecution.java:42 analysis.AnalysisExecution.generateObjects()	5,781	45 %	15
AnalysisExecution.java:71 analysis.CallTree.<init>(String)	2,906	23 %	0
CallTree.java:30 analysis.CallTree.readFile()	2,906	23 %	0
CallTree.java:44 javax.xml.bind.helpers.AbstractUnmarshallerImpl.unmarshal(File)	2,671	21 %	2,671
CallTree.java:40 javax.xml.bind.JAXBContext.newInstance(String)	234	2 %	234
AnalysisExecution.java:55 analysis.CallTree.<init>(String)	859	7 %	0
CallTree.java:30 analysis.CallTree.readFile()	859	7 %	0
CallTree.java:44 javax.xml.bind.helpers.AbstractUnmarshallerImpl.unmarshal(File)	765	6 %	765
CallTree.java:40 javax.xml.bind.JAXBContext.newInstance(String)	93	1 %	93
AnalysisExecution.java:73 analysis.CallTree.<init>(String)	765	6 %	0
CallTree.java:30 analysis.CallTree.readFile()	765	6 %	0
CallTree.java:44 javax.xml.bind.helpers.AbstractUnmarshallerImpl.unmarshal(File)	625	5 %	625
CallTree.java:40 javax.xml.bind.JAXBContext.newInstance(String)	140	1 %	140
AnalysisExecution.java:51 analysis.Hotspots.<init>(String)	375	3 %	0
Hotspots.java:24 analysis.Hotspots.readXMLFile()	375	3 %	0
Hotspots.java:31 javax.xml.bind.JAXBContext.newInstance(String)	265	2 %	265
Hotspots.java:35 javax.xml.bind.helpers.AbstractUnmarshallerImpl.unmarshal(File)	93	1 %	93
Hotspots.java:32 com.sun.xml.internal.bind.v2.runtime.JAXBContextImpl.createUnmarshaller()	15	0 %	15

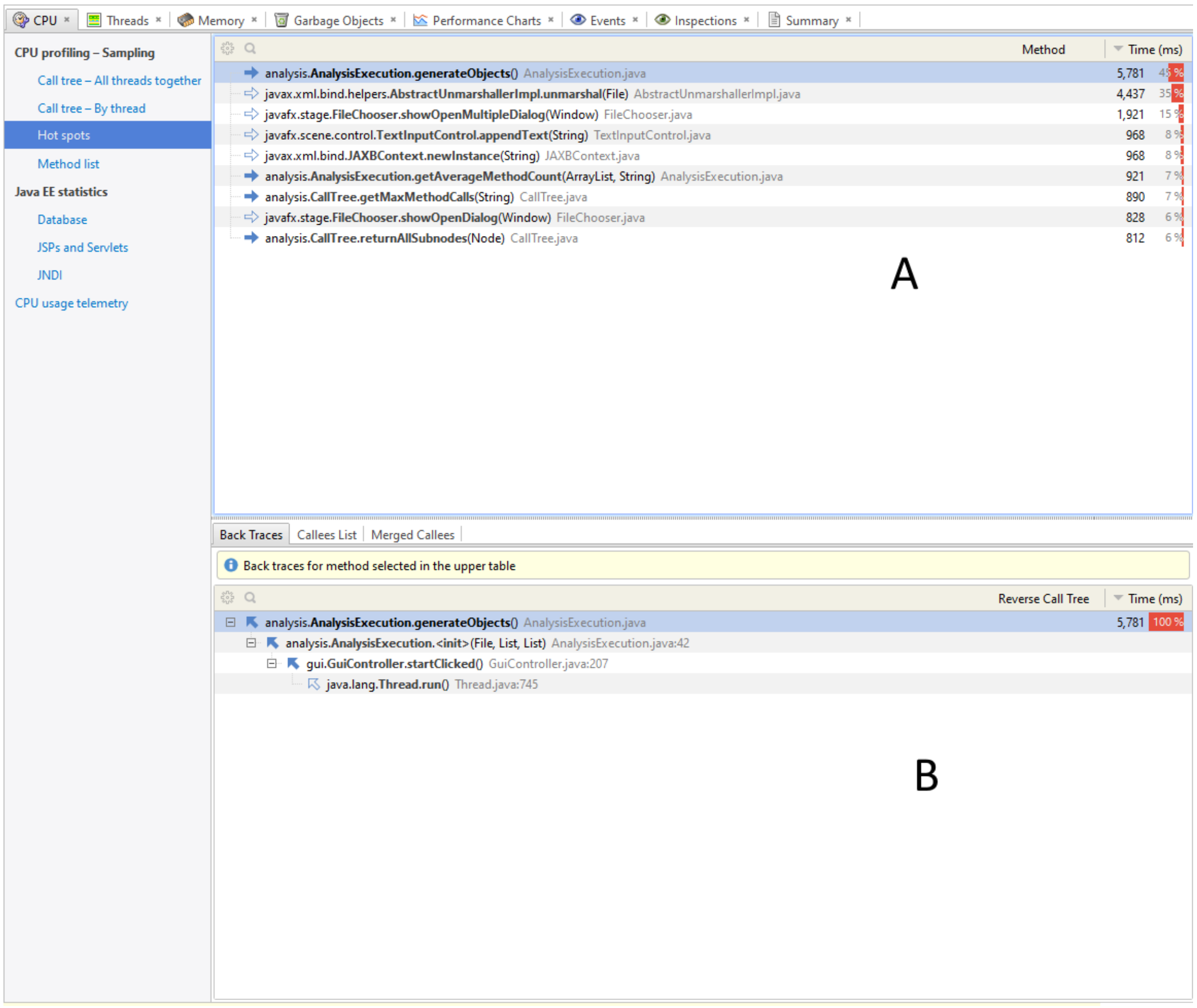
Methods called from the method selected in the upper table

Method	Time (ms)		Own Time (ms)
java.lang.Thread.run() Thread.java	12,781	100 %	1,921
gui.GuiController.startClicked() GuiController.java	7,125	56 %	0
analysis.AnalysisExecution.<init>(File, List, List) AnalysisExecution.java	5,781	45 %	0
analysis.AnalysisExecution.generateObjects() AnalysisExecution.java	5,781	45 %	15
analysis.CallTree.<init>(String) CallTree.java	4,640	36 %	0
analysis.CallTree.readFile() CallTree.java	4,640	36 %	0
javax.xml.bind.helpers.AbstractUnmarshallerImpl.unmarshal(File) AbstractUnmarshallerImpl.java	4,437	35 %	4,437
guiController.MainGui.getMultipleFiles() MainGui.java	1,921	15 %	0
javafx.stage.FileChooser.showOpenMultipleDialog(Window) FileChooser.java	1,921	15 %	1,921
gui.GuiController.problemFileSelector() GuiController.java	1,078	8 %	0
analysis.AnalysisExecution.cthAnalysis(double, int, String) AnalysisExecution.java	1,000	8 %	0
gui.GuiController$Console.lambda$0(String) GuiController.java	968	8 %	0
gui.GuiController$Console$$Lambda$267.run()	968	8 %	0
javafx.scene.control.TextInputControl.appendText(String) TextInputControl.java	968	8 %	968
javax.xml.bind.JAXBContext.newInstance(String) JAXBContext.java	968	8 %	968
analysis.AnalysisExecution.getAverageMethodCount(ArrayList, String) AnalysisExecution.java	921	7 %	31
analysis.CallTree.getMaxMethodCalls(String) CallTree.java	890	7 %	203
gui.GuiController.comparisonFileSelector() GuiController.java	859	7 %	0

Figure 3.2.: Screenshot of the "Call tree - All threads together"

1. First of all the "Call tree – All threads together" section which is shown in Figure 3.2. Here, all the different methods that are called during the CPU tracing are summarized in a call tree. As the name says, there is no categorization between different threads. Everything is packed together and also the method times are added from all threads if a method is called in multiple threads of the system. In the exported data the method name, the time each method took and the own time (time spent in method excluding the subcalls) are included.

2. The "Call tree – By thread" section is very similar to the first one, except that there is a call tree for every single thread of the system. This view is good if an analysis or comparison should be made of different threads. The structure of the XML file is exactly the same as the one from the first section. For this reason, also the included data is similar.



Method	Time (ms)	
analysis.AnalysisExecution.generateObjects() AnalysisExecution.java	5,781	45 %
javax.xml.bind.helpers.AbstractUnmarshallerImpl.unmarshal(File) AbstractUnmarshallerImpl.java	4,437	35 %
javafx.stage.FileChooser.showOpenMultipleDialog(Window) FileChooser.java	1,921	15 %
javafx.scene.control.TextInputControl.appendText(String) TextInputControl.java	968	8 %
javax.xml.bind.JAXBContext.newInstance(String) JAXBContext.java	968	8 %
analysis.AnalysisExecution.getAverageMethodCount(ArrayList, String) AnalysisExecution.java	921	7 %
analysis.CallTree.getMaxMethodCalls(String) CallTree.java	890	7 %
javafx.stage.FileChooser.showOpenDialog(Window) FileChooser.java	828	6 %
analysis.CallTree.returnAllSubnodes(Node) CallTree.java	812	6 %

Back traces for method selected in the upper table

Reverse Call Tree	Time (ms)	
analysis.AnalysisExecution.generateObjects() AnalysisExecution.java	5,781	100 %
analysis.AnalysisExecution.<init>(File, List, List) AnalysisExecution.java:42		
gui.GuiController.startClicked() GuiController.java:207		
java.lang.Thread.run() Thread.java:745		

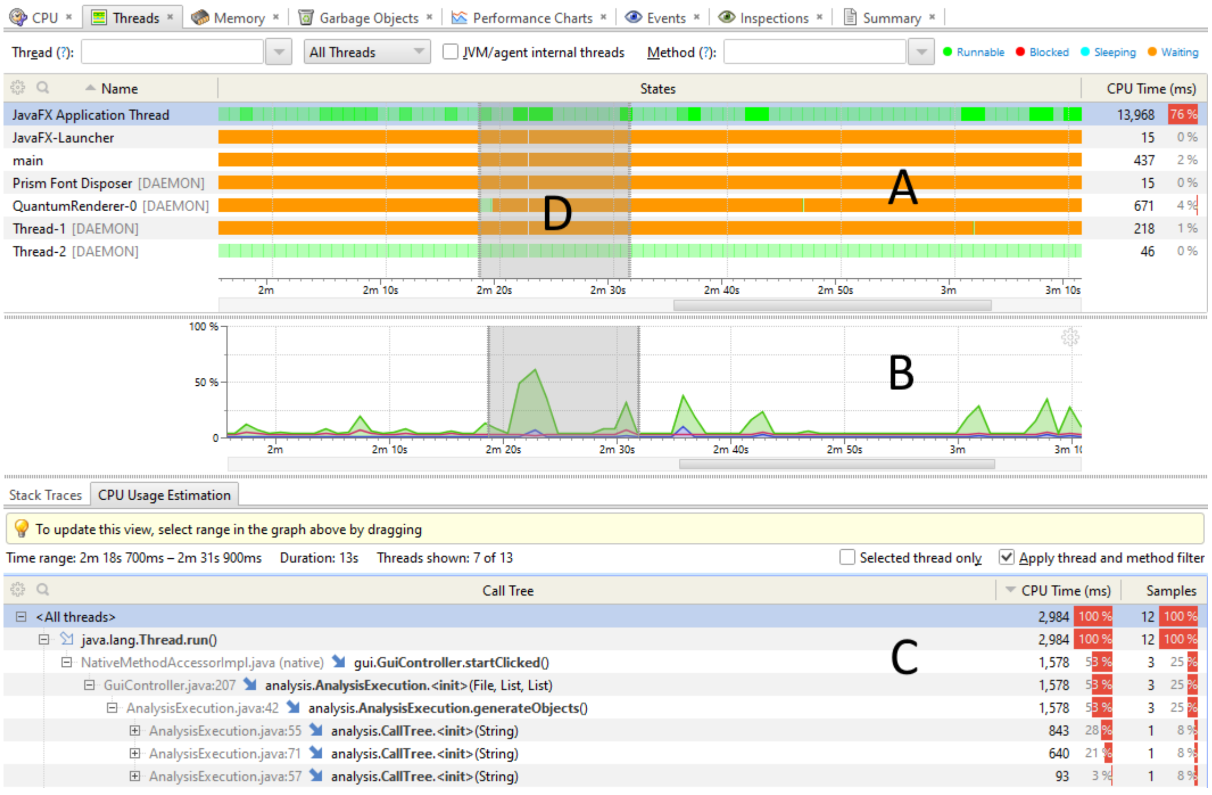
Figure 3.3.: Screenshot of the hotspot section of YourKit

3. The next very important section is called “Hot spots” which is shown in Figure 3.3. YourKit lists all the methods that are suspicious because they have a high method time in comparison to the other methods of the system. This is also the main data for the detection, because from here we extract the methods names which we want to investigate further by analyzing, if the method time is really abnormal and, if it is so which anti-pattern can be assigned to it. The list of hotspot methods includes only the method time in milliseconds.

4. The last section is called “Method list”. We never use it, because the methods are also notated in the first two sections with better representation of the system and not just listed one after another.

For each of the subsections there is a separate view under the main view which can also be exported separately. Namely, there is the "Callees list" (marked with B in Figure 3.2), which is a list of all the methods invoked inside a selected subtree in the main view. Secondly, there is the "Back Trace" (marked with B in Figure 3.3), which is the call tree of a selected method in reverse. Lastly, there is the "Merged Callees". This view shows merged callees for a particular method, i.e. all call traces started from this method. This gives a summary of method execution and its "overall" behavior. We never use data from any of these three views. Examples of the two call trees can be found in Appendix A.1 and Appendix A.2, an example of a hotspot file can be found in Appendix A.3.

3.3.4. Threads View



Name	CPU Time (ms)	
JavaFX Application Thread	13,968	76 %
JavaFX-Launcher	15	0 %
main	437	2 %
Prism Font Disposer [DAEMON]	15	0 %
QuantumRenderer-0 [DAEMON]	671	4 %
Thread-1 [DAEMON]	218	1 %
Thread-2 [DAEMON]	46	0 %

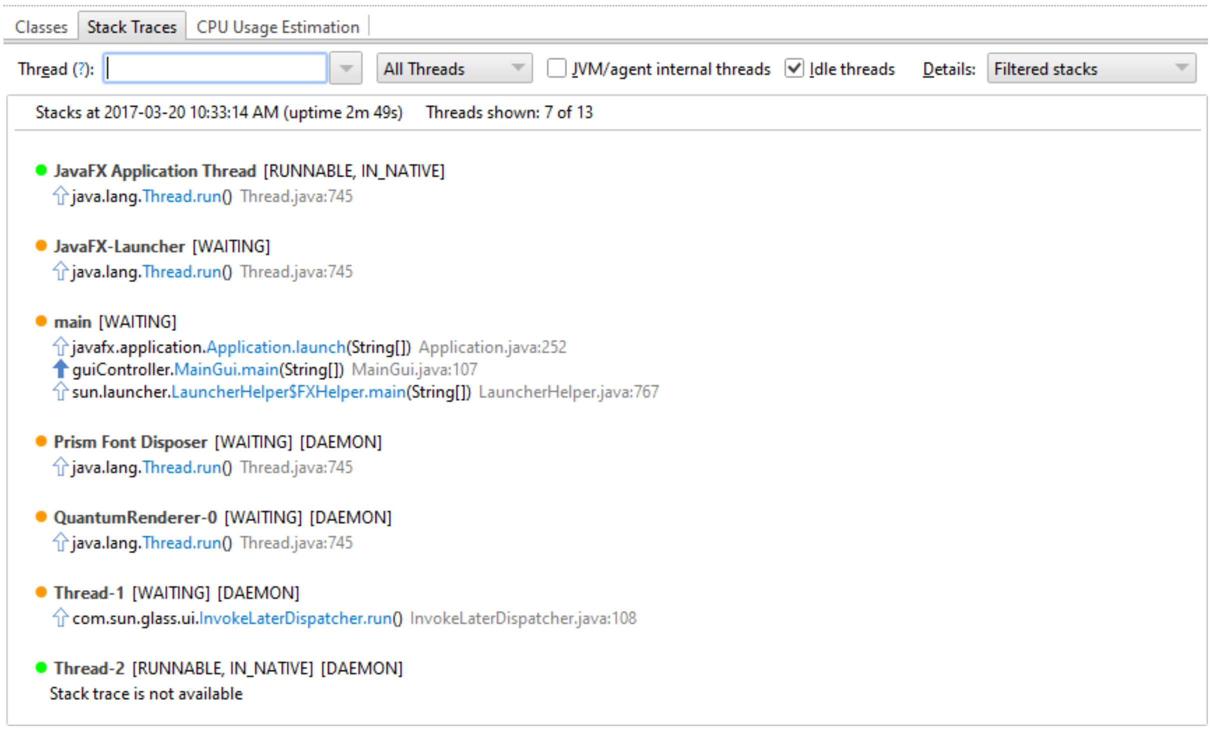
To update this view, select range in the graph above by dragging

Time range: 2m 18s 700ms – 2m 31s 900ms Duration: 13s Threads shown: 7 of 13

Call Tree	CPU Time (ms)		Samples	
<All threads>	2,984	100 %	12	100 %
java.lang.Thread.run()	2,984	100 %	12	100 %
NativeMethodAccessorImpl.java (native) gui.GuiController.startClicked()	1,578	53 %	3	25 %
GuiController.java:207 analysis.AnalysisExecution.<init>(File, List, List)	1,578	53 %	3	25 %
AnalysisExecution.java:42 analysis.AnalysisExecution.generateObjects()	1,578	53 %	3	25 %
AnalysisExecution.java:55 analysis.CallTree.<init>(String)	843	28 %	1	8 %
AnalysisExecution.java:71 analysis.CallTree.<init>(String)	640	21 %	1	8 %
AnalysisExecution.java:57 analysis.CallTree.<init>(String)	93	3 %	1	8 %

Figure 3.4.: Screenshot of the Threads View

The threads view is shown in Figure 3.4. It is separated horizontally in three parts marked with A, B and C in the screenshot. On the top there is a timeline, where all threads are displayed as a long beam. Different colors are showing the state in which the process is. The state can be runnable, blocked sleeping, or waiting. Beneath this bar chart in section B, there is a graph which is showing the CPU usage from the kernel and the user aggregated as well as just the usage from the kernel. Additionally, there is also a graph for the time that is spent in the garbage collection. The bottom part of this view can display the CPU Usage Estimation if a time range is selected by marking an area with the mouse in the bar chart/graph as it is done in D of the Figure 3.4. Furthermore, stack traces can be displayed in the same area if a time is selected by clicking in the chart/graph as shown in Figure 3.5. It is possible to export a call tree by selecting a time range in the graph of the different threads. It is also possible to just get the call tree from a selected thread. The call tree includes CPU time in milliseconds (see Section 3.3.8 for explanation) and samples. This is the number of time points in which a corresponding stack trace has been registered. The charts can be exported as CSV files, whereas the stack traces can only be copied to the clipboard.

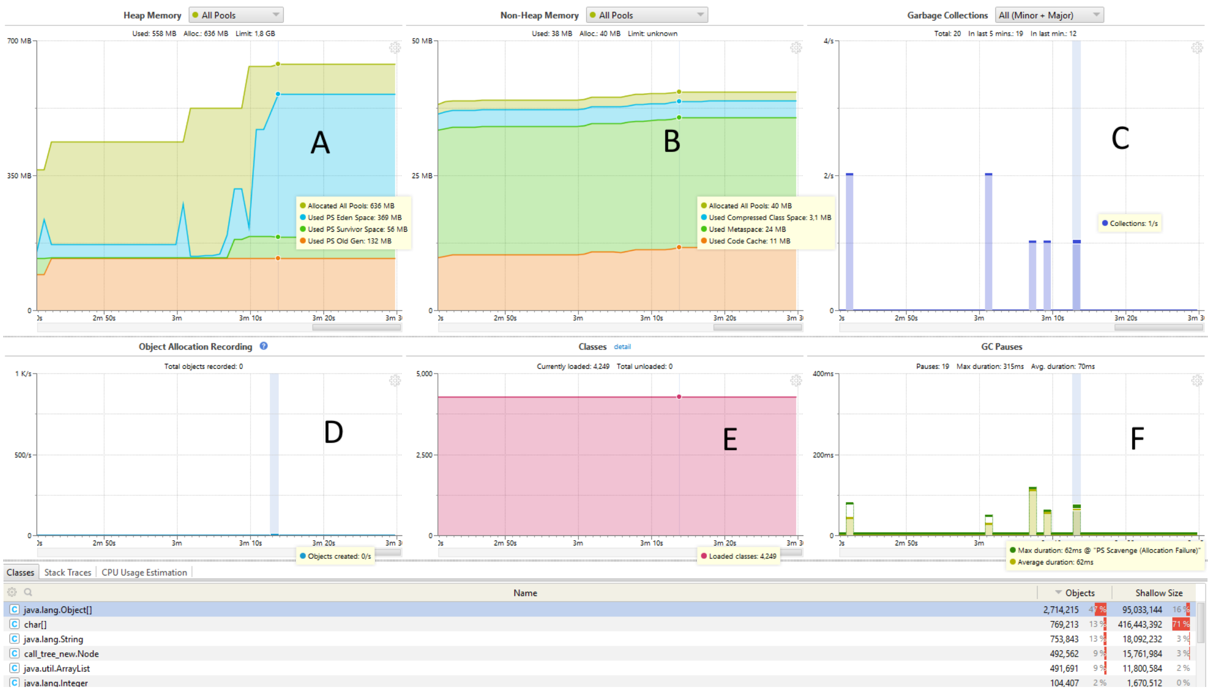


```
Stacks at 2017-03-20 10:33:14 AM (uptime 2m 49s)    Threads shown: 7 of 13

● JavaFX Application Thread [RUNNABLE, IN_NATIVE]
   java.lang.Thread.run() Thread.java:745

● JavaFX-Launcher [WAITING]
   java.lang.Thread.run() Thread.java:745

● main [WAITING]
   javafx.application.Application.launch(String[]) Application.java:252
   guiController.MainGui.main(String[]) MainGui.java:107
   sun.launcher.LauncherHelper$FXHelper.main(String[]) LauncherHelper.java:767

● Prism Font Disposer [WAITING] [DAEMON]
   java.lang.Thread.run() Thread.java:745

● QuantumRenderer-0 [WAITING] [DAEMON]
   java.lang.Thread.run() Thread.java:745

● Thread-1 [WAITING] [DAEMON]
   com.sun.glass.ui.InvokeLaterDispatcher.run() InvokeLaterDispatcher.java:108

● Thread-2 [RUNNABLE, IN_NATIVE] [DAEMON]
  Stack trace is not available
```

Figure 3.5.: Screenshot of the Stack Trace compartment in YourKit

3.3.5. Memory View

In the memory view, which is shown in Figure 3.6, there are six different graphs that show Heap Memory (A), Non-Heap-Memory (B), Garbage Collections (C), Object Allocation Recording (D), Classes (E) and Garbage Collection Pauses (F). The two memory graphs hold data for the different memory states of Java which is visualized by different colors. In the Heap Memory chart (A) are graphs for the “PS Old Gen”, “PS Survivor Space”, “PS


Heap Memory
All Pools
Used: 558 MB Alloc.: 636 MB Limit: 1.8 GB
A
Allocated All Pools: 636 MB
Used PS Eden Space: 369 MB
Used PS Survivor Space: 56 MB
Used PS Old Gen: 132 MB
Non-Heap Memory
All Pools
Used: 38 MB Alloc.: 40 MB Limit: unknown
B
Allocated All Pools: 40 MB
Used Compressed Class Space: 3.1 MB
Used Metaspace: 24 MB
Used Code Cache: 11 MB
Garbage Collections
All (Minor + Major)
Total: 20 In last 5 mins.: 19 In last min.: 12
C
Collections: 1/s
Object Allocation Recording
Total objects recorded: 0
D
Objects created: 0/s
Classes
Currently loaded: 4,249 Total unloaded: 0
E
Loaded classes: 4,249
GC Pauses
Pauses: 19 Max duration: 315ms Avg. duration: 70ms
F
Max duration: 62ms @ "PS Scavenge (Allocation Failure)"
Average duration: 62ms
Classes
Stack Traces
CPU Usage Estimation

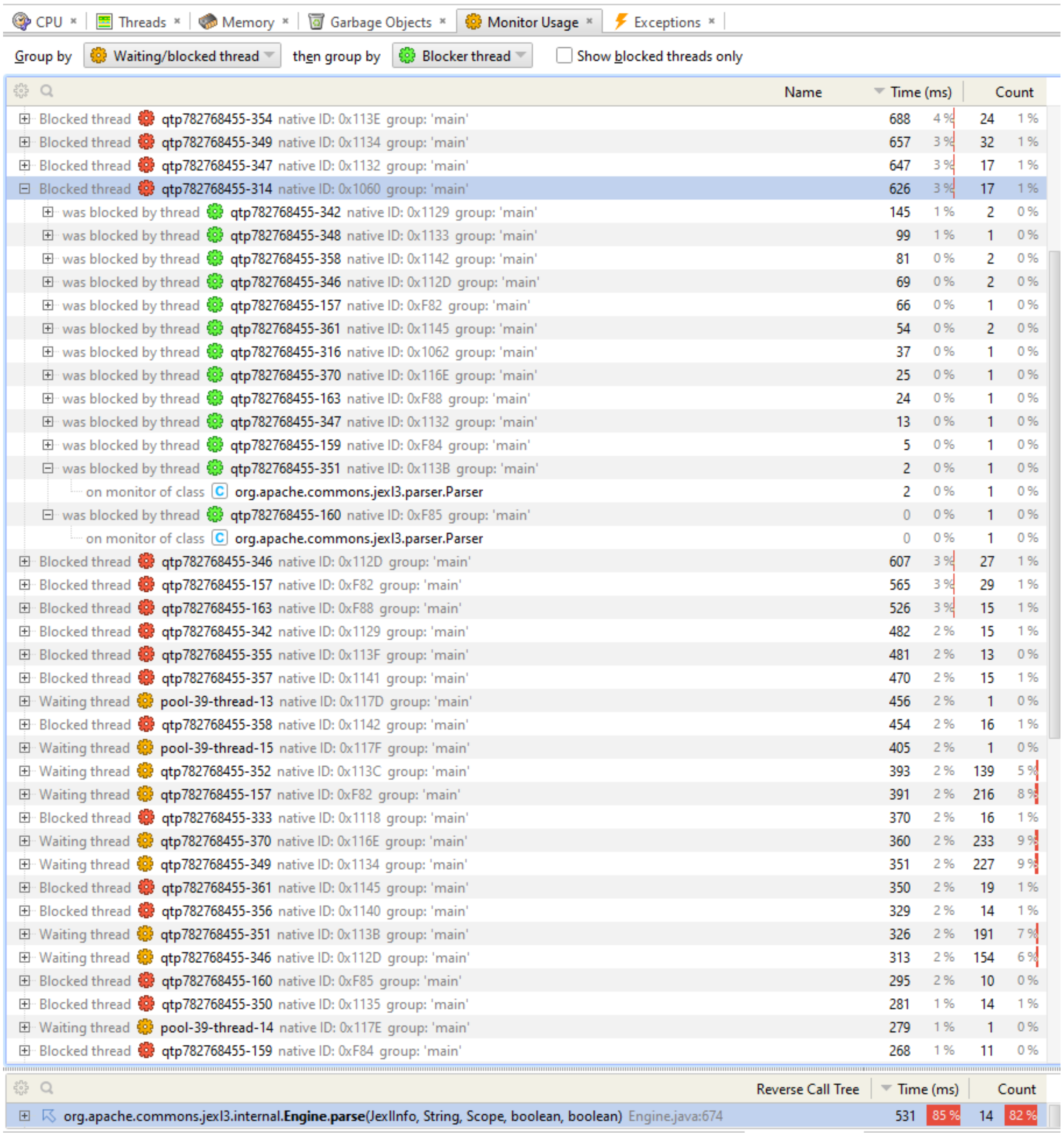
Name	Objects		Shallow Size	
java.lang.Object[]	2,714,215	47 %	95,033,144	16 %
char[]	769,213	13 %	416,443,392	71 %
java.lang.String	753,843	13 %	18,092,232	3 %
call_tree_new.Node	492,562	9 %	15,761,984	3 %
java.util.ArrayList	491,691	9 %	11,800,584	2 %
java.lang.Integer	104,407	2 %	1,670,512	0 %

Figure 3.6.: Screenshot of the Memory section of YourKit

Eden Space" and the complete allocated memory usage from all pools. In the Non-Heap Memory chart (B) are graphs for "Code Cache", "Metaspace", "Compressed Class Space" and the memory allocated from all pools can be found. More information about the different memory types can be found in [YourKit Documentation, Section 5.12]. The graph marked with C shows the number of garbage collections per second over the complete time. All the graphs can be only exported in the CSV format. Also, in all graphs it is marked with a vertical red line, when the "Monitor Profiling" and "CPU Sampling" is started and finished. In a separate view, it is possible to get information about the classes that are loaded in form of the name, the object count, and the shallow size (marked with G in Figure 3.6). Additionally, it is possible to click in one of the graphs and get the stack traces as well as marking a time range in one graph and getting the CPU Usage Estimation which should be the same as in the Threads view. Furthermore, it is also possible to get more information about the objects and when they are allocated in a memory snapshot, to which we did not have access. An example of the output CSV file can be found in Appendix A.6.

3.3.6. Monitor Usage View

In the Monitor Usage View, it is possible to inspect which threads are waiting or blocked during the profiling. Additionally, it holds the wait-/block-time as well as the class name



Group by: Waiting/blocked thread — then group by: Blocker thread — Show blocked threads only [ ]

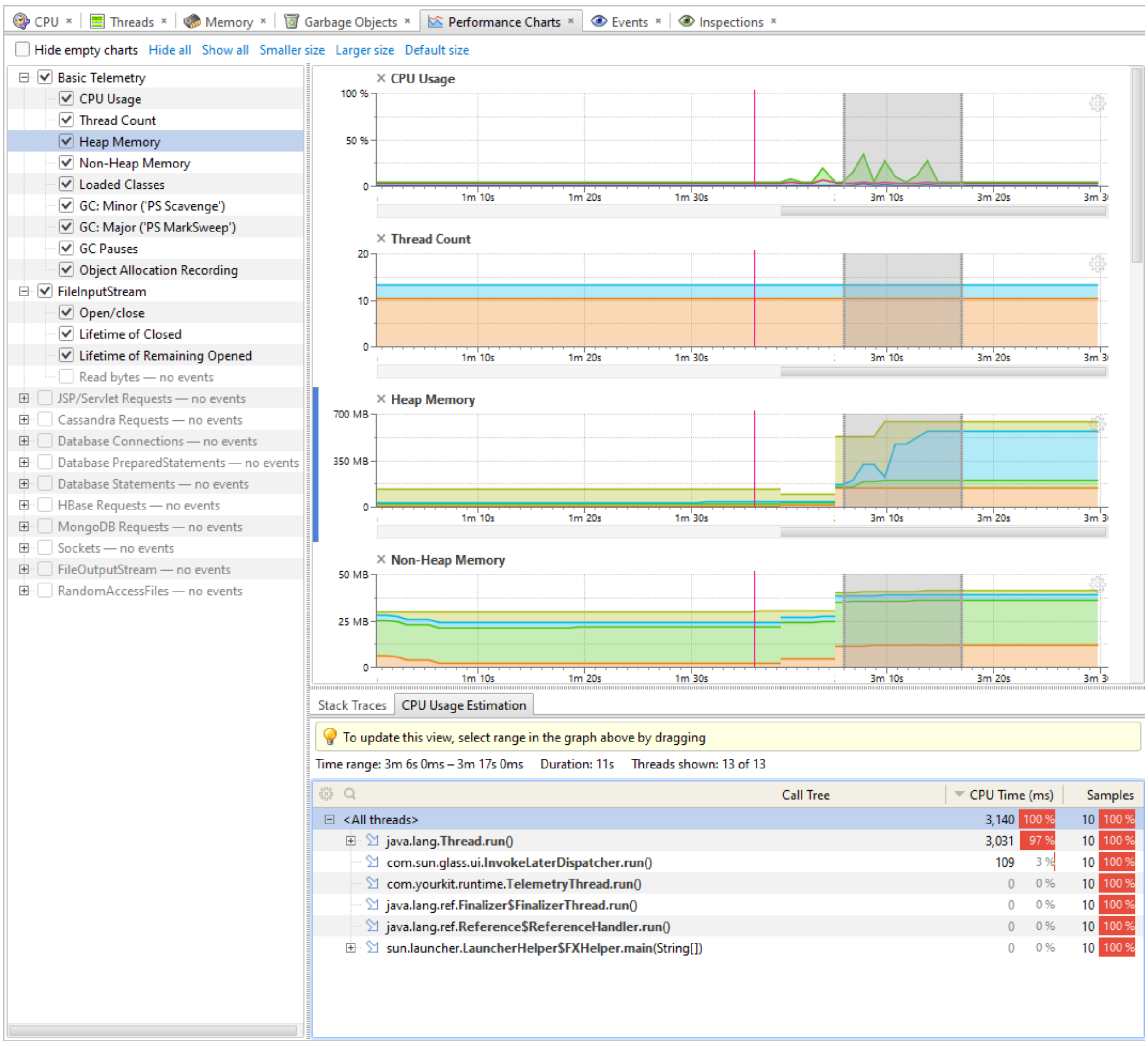
Name	Time (ms)		Count	
Blocked thread qtp782768455-354 native ID: 0x113E group: 'main'	688	4 %	24	1 %
Blocked thread qtp782768455-349 native ID: 0x1134 group: 'main'	657	3 %	32	1 %
Blocked thread qtp782768455-347 native ID: 0x1132 group: 'main'	647	3 %	17	1 %
Blocked thread qtp782768455-314 native ID: 0x1060 group: 'main'	626	3 %	17	1 %
was blocked by thread qtp782768455-342 native ID: 0x1129 group: 'main'	145	1 %	2	0 %
was blocked by thread qtp782768455-348 native ID: 0x1133 group: 'main'	99	1 %	1	0 %
was blocked by thread qtp782768455-358 native ID: 0x1142 group: 'main'	81	0 %	2	0 %
was blocked by thread qtp782768455-346 native ID: 0x112D group: 'main'	69	0 %	2	0 %
was blocked by thread qtp782768455-157 native ID: 0xF82 group: 'main'	66	0 %	1	0 %
was blocked by thread qtp782768455-361 native ID: 0x1145 group: 'main'	54	0 %	2	0 %
was blocked by thread qtp782768455-316 native ID: 0x1062 group: 'main'	37	0 %	1	0 %
was blocked by thread qtp782768455-370 native ID: 0x116E group: 'main'	25	0 %	1	0 %
was blocked by thread qtp782768455-163 native ID: 0xF88 group: 'main'	24	0 %	1	0 %
was blocked by thread qtp782768455-347 native ID: 0x1132 group: 'main'	13	0 %	1	0 %
was blocked by thread qtp782768455-159 native ID: 0xF84 group: 'main'	5	0 %	1	0 %
was blocked by thread qtp782768455-351 native ID: 0x113B group: 'main'	2	0 %	1	0 %
on monitor of class org.apache.commons.jexl3.parser.Parser	2	0 %	1	0 %
was blocked by thread qtp782768455-160 native ID: 0xF85 group: 'main'	0	0 %	1	0 %
on monitor of class org.apache.commons.jexl3.parser.Parser	0	0 %	1	0 %
Blocked thread qtp782768455-346 native ID: 0x112D group: 'main'	607	3 %	27	1 %
Blocked thread qtp782768455-157 native ID: 0xF82 group: 'main'	565	3 %	29	1 %
Blocked thread qtp782768455-163 native ID: 0xF88 group: 'main'	526	3 %	15	1 %
Blocked thread qtp782768455-342 native ID: 0x1129 group: 'main'	482	2 %	15	1 %
Blocked thread qtp782768455-355 native ID: 0x113F group: 'main'	481	2 %	13	0 %
Blocked thread qtp782768455-357 native ID: 0x1141 group: 'main'	470	2 %	15	1 %
Waiting thread pool-39-thread-13 native ID: 0x117D group: 'main'	456	2 %	1	0 %
Blocked thread qtp782768455-358 native ID: 0x1142 group: 'main'	454	2 %	16	1 %
Waiting thread pool-39-thread-15 native ID: 0x117F group: 'main'	405	2 %	1	0 %
Waiting thread qtp782768455-352 native ID: 0x113C group: 'main'	393	2 %	139	5 %
Waiting thread qtp782768455-157 native ID: 0xF82 group: 'main'	391	2 %	216	8 %
Blocked thread qtp782768455-333 native ID: 0x1118 group: 'main'	370	2 %	16	1 %
Waiting thread qtp782768455-370 native ID: 0x116E group: 'main'	360	2 %	233	9 %
Waiting thread qtp782768455-349 native ID: 0x1134 group: 'main'	351	2 %	227	9 %
Blocked thread qtp782768455-361 native ID: 0x1145 group: 'main'	350	2 %	19	1 %
Blocked thread qtp782768455-356 native ID: 0x1140 group: 'main'	329	2 %	14	1 %
Waiting thread qtp782768455-351 native ID: 0x113B group: 'main'	326	2 %	191	7 %
Waiting thread qtp782768455-346 native ID: 0x112D group: 'main'	313	2 %	154	6 %
Blocked thread qtp782768455-160 native ID: 0xF85 group: 'main'	295	2 %	10	0 %
Blocked thread qtp782768455-350 native ID: 0x1135 group: 'main'	281	1 %	14	1 %
Waiting thread pool-39-thread-14 native ID: 0x117E group: 'main'	279	1 %	1	0 %
Blocked thread qtp782768455-159 native ID: 0xF84 group: 'main'	268	1 %	11	0 %

Reverse Call Tree	Time (ms)		Count	
org.apache.commons.jexl3.internal.**Engine.parse**(JexlInfo, String, Scope, boolean, boolean) Engine.java:674	531	85 %	14	82 %

Figure 3.7.: Screenshot of the Monitor Usage section of YourKit

of the waiting class, respectively the thread(s) that are blocking another thread. With this information it is possible to count how many threads are blocked and also of what quantity of threads they are blocked. Furthermore, because there is information about every blocked or waiting thread, it is possible to elaborate the maximum/average/minimum amount of threads that block another thread. As shown in Figure 3.7 the blocked threads are marked with a red, waiting ones with an orange and the threads that block other ones are marked with a green color. An example of the output XML file can be found in Appendix A.4.

3.3.7. Performance Charts



- Basic Telemetry
  - CPU Usage
  - Thread Count
  - Heap Memory
  - Non-Heap Memory
  - Loaded Classes
  - GC: Minor ('PS Scavenge')
  - GC: Major ('PS MarkSweep')
  - GC Pauses
  - Object Allocation Recording
- FileInputStream
  - Open/close
  - Lifetime of Closed
  - Lifetime of Remaining Opened
  - Read bytes — no events
- JSP/Servlet Requests — no events
- Cassandra Requests — no events
- Database Connections — no events
- Database PreparedStatements — no events
- Database Statements — no events
- HBase Requests — no events
- MongoDB Requests — no events
- Sockets — no events
- FileOutputStream — no events
- RandomAccessFiles — no events

To update this view, select range in the graph above by dragging

Time range: 3m 6s 0ms – 3m 17s 0ms   Duration: 11s   Threads shown: 13 of 13

Call Tree	CPU Time (ms)		Samples	
<All threads>	3,140	100 %	10	100 %
java.lang.Thread.run()	3,031	97 %	10	100 %
com.sun.glass.ui.InvokeLaterDispatcher.run()	109	3 %	10	100 %
com.yourkit.runtime.TelemetryThread.run()	0	0 %	10	100 %
java.lang.ref.Finalizer$FinalizerThread.run()	0	0 %	10	100 %
java.lang.ref.Reference$ReferenceHandler.run()	0	0 %	10	100 %
sun.launcher.LauncherHelper$FXHelper.main(String[])	0	0 %	10	100 %

Figure 3.8.: Screenshot of the Performance Charts section of YourKit

In the view, that is shown in Figure 3.8, only diagrams of different metrics are displayed. Every chart can be exported as a CSV file. It is not possible to select a custom time range. Furthermore by default, every chart is limited to the last sixty minutes of the snapshot. That means, if a snapshot is one hour and thirty minutes, only the time range from minute thirty until the end of the snapshot is shown. This setting can be changed with a startup option [YourKit Documentation, Section 5.1]. Different charts like the CPU usage or the memory usage can also be found in previous views. An export example of the CPU usage can be found in Appendix A.5.

3.3.8. Problem with CPU Usage Estimation

The first way of getting information about the method times is to have a look at the call tree data that is only recorded when "CPU sampling" is enabled. The second way of getting method times is looking at the "CPU usage estimation" which is available as long as "stack telemetry" is enabled [YourKit Documentation, Section 5.12]. In our data, this is applicable and the data is gathered from the beginning throughout the complete snapshot. But, as the name says, the estimation is not as accurate as the sampling method. Especially, the measurement struggles when the method times are very low, because the standard sampling rate is set to one sample per second which can be changed if necessary. So, it is only good for measuring events or method calls that take at least several seconds, which should not be a problem, because it is mainly used to locate problematic code responsible for a CPU spike. For our purpose, there are several problems in this kind of data. First of all, it is not very accurate and we want to compare different results to each other, which leads to inaccurate comparison and their results are then not very representative. Also, the export process to get the CPU usage estimation data needs to be done completely manually by selecting a time range in the telemetry graph and then hitting export. Plus, the user cannot really know which time range he should pick. If the whole time range is selected, the spikes cannot be analyzed because they almost disappear in the big data set. Also, the exportation of the complete time range is very frustrating, because there is no way of selecting everything with one click. There is also another big problem with this export method. We tested different time ranges for the same method in different snapshots. Then we noted the percentage the method took in relation to the whole execution time. We look at the percentages, because we need to somehow look at the data in relation, because the time range length differs from snapshot to snapshot, regardless of which of the two methods we use for getting the data.

Example results for one specific method:

- Circuitous Treasure Hunt: 76% - 82%
- Extensive Processing: 69% - 81%
- Wrong cache Strategy: 71% - 76%
- Unnecessary Processing: 12% - 24%

There is a significant difference between the percentages just because a different time range is selected in the same data graph. Therefore, this data cannot be used for our purpose because the comparison between it would be too inaccurate and random. So, we cannot use this data, but in different chats we had with Mr. Avritzer, he said, that the “CPU sampling” is just started if they see that there is a problem. He also said that they ensure that the problem is included in the recorded data. Thus, we should not need more data than the call tree is covering for our detection.

3.3.9. Export Process

As said before, there is not just one data file that YourKit exports. There are several ones per view and not every export is feasible in every data format. The available data formats in YourKit are XML, CSV, HTML, plain text, and zip. In this research, the focus is set on the XML and CSV data outputs, because they are more suited for an automatic analysis. It is possible to export some of the views automatically with the help of a command line tool [YourKit Documentation, Section 21]. This is particularly useful, because the export process of some views is fairly complicated as described in Section 3.3.2 for the call trees. The main command for the export is:

```
java -jar <Profiler Installation Directory>/lib/yjp.jar -export <snapshot file> <target
    directory>
```

With this command, every available view is being exported. However, because we do not need every single view, we can add filters. We add one filter to just export XML files and one to only export the call trees. The result would be:

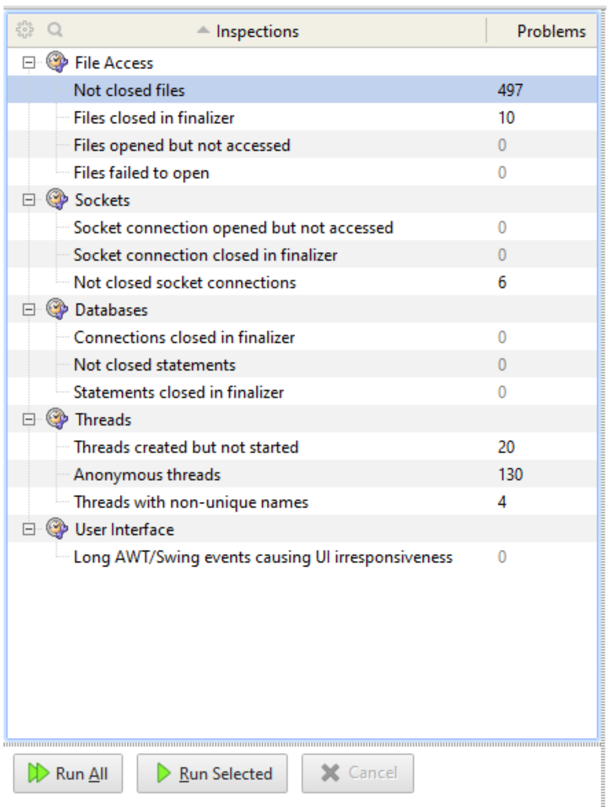
```
java -Dexport.xml -Dexport.call.tree.cpu -jar <Profiler Installation
    Directory>/lib/yjp.jar -export <snapshot file> <target directory>
```

It is also possible to export the needed CSV data by adding `-Dexport.csv -Dexport.charts` to the command. But then, all the charts that are available (46 in total) and additionally the call trees are again exported as CSV files. Our recommendation is to only export the call trees automatically and the carts by hand, because their export process is simple and therefore no unnecessary files are exported.

Both of the charts can be exported by right clicking in the correspondent chart and selecting “Export to...”. The CPU chart can be found in the CPU, Threads, and Performance Charts view. The heap memory usage graph can be exported the same way and is located in the Memory view as well as in the Performance Charts view. Sadly, the “Monitor-usage-statistic” XML file cannot be exported automatically. Thus, a manual exportation is necessary. In order to get the data we need for the analysis, it is necessary to first tick the checkbox "Show block threads only" on the top of the view and then click on "Export to..." in the File menu.

3.3.10. Available Analysis

Our goal is to automatically analyze the exported files from YourKit in order to detect possible anti-patterns in the methods. But, there are also different kinds of analysis methods already built in YourKit.
The first one, which we use as well, is the hotspot section. In this view (Figure 3.3), YourKit already lists the methods that consumed the most time. For us, this is a good indicator for a problem method. YourKit also provides a good documentation on how to find different performance problems [YourKit Documentation, Section 4], like memory leaks by using different capabilities of the profiler. But in all cases, the problem needs to be found manually, the tool just helps by displaying useful data.



Inspections	Problems
File Access	
Not closed files	497
Files closed in finalizer	10
Files opened but not accessed	0
Files failed to open	0
Sockets	
Socket connection opened but not accessed	0
Socket connection closed in finalizer	0
Not closed socket connections	6
Databases	
Connections closed in finalizer	0
Not closed statements	0
Statements closed in finalizer	0
Threads	
Threads created but not started	20
Anonymous threads	130
Threads with non-unique names	4
User Interface	
Long AWT/Swing events causing UI irresponsiveness	0

Figure 3.9.: Screenshot of the Inspections section of YourKit

There is also a view, called "Inspections" (Figure 3.9), where different analysis processes can be run in order to identify various of problems, like not closed files, created threads that are not started or not closed socket connections [YourKit Documentation, Section 14]. Last, but not least, there is also a way of comparing two snapshots to each other in order to detect deviations [YourKit Documentation, Section 5.11]. But, as before, no problem suggestions are delivered. Just the data is compared to each other and the deviation is displayed. In general, it can be said that YourKit provides a lot of different measurements, but we have not found a real automatic problem detection which says where and what the problem is.

3.4. Current Analysis Process

This is the current process, from discovering a problem until finding a solution to it.

1. A performance decrease is noticed, for example the response times are higher than usual.
2. Load tests are started and the system meanwhile is profiled with YourKit in the.
3. A software architect investigates the gathered profiler data in order to find what the problem is and where it is located in the source code.
4. He contacts the responsible developer of the code who then thinks of a solution for the problem.
5. After the problem is fixed, new load tests needs to be made in order to verify that the performance is up to its paces.

Chapter 4

Automated Anti-Pattern Detection

In this chapter, the different analysis processes for detecting the Software Performance Anti-Patterns are presented. In order to detect a SPA, certain data is used which is also introduced in this chapter. Lastly, the program is presented that was developed within this thesis.

In Section 4.1, it is shown how our software integrates into the test process of the development and some differences to the manual analysis process are noted. In Section 4.2, all the used export data from YourKit is presented. It is also explained why exactly this kind of data is used for the analysis. Finally, a summary of problems and limitations of the chosen data is given. In Section 4.3, for every anti-pattern, our ideas, the analysis process, the use of the data and the limitations of the analysis itself are explained. Also, the problems and difficulties the detection had are exhibited. In Section 4.4 the software that we have developed in this research is presented. Therefore, it is explained how it works and how it should be used in order to perform the automatic detection by yourself.

4.1. Analysis Process

Figure 4.1 shows the workflow with Performance Anti-pattern Detection from profiling data (PADprof) which will be introduced in Section 4.4. At first, load tests are ran periodically with the instrumentation of a profiler. After all data is gathered, the automatic detection can start. As described in the next section, in order to use PADprof, more than one snapshot is necessary, aside from the snapshot that should be investigated, a minimum of one comparison snapshot is required. After the detection is finished, PADprof outputs the name of problem methods where a performance anti-pattern is detected. With this information, a developer can directly think of a solution and fix

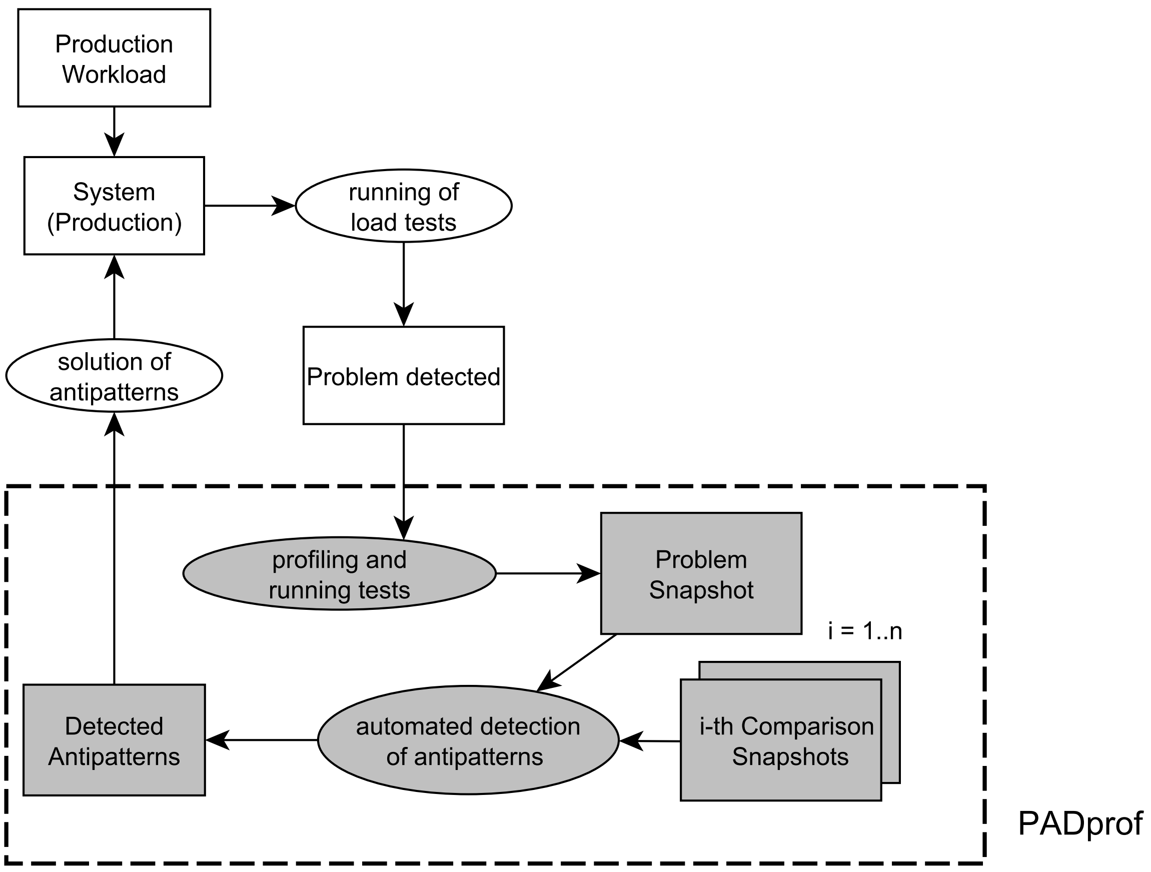


Figure 4.1.: Automatic detection of anti-patterns with PADprof

the problem, because he knows where the problem is located, why the slow down is happening and what the standard solutions for this anti-pattern are. In comparison to the manual detection described in Section 3.4, nobody needs to look at the profiler data and the responsible developer can be found directly with the name of the method. After a problem is fixed, it is possible to run a load test again in order to check if the performance has increased. Apart from that, after a certain time, another test can be run in order to identify new performance problems caused by anti-patterns.

4.1.1. General Analysis Explanations

There are two kinds of Software Performance Anti-Patterns: Single value ones that can be detected by a single value of a performance metric like mean, maximum or minimum value. But there are also SPAs that can only be detected with a trend of performance indices. Therefore also historical data is required in order to detect the anti-pattern.

These are called multiple-values performance anti-patterns [CDT14].
In our research, we noticed that these rules can not be applied, because in a system we don't know, it is not possible to say that a certain mean, max or min value is bad. Hence, we need a comparison between multiple data points in order to distinguish between good and bad values. This means that for every anti-pattern we analyze here, it is necessary to have more than just one snapshot for detecting it. More of how the comparison works is described in the sections below.

A SPA is always related to a method. Therefore, it is necessary to select methods that can cause a problem and analyze them. For this purpose, YourKit already filters methods with the highest execution time and aggregates them in the hotspot section (see Section 3.3.10). These methods are very likely to have a performance problem, because they have a very high execution time in the snapshot compared to the others. So it is necessary to get those methods by reading the hotspot export. Afterwards, the methods are checked if one of the supported anti-patterns causes the problem.

4.2. Used Data

As described in Section 3.3.2, there is a variety of data that can be exported from YourKit. In this chapter, the differences between the data sets is discussed as well as the use of them in the analysis.

Profiling Data

All the data, we are using for the detection is listed in Table 4.1. Call trees can be exported in two ways: one is the "By thread" export, and the other is the "All threads together" export (see Appendix A.1 and A.2 for examples). The difference between those two is pretty simple. In the first data set, the method calls are separated for each thread. Hence, it is possible to investigate every thread separately. For example, this is good when the problem does not occur in any thread, but in certain ones. Also, as we discovered, this data holds all the methods that are called, whereas the "All threads together" data does not need to list every single method. In the export file, each method is listed with the name including the source code line and Java class name, how long the execution of the method took in milliseconds and the, so called, own time which is the time that just the method itself took excluding all methods that are called from it. These same values are also available in the "All threads together" data with the difference that it is not possible to assign a method to a specific thread. Here, the data is aggregated throughout every thread. Therefore all method times are summarized as well as the call

Exported data	CTH	EP	WCS
CPU-hot-spots	method names	method names	method names
Call-tree All-Threads-together		method time	method time
Call-tree By-Thread	method call count		
Monitor usage-statistics		blocked threads count	
Chart-CPU-time	avg CPU usage		
Chart-Heap-Memory			(avg memory usage)

Table 4.1.: Overview of the used data in the different detection strategies

tree itself. Both of these data sets are only recorded when "CPU profiling" is started, which needs to be done manually.

Furthermore, there is the "Monitor Usage Statistics" data, from which it is possible to determine how many threads are blocked, including the time how long they are blocked and also the count of how often they are blocked. Additionally, there is the information included which threads are the cause for a blocked thread. This means, we can extract which threads are blocking a specific thread and also the reason why they are blocking it. It is also possible to get information about the threads that are calling `wait()`, but we are not interested in this data so we excluded it in the export file. More of how the data is exported can be found in Section 3.3.9. This kind of data is only gathered when "Monitor profiling" is enabled.

All data we presented until now is exportable as an XML file that we use. The following two other data sets are only available as a CSV file. For our analysis, we also consider values from two charts. The first one is the "CPU-time" chart which is exportable in the CPU view, the memory view and in the performance charts view. It holds the CPU usage over a maximum time of sixty minutes (can be changed [YourKit Documentation, Section 5.1]). This data is available from the beginning where the profiler was connected to the application. If the data set is longer than one hour, only the last sixty minutes are shown by default, but can also be changed manually. The second chart data that we use in form of a CSV file is the "Heap-Memory" usage. Like before, the data is only available for a certain time range of the snapshot if nothing is customized. For our analysis, we are only using the "PS Eden Space" data although there is more (see Section 3.3.2). It shows the usage of the heap memory space in megabytes over time.

4.2.1. Data Limitations

Because the analysis is done with a comparison between different snapshots, each of the data set needs to be comparable. As described in the section before, we need to rely on the CPU profiling data which needs to be turned on manually. For that reason the CPU profiling record times are different in the snapshots (see Section 3.3.1 for explanation). Here is a comparison between the snapshots we investigated:

- Circuitous Treasure Hunt: 53 000 ms
- Extensive Processing: 20 000 ms
- Wrong cache strategy: 100 000 ms
- Unnecessary Processing: 9 000 ms
- Baseline data: 3000, 4000 and 18 000 ms

Now for instance, the unnecessary processing anti-pattern file is not comparable with the other ones, because certain methods are not executed yet and also the relation between the different method times are not meaningful because of the short recording time. For that reason also the “baseline files” are not really comparable, in 3000 ms not a whole execution is covered and also methods that are executed in a short time are in this little snapshots relatively higher in comparison to the others. Also in comparison, the wrong cache strategy file has throughout the worst method times which can be explained by the high recording time in comparison to the others, because in a short time it is less probable for a method to take an abnormal period of time than when data is gathered for a long time.
The conclusion of this analysis is that for a good detection of SPAs the CPU profiling time should not be too low, and for a good comparison between different snapshots the time should be similar. How long a recording needs to be made depends on the software system that is being investigated.
Here, in our research, everything above 50 000 ms was reasonable. For a good comparison between the different snapshots, we instrumented the EP and the UP once again with 140 000 ms.

Also we are limited due to the fact that we do not have access to so-called “memory snapshots”. This snapshots can offer more data like recording of object allocations and information about garbage objects. This could be relevant for anti-patterns like “Excessive Dynamic Allocation” [Tru11].

4.2.2. Analysis Limitations

Because we cannot detect a SPA just by investigating one snapshot, there are a couple of limitations that come along with this kind of analysis. First of all, there is the limitation that more than one data file is required in order to detect any anti-pattern. Therefore it is not possible to analyze a complete new system. Also, if there is more than just one file for comparison, the rate of wrong detections can be decreased because we have more

data points to compare the problem file with.
Also, it is not possible to detect a problem which exists from the beginning, because it is necessary that a value, like the method count or the method time, does poorly in the problem file. If the problem exists from the beginning and does not get worse over time, our analysis thinks that this is normal and should be like that, because no deviation can be found in comparison to the historical data. For this reason, only performance problems that occur by adding new stuff or by changing something, can be detected. This means, that problems existing from the beginning of the data recording cannot be detected.

4.3. Detection Strategies

This section is subdivided into the three anti-patterns Circuitous Treasure Hunt (CTH), Extensive Processing (EP), and Wrong Cache (WCS). Then, in each subsection the idea of the analysis, the process of detection, the use of the data, and the limitations are explained. Therefore, pseudo code is provided for each analysis. Because the WCS anti-pattern cannot be detected as good as the other ones, in Section 4.3.3 there is additionally a subsection called “Difficulties” which explains why this SPA is more complicated to detect.

4.3.1. Circuitous Treasure Hunt

Idea

The basic idea for the analysis is directly derived from the definition of the anti-pattern itself (see Section 2.1. The main aspect is the count of how many methods are called from each method. Therefore it is necessary to detect, if too many calls are required in order to execute a method until the end. Firstly, we thought that this is possible with just one snapshot file. The idea was to somehow define an average method count over all methods in one snapshot. This sounded easy because we have the call tree of the whole execution and it should be a simple task to just divide the tree into certain “main” methods in order calculate the method count of each of these. Afterwards, the last step only serves for calculating an average over these method calls.

But it did not work, because all the snapshots had just three to four “main” methods. Also one of them was the `java.lang.Thread.run()` method, which can be found in every thread and starts all the work. Therefore, it also had a big method call count (500+), because it is the main method that is being executed in every thread. This high method count was afterwards being compared to the very low count of the other two to three methods (mostly below 50 method calls). The outcome was a not very meaningful average method count of the snapshot.

The next approach was then to divide these main methods, especially the `Thread.run()` method, into smaller sub methods to get a finer granularity of the count. An automatic partitionment just by iterating through the call tree and separating methods that have a certain method call count was not successful because then the parts were more or less random and would not have been comparable to other divisions.

We think that dividing a big method into smaller sub methods only makes sense when they also call many other methods. Otherwise, we end up on a huge number of methods which call only one other method. To test if this can work, we looked how many methods are calling other "blocks" of methods. And this is our result: The test file contained 19106 method calls. 985 of them did not call any other method. The majority with 17408 called one other method. 688 called two, 22 called three and 64 called 4 other methods, which is the maximum of direct methods calls. Sure, every method that is called, can then also call another method, and then this method can call another one and so on. But this is not the apportionment we want. Because if we want to divide these chains of one method calls another and the other calls another again and so on, it is really difficult to set a boundary for when this chains should be cut. The cutting can just be set with some thresholds which may cause a pretty random decision when a method is divided, which leads into incomparability between different results as described before.

After this investigation we looked for methods that at least call two different other methods in order to identify important nodes from the tree. We also thought, it is not a good idea to divide methods into very many parts, because then the method count is again not representative. So we set some boundaries for dividing the methods which included a minimum method count. This worked pretty good, but because of the somehow "calculated" division, the method counts of different snapshot were nearly similar. Also, the count of problem methods from the hotspot section were significantly lower than the calculated average count over the whole snapshot, which leaded to no possible detection at all.
At the end, we opted for a comparison between different snapshots instead of relying on just one file. This gave us the possibility to compare the method count of the same method in different snapshots, which is a much better approach than comparing an average count of one snapshot, because this count gets higher if a method really has the Circuitous Treasure Hunt anti-pattern, especially when the method is called multiple times.

As operational data, we use the CPU-usage to determine if the CTH is present in a certain snapshot. We use this information, because we found out that the CPU utilization is significantly higher in the CTH snapshot in comparison to the other snapshots we have. The CPU usage averaged out at 95% and the maximum was at 100%. The remainder snapshots reported an average usage of 71%, 76% and 18%. Furthermore, the maximum usage was with 69% and 80% significantly lower than in the CTH snapshot. Also, this value is a good indicator that the system is stressed by calling many methods. It is not a problem, if there are many method calls, but the CPU utilization is low, because then the calls can be made without decreasing the performance.

Used Data

For this analysis we use the data that holds information about every thread, because only in this data set we can compare the execution of all threads to each other. Hence, we can go through the call tree and identify the thread that has the highest or smallest method count. Furthermore it is also possible to calculate the average method count throughout all threads. Every option (min/max/average) is selectable in the PADprof tool and can be helpful for different analysis approaches. Mainly important is the maximum count, because in software performance the spikes count, not the overall performance. For the second part of the analysis, we use the CPU usage CSV file in order to calculate the average CPU utilization.

Analysis Process

The pseudo-code of the CTH detection procedure is reported in Algorithm 4.1. The analysis process is divided in two parts. First of all, we need to check, if the CPU usage of the problem file is higher than the one from the average overall files. For this purpose, we analyze the CPU usage chart CSV file. Because the data is recorded as soon as the profiler is connected, the system often idles in the beginning since the load test has not started yet. Therefore we filter the CPU usage data by deleting all values that are below five percent utilization. This method gave us good results and could be easily realized. Also, it is no problem to delete all the values below a small threshold, because when the system is load tested, a significantly higher utilization of the CPU is expected as described in the previous section. After this filtering, the average usage can be calculated from the left over data. Then, the CPU usages from the problem file and the average overall files can be compared by adding the CPU threshold for this analysis. The threshold is customizable, but the default for this analysis is set to 10%. So, the problem files' average CPU usage needs to be ten percent higher than the average overall files.
When the condition is fulfilled, the hotspots from the problem file are read in order to know which methods can cause a problem. Then, the hotspot method names are

searched in the call tree of every snapshot that is available for the analysis. The name is searched in every thread, and the number of method calls is compared to each other. Then selectively the highest, lowest or the average count of all threads of the snapshot is saved for the comparison at the end. After all files are analyzed, the average of all method counts, including the problem file, is calculated. This value is then compared to the specific method count of the problem file.

For this comparison, we first look if the problem files' method count is higher than the calculated average count over all snapshot. If this is the case, we need to make sure that the deviation is big enough to say that the Circuitous Treasure Hunt can be the problem. For this analysis, we looked at the results and decided that the anti-pattern is detected, when the problem method has 25% more calls than the average count from all the inspected files. But, because this software can be used in different scenarios where a higher or lower threshold could be useful, the number of the percentage for the detection can be customized. Thus, the developer can investigate the profiler data as he wants. Also, with this feature, the data can be analyzed multiple times with different thresholds if it is reasonable.

Algorithmus 4.1 Circuitous Treasure Hunt

```
 1: procedure CTHANALYSIS(countThreshold, cpuThreshold, analysisOpt)
 2:     avgCpuUsageAll ← getAvgCpuUsage(allData, filterThreshold)
 3:     avgCpuUsageProb ← getAvgCpuUsage(probData, filterThreshold)
 4:
 5:     if $avgCpuUsageProb > (avgCpuUsageAll * cpuThreshold)$ then
 6:         for all hotspotMethods do
 7:             probMethodCount ← getMethodCount(probData, hotspotMethod, analysisOpt)
 8:             avgMethodCount ← getMethodCount(allData, hotspotMethod, analysisOpt)
 9:             if $probMethodCount > (avgMethodCount + countThreshold)$ then
10:                 display(hotspotMethod detected as Circuitous Treasure Hunt anti-pattern!)
11:             end if
12:         end for
13:     end if
14: end procedure
```

Limitations

Because, we are looking at the CPU usage, it is possible that this anti-pattern can be marked as solved if only the power of the CPU is increased. But in reality, there is a problem in the procedure itself. Otherwise just the limitations from Section 4.2.2 were discovered.

4.3.2. Extensive Processing

Idea

Like before, the main idea is derived from the definition of the anti-pattern itself (see Section 2.1). The problem is present, if a long running process is blocking other processes to use a certain resource. So, we first look at the amount of blocked threads in every snapshot and compare the problem count to the average count. Second, we need to detect a high method time, because the resource is blocked and the method cannot be executed as fast as it could. And second, we need to look for a resource that is blocked because of a certain process. To get the first information we need, we analyze the "Monitor usage view" which was presented in Section 3.3.2. With this data it is possible to determine how many threads are blocked. For the second value that we need for the analysis, we can just look at the method times of the different hotspot methods in the call tree.

Used Data

For the first part of the analysis, we use the "Monitor usage statistics" data in order to find out how many threads are blocked. There is more data available, as described in Section 3.3.2, but the amount of blocked threads is sufficient for our analysis.

For getting the different method times, we prefer the export file that has all threads summarized. This is particular advisable, because then a comparison between methods is simpler as comparing every thread to each other. Also, with this data, we have a better overview over all the method times. Also, we can not use the CPU estimation data, as described in Section 3.3.8. In the same part, we also spoke over the problems that come with the CPU profiling. Hence, we can not use the absolute method times and need to calculate the percentages of each method in relative to the complete execution time.

Analysis Process

The pseudo code of the EP detection procedure is reported in Algorithm 4.2. Like in the other anti-patterns, the first step is to look if the operational threshold is fulfilled. In this case, we are investigating the number of blocked threads in every snapshot. Therefore we analyze the "Monitor-usage" export file. Because we have exported the file as described in Section 3.3.9, we just need to count the number of entries in this file to get the amount of blocked threads. After that a check is made to find out, if the problem count is bigger than the average count, plus the defined threshold. Like in the Circuitous

Treasure Hunt analysis, the threshold is customizable. The default value is set to 25%, which means that the problem files' count needs to be 25% higher than the average. If this is fulfilled, we look for the hotspots of the snapshot by analyzing its export file. Then we go through the call tree of the "All threads together" file and search for the method names. After we found these, the next step is to calculate the method time relative to the complete execution time in percentage. This could be done by dividing the method time through the complete time profiling. In the end, these values can be compared to other snapshots. In the comparison, a threshold is necessary for detecting the SPA, because in different snapshots there are small deviations that should not be detected by the software. For this reason, a method is only detected as the Extensive Processing anti-pattern when the percentage of the problem method is 10 points higher than the average over all the available snapshot files. Like in the Circuitous Treasure Hunt analysis, this threshold can be changed by the developer to fit custom scenarios.

Algorithmus 4.2 Extensive Processing

```
procedure EPANALYSIS(blockedThreadsThreshold, methodTimeThreshold)
    $blockedThreadsCountAll \leftarrow$ getBlockedThreadsCount(allData)
    $blockedThreadsCountProb \leftarrow$ getBlockedThreadsCount(probData)

    if $blockedThreadsProb > (blockedThreadsAll * blockedThreadsThreshold)$ then
        for all hotspotMethods do
            $probMethodTime \leftarrow$ getMethodTimeInPercent(probData, hotspotMethod)
            $avgMethodTime \leftarrow$ getMethodTimeInPercent(allData, hotspotMethod)
            if $probMethodTime > (avgMethodTime + methodTimeThreshold)$ then
                display(hotspotMethod detected as Extensive Processing anti-pattern!)
            end if
        end for
    end if
end procedure
```

Limitations

The snapshots that are compared should have fairly the same length in order to compare the amount of blocked threads. Also, the longer method times does not need to be connected to the blocked threads count. It is also possible that the correlation is just a unlucky accident. This is because the method time itself is a very good indicator for almost every anti-pattern.

4.3.3. Wrong cache strategy

Idea

Like in the Extensive Processing anti-pattern, we thought that this analysis consists of two parts. Firstly, we need to compare the method times to identify a performance problem. Because if the cache is used wrong, the method execution should be slowed down. This should be the case, because when the cache is implemented incorrectly, accessing it could miss and the data needs to be loaded once again. Secondly, we need to identify that there is a problem in the cache itself. For that reason, we need data about the memory usage which can be found in the memory section of YourKit described in Section 3.3.2. Then, if a method is slower than before and the memory usage is worse, we can assign this anti-pattern to it. We had also the idea to calculate the time how long methods with the name "cache" in it take in every snapshot. And we also thought of counting the calls of this "cache-methods" like in the Circuitous Treasure Hunt anti-pattern. The results of this testing are presented in the next section.

Difficulties

We know, that the method time comparison can be made, because this was already done in the Extensive Processing anti-pattern analysis. But with the memory data analysis, we had our problems. It is possible to gain data from all possible kinds of memory types. But, as we investigated all the different snapshot, we noticed that this data does not really change. Because there is a graphical representation of the memory usage in form of a graph, the data is easily comparable by just looking at it. We found out that sometimes the graphs look different, but it cannot be said that especially the data from the Wrong Cache Strategy snapshot has noticeable problems. There are different kinds of spikes in the graph, but nothing that could be automatically detectable, because everything is more random than has a specific structure which can be seen in Figure 4.2. As can be seen in the image, there is a difference when comparing the first two graphs with the last ones. But this did not help us, as described later. In order to proof this, we analyzed the heap-memory usage in all snapshots. Therefore, we needed to filter the values, because of the idle times of the system which we introduced in the Circuitous Treasure Hunt anti-pattern. Hence, we looked again at the CPU usage data, filtered it as before and then transferred it to the memory usage by just using the data points in which the CPU usage is above five percent. Then we calculated the average over the complete analysis time of the snapshots. The results are shown in Table 4.2.

The amount of data in the different spaces are very similar. For us, the "EdenSpace" is the important part of this analysis, because this is the memory section which is

average usage	CTH	EP	WCS	UP
EdenSpace	673 MB	671 MB	643 MB	676 MB
SurvivorSpace	4 MB	4 MB	5 MB	9 MB
Old Gen	325 MB	987 MB	163 MB	220 MB
used values	3039	2128	182	928

Table 4.2.: Average filtered heap memory usage

most related to cache, since it is the space where all data is initially allocated [Java documentation16]. Furthermore, it is the space which changes the most in the different snapshots. However, in the Wrong Cache Strategy snapshot this space was even used at least from all snapshots. We also did a run, where no filtering was applied. The results are shown in Table 4.3.

average usage	CTH	EP	WCS	UP
EdenSpace	662 MB	679 MB	828 MB	723 MB
SurvivorSpace	4 MB	6 MB	1 MB	17 MB
Old Gen	300 MB	655 MB	102 MB	124 MB
used values	3583	3592	845	3428

Table 4.3.: Average heap memory usage without filtering

This looks better in view of the usage in the Wrong Cache snapshot, but we cannot use this data, because it does not represent the situation when the system is running. We also noted the used amount of values we used for the analysis. The least amount of values is used in the Wrong Cache Strategy snapshot. This is because the snapshot is the shortest one with just about 14 minutes, the other ones have data for over one hour, but it is by default cut by YourKit to sixty minutes (this limit can be changed [YourKit Documentation, Section 5.1]). But, it should not affect our analysis, because we are looking at the average usage. Furthermore, just to be sure, we also made a test, where we selected a two-minute time range, in order to also compare the complete usage and not just the average. The results for the "EdenSpace" is shown in Table 4.4.

After this test, it was clear that this data can not be part of the detection. Here the WCS was the second weakest, which does not help for the detection. Just to be sure, we also investigated the non-heap-memory usage. The results with 5% Threshold for the CPU Usage and without filtering can be seen in Table 4.5 and in Table 4.6.

Also this data did not show us a significant difference in the Wrong Cache snapshot. So, in our opinion, it is not possible to detect something abnormal in this data in order to assign the Wrong Cache anti-pattern. We then investigated the method times as well as

	CTH	EP	WCS	UP
Total	71,33 GB	82 GB	80,80 GB	83,36 GB
Count	119	119	119	119
Time spent	118,2 s	118,1 s	120,6 s	119 s
Data/timepoint	0,6 GB	0,69 GB	0,67 GB	0,7 GB

Table 4.4.: Total heap memory usage in two minutes

average usage	CTH	EP	WCS	UP
Compressed Class Space	16 MB	16 MB	15 MB	15 MB
Metaspace	156 MB	152 MB	137 MB	148 MB
Code cache	97 MB	98 MB	61 MB	85 MB
used values	3039	2128	182	928

Table 4.5.: Average filtered non-heap memory usage

average usage	CTH	EP	WCS	UP
Compressed Class Space	16 MB	16 MB	15 MB	15 MB
Metaspace	155 MB	151 MB	134 MB	137 MB
Code cache	93 MB	97 MB	48 MB	56 MB
used values	3583	3592	845	3428

Table 4.6.: Average non-heap memory usage without filtering

the method count. First, we looked how long methods take with the word “cache” in the name in relation to the complete execution time:

- Circuitous Treasure Hunt: 13,72% (11 main methods)
- Extensive Processing: 96,8% (11 main methods)
- Wrong cache strategy: 85,9% (8 main methods)
- Unnecessary Processing: 96,9% (23 main methods)

This analysis did not show what we expected, because the WCS does not stick out. So, we looked for the occurrences of the word cache in all methods (“All threads together data”). The search included method names, package names and class names.

- Circuitous Treasure Hunt: 90 calls
- Extensive Processing: 336 calls

- Wrong cache strategy: 118 calls
- Unnecessary Processing: 198 calls

Filtered to only look at the method names:

- Circuitous Treasure Hunt: 56 calls
- Extensive Processing: 216 calls
- Wrong cache strategy: 74 calls
- Unnecessary Processing: 144 calls

After looking at this data, it should be clear, that we also did not find here a good evidence for the automatic detection of the WCS anti-pattern. Every data we look at, is either the same in every snapshot or has not the effect in the Wrong cache snapshot as we thought. We think the solution that was invented to solve this specific problem is not represented in the YourKit data. Now data is passed by reference and not by value, in order to not have to deserialize the object. This cannot be seen in the YourKit data, and as also Mr. Avritzer said, that is “pretty tough” to detect, because therefore the program needs to know what every method is doing and how they are connected. But, we think that maybe other cache-related problems can be detected. Hence, we thought of a more basic approach to detect this anti-pattern which is described in the next section.

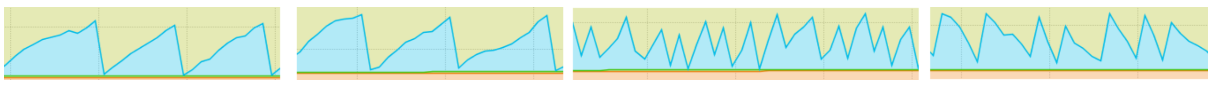


Figure 4.2.: Heap memory comparison for CTH, EP, WCS, UP from left to right

Used data

Like in the Extensive processing anti-pattern, we use the “All threads together” data in order to compare the method times of the hotspots and also count the method calls for testing reasons. Additionally, we are using the “Heap-Memory” data to help the user identify a problem in the memory.

Analysis Process

The pseudo-code of the WCS detection procedure is reported in Algorithm 4.3. This process is very similar to the one of the the EP, because we first analyze the method times of the hotspots. But this time we also concentrate on the name of the method.

If a certain method is slower than before and also has the word “cache” as part of the method name, we clarify it as the Wrong Cache Strategy anti-pattern. Additionally, the average heap-memory usage is also displayed as a help for the user but does not affect the detection. For this purpose, we use the filtered “EdenSpace” values as described in the previous section.

Algorithmus 4.3 Wrong Cache Strategy

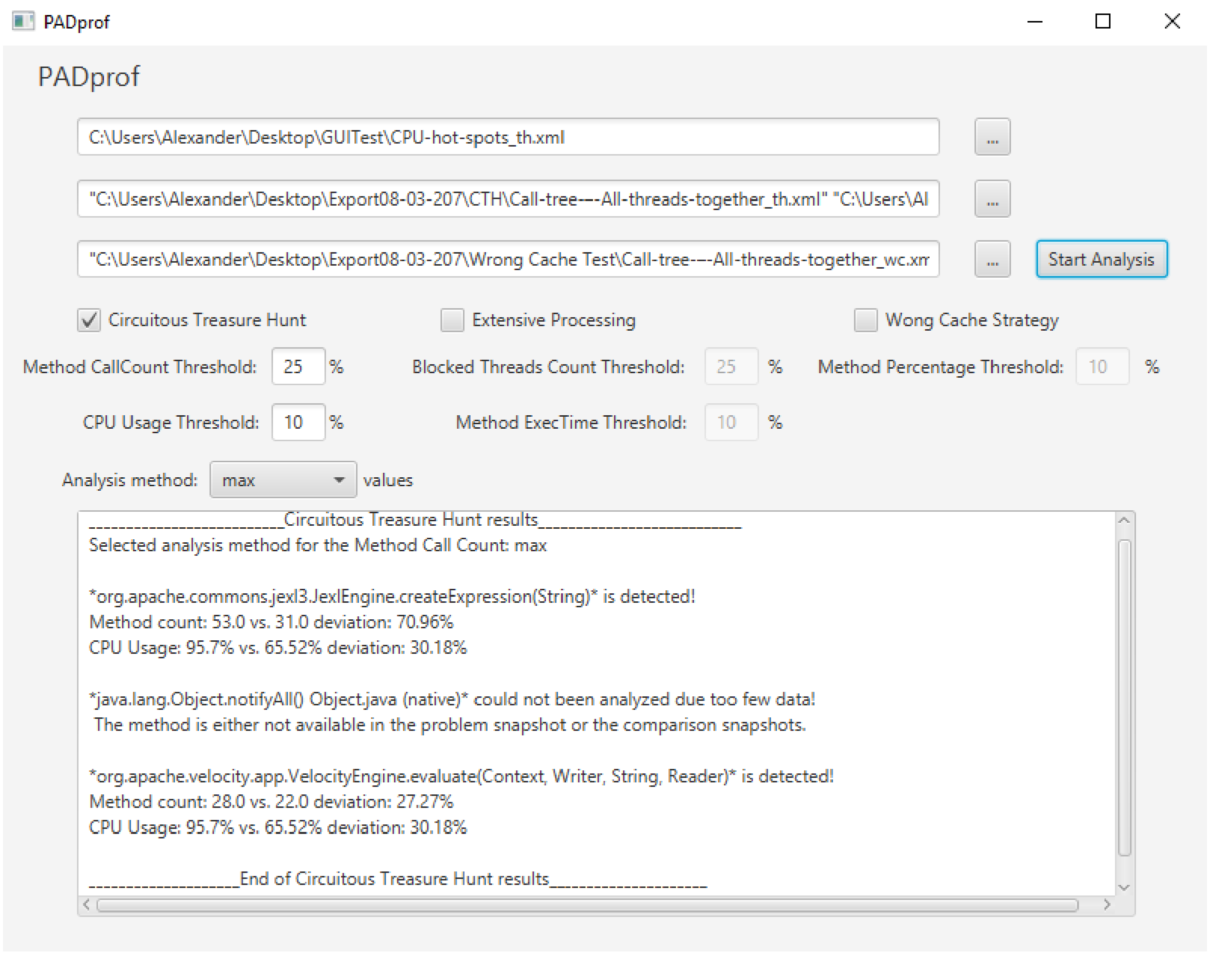
```
 1: procedure WCSANALYSIS(methodTimeThreshold)
 2:     blockedThreadsCountAll ← getBlockedThreadsCount(allData)
 3:     blockedThreadsCountProb ← getBlockedThreadsCount(probData)
 4:     for all hotspotMethods do
 5:         probMethodTime ← getMethodTimeInPercent(probData, hotspotMethod)
 6:         avgMethodTime ← getMethodTimeInPercent(allData, hotspotMethod)
 7:         if hotspotName.contains(“cache”) then
 8:             if $probMethodTime > (avgMethodTime + methodTimeThreshold)$ then
 9:                 avgMemUsage ← getavgMemUsage(allData)
10:                 probMemUsage ← getavgMemUsage(probData)
11:                 display(hotspotMethod detected as Wrong Cache Strategy anti-pattern!)
12:                 display(Average Heap-Memory usage: probMemUsage vs. avgMemUsage)
13:             end if
14:         end if
15:     end for
16: end procedure
```

Conclusion

After finding out that the memory data is useless for the analysis, we can say that an automatic detection of this anti-pattern is , with our data, not possible. Also, the performance expert Mr. Avritzer told us, that the architect, who detected the problem, looked at the hotspots and then knew that the listed method does many things with the cache. So, all the further investigation were then related to the cache and also by trying to fix the problem they changed the use of the cache and had success. For us, this is a good validation. Without knowing what the specific method is doing, it is not possible to detect this kind of problems. But, we also think that this case may be special, and in other cases an automatic detection can be made. Therefore, we thought about a more basic approach to tackle this problem and hope that with future data the detection can be improved.

4.4. PADprof Tool

In this section our tool, PADprof that we developed within this research, is presented. Primarily it is described which data from YourKit can be read, later in Section 4.4.1 is explained how to use the tool. Lastly, the work flow with the tool is explicated in Section 4.4.2.



C:\Users\Alexander\Desktop\GUITest\CPU-hot-spots_th.xml

"C:\Users\Alexander\Desktop\Export08-03-207\CTH\Call-tree---All-threads-together_th.xml" "C:\Users\Al

"C:\Users\Alexander\Desktop\Export08-03-207\Wrong Cache Test\Call-tree---All-threads-together_wc.xm

✓ Circuitous Treasure Hunt | Extensive Processing | Wong Cache Strategy

Method CallCount Threshold: 25 % | Blocked Threads Count Threshold: 25 % | Method Percentage Threshold: 10 %

CPU Usage Threshold: 10 % | Method ExecTime Threshold: 10 %

Analysis method: max values

```
_________________________Circuitous Treasure Hunt results_________________________
Selected analysis method for the Method Call Count: max

*org.apache.commons.jexl3.JexlEngine.createExpression(String)* is detected!
Method count: 53.0 vs. 31.0 deviation: 70.96%
CPU Usage: 95.7% vs. 65.52% deviation: 30.18%

*java.lang.Object.notifyAll() Object.java (native)* could not been analyzed due too few data!
 The method is either not available in the problem snapshot or the comparison snapshots.

*org.apache.velocity.app.VelocityEngine.evaluate(Context, Writer, String, Reader)* is detected!
Method count: 28.0 vs. 22.0 deviation: 27.27%
CPU Usage: 95.7% vs. 65.52% deviation: 30.18%

___________________End of Circuitous Treasure Hunt results___________________
```

Figure 4.3.: Screenshot of PADprof under windows

After investigating the different snapshots and developing the detection processes, we implemented the Performance Anti-pattern Detection from profiling data (PADprof) tool. With this program all the analysis from the section before can be made and additional strategies can be added in the future. The software is an open source project and is available on GitHub [TBH+17]. PADprof can read the following exportable data from YourKit (see Section 3.3.2):

- CPU-hot-spots.xml

- Call-tree–All-threads-together.xml
- Call-tree–By-thread.xml
- Monitor-usage-statistics.xml
- Chart–CPU-time.csv
- Chart–Heap-memory.csv

These are the standard names, when the data is exported from YourKit. More information about the data and the export process itself can be found in Section 3.3. Which data is needed for the different analysis processes is noted in Table 4.1. The program, including the detection rules, are written in Java. For the XML input we used JAXB as well as opencsv for reading the CSV data. A GUI, which can be seen in Figure 4.3, is also provided in order to simplify the analysis process for the user.

4.4.1. Using PADprof

The following steps need to be made in order to analyze YourKit snapshots for anti-patterns:

1. Export the required files (see Table 4.1) from YourKit as described in Section 3.3.9
2. Select the hotspot file from the problem snapshot
3. Select the needed data for the analysis from the problem file in the second text field of the GUI
4. Lastly, select the same type of files from all the comparison snapshots with the third file selector
5. Tick which analysis should be run and additionally choose custom thresholds if the default ones do not fit
6. Press “Start Analysis”
7. After the analysis the results are shown in the text box

Side notes: The names of the exported files must not be changed. Otherwise, the program cannot detect which file is being processed. It is possible to add a postfix to the standard name. Also, currently only files exported from the YourKit version 2016.02-b46 are supported. Furthermore, in order to have the best visual experience, DPI-scaling in Windows should be disabled for PADprof. Additionally, our tool recognizes if too few data is available for a proper comparison and displays this in the output. This can be the case if, for instance, a problem method is new and is not available in older snapshots. Then there is no data to which PADprof could make a comparison.

4.4.2. Application Flow

After clicking on the start button, the program first checks if all required fields are filled before all the files are read. Because, in the GUI all files are simultaneously selected, PADprof needs to distinguish which files are available. This is done by searching for keywords in the file names which is also the reason why the names must not be changed. Now, the selected anti-pattern detection processes are run. How they work was presented in Section 4.3. There, also pseudo-code can be found. If a method is detected as a certain SPA the results of the individual method is shown in the text box. If no method at all matches the chosen thresholds, PADprof communicates this to the user in the text field. This happens also, when there is not enough data for an analysis, e.g., if a certain method cannot be found in the comparison files, because for example it is new in the system.

Chapter 5

Evaluation

In this chapter the evaluation of Performance Anti-pattern Detection from profiling data is presented.
Section 5.1 describes the questions that should be answered with this evaluation. In Section 5.2 the experimental set-up as well as the used exported data for the analysis is described. Section 5.3 lists all the results that came out of the testing process. For every analysis a summary table is provided. The threats to validity in this evaluation are described in Section 5.5. Finally, in Section 5.4, the results from the evaluation are discussed in order to rate them and give explanations for the data.

5.1. Evaluation Goals

We have developed rules and implemented PADprof in order to detect anti-patterns automatically from the exportable data of YourKit. Now, we want to evaluate if we fulfilled this goal by answering the following questions:

RQ1: Does PADprof correctly detect the anti-patterns?

This is the most logical question, because it is the main reason why PADprof was invented within this research. In this question is also integrated to check if a certain anti-pattern is no longer detected after it is fixed. Thereby, we want to investigate if the tool can also be used to identify if a refactoring of the source code has fixed the performance issue.

RQ2: How many false positive/negative results are detected?

With this question, we want to investigate, how often PADprof does not recognize an existing anti-pattern and how often it detects an anti-pattern although there is none present or not the detected one present.

RQ3: How can the detection be improved?

The last question deals with the problems and false rate of the detection. We want to find out what can be improved in order to fix the flaws of the detection that we discovered within this evaluation.

5.2. Experiment Settings

We use the first release version of PADprof for our tests. The current version of it is available on GitHub [TBH+17]. The tool is run on a Windows 10 laptop with the necessary Jave Runtime Enviroment installed.

The following snapshots are used for the analysis:

- experiment snapshot from the Circuitous Treasure Hunt anti-pattern
- experiment snapshot from the Extensive Processing anti-pattern
- experiment snapshot from the Wrong Cache anti-pattern
- experiment snapshot from the Unnecessary Processing anti-pattern
- three baseline snapshots

The experiments from the case study are used for the evaluation, additionally three baseline snapshots (data where no SPA should be included) are used as historical data for the Circuitous Treasure Hunt. That is possible because, for example, the EP anti-pattern is solved in the following, refactored snapshot (WCS). We cannot ensure that an anti-pattern was already present in an earlier snapshots, but the CTH snapshot can be used as historical data to find Extensive Processing, because it was profiled with an earlier version of the system. The UP data is just used for the comparison, because there was no detection developed for this anti-pattern.

Three test scenarios are conducted in order to be able to answer the research questions:

1. The first scenario is used to investigate if the manually found anti-pattern is also detected with our tool. Therefore, the hotspots from the problem snapshot, where the anti-pattern is present, is used and the data is compared to older snapshots.

2. In the second scenario the same hotspot data is used, but now snapshots that were recorded after the problem snapshot are compared to it. With this, we want to look if the hotspot methods got better or worse in future data. This evaluation can also be helpful in view of the utilization of PADprof, because with this kind of analysis, it is possible to check if certain methods became better over time (for example they call fewer other methods). This can be a use case when developing a system.
3. In the last scenario, we decided to make sure, that the performance issue was really solved after the refactoring. Therefore, we take the hotspots from the old problem file and use it with a snapshot of the refactored software version. If the same anti-patterns are detected, then the problem still exists, if not, it is successfully being solved in the new version. This test is also useful to find false positive results, because naturally no SPA should be included in the data, because the problem was fixed by refactoring.

All of the used data is exported as described in Section 3.3.9 and the tool is operated with the default thresholds as explained in Section 4.4.1. The analysis option for the CTH strategy is set to maximum call counts, because it is the most common analysis method as the performance engineer approved. Only in the CTH evaluation, tests are also conducted with all of the available options.

5.3. Description of Results

This section is divided in the three anti-patterns which are supported by PADprof. Every subsection is additionally split into the three evaluation scenarios that were presented in Section 5.2.

5.3.1. Circuitous Treasure Hunt

The following tests were conducted in order to evaluate the detection for the CTH anti-pattern. A summary of all results is shown in Table 5.1.

First Scenario

In order to evaluate if the Circuitous Treasure Hunt anti-pattern is detected, the baseline data is used for comparison, because the other experiment snapshots were all recorded after the CTH was fixed. The hotspots are extracted from from the CTH snapshot.

The results lists Method A as CTH and EP anti-pattern. It is listed for the CTH with a deviation of 31.7% in call counts and 55.08% in CPU usage. For the EP, the deviation is represented with 28.96% for the method time and 56.36% for the blocked threads count. This is the only method that was detected and none was listed in the WCS results. Method A represents the actual problem method of this snapshot.

An additional scenario was conducted with this SPA in order to evaluate if the anti-pattern is detected in every selectable option for the method call count (min/max/average). The circumstances are the same. The results do just change for the CTH analysis, because the different options do not affect the other strategies.

Noting has changed, in every possible option Method A is detected as the CTH anti-pattern. The deviation for the average count is 40%, for the minimum count 183.33% and for the maximum count it is the same as before (31.7%).

Second Scenario

In this scenario the comparison is among the CTH snapshot and the other three as future data. Here, the hotspots are also exported from the CTH data.

Two methods were detected in the CTH analysis. The first is the same as in the first scenario with a deviation of 74.19% in method call counts and 30.18% in CPU usage. For the other method, the call count deviation is 31.81% and the aberration for the CPU usage is the same. No method was detected in the EP or WCS section.

This scenario was also additionally conducted with the other two selectable options. In the minimum and average analysis, a new method was discovered with a deviation of 33.33% but, in comparison, just one more method is called (3 vs. 4 calls).

Third Scenario

Now, the EP snapshot is selected as the problem instance. For comparison, the CTH is used. In order to detect if the CTH anti-pattern is still present in the refactored data, the original CTH hotspots are used.

No method was detected as any anti-pattern.

5.3.2. Extensive Processing

The following tests were conducted in order to evaluate the detection for the EP anti-pattern. A summary of all results is shown in Table 5.2.

Evaluation Scenario	CTH	EP	WCS
Scenario 1: CTH vs. Baseline	Method A	Method A	—
Scenario 2: CTH vs. EP, WCS, UP	Method A, B	—	—
Scenario 3: EP vs. CTH	—	—-	—

Table 5.1.: Evaluation summary for the Circuitous Treasure Hunt analysis

First Scenario

In the first scenario, the comparison is made between the EP and the CTH snapshot. The hotspots are extracted from the Extensive Processing data.

Three methods are detected as the EP anti-pattern. For all of them the deviation for the blocked threads count is 27.11%. The deviation does not change with different methods, because the blocked thread count is a value that is dependent from the snapshot, not a certain method. The deviations of the method time for the methods C, D and E is 18.09%, 19.27% and 15.27%. Method C caused the problem in the case study. Method D and E are also detected in the WCS analysis. The method time deviation is logically the same as before (19.27% and 15.27%). The additionally displayed heap memory usage deviation is -0.15%. No method is declared as the Circuitous Treasure Hunt anti-pattern.

Second Scenario

Here, the EP snapshot is compared to the newer, refactored ones (WCS and UP). The hotspot section from the EP data is used.

In the Circuitous Treasure Hunt department, nine methods are detected including the methods C, D and E. The exact values of the detection are not listed here for every method because we think there is simple answer to that big amount of detected methods. Therefore, the exact values are not important. The thoughts about this detection are presented in the discussion part of this section (Section 5.4). Aside from this, no other no other method is detected, neither in the EP analysis nor in the WCS analysis.

Third Scenario

In the last scenario, the refactored snapshot (WCS) is compared to the EP one. For this kind of analysis we use the hotspots from EP with the WCS snapshot as the problem file.

No method is detected for any of the SPAs.

Evaluation Scenario	CTH	EP	WCS
Scenario 1: EP vs. CTH	—	Method C, D, E	Method D, E
Scenario 2: EP vs. WCS, UP	Method C, D, E + 6 others	—	—
Scenario 3: WCS vs. EP	—	—-	—

Table 5.2.: Evaluation summary for the Extensive Processing analysis

5.3.3. Wrong Cache

The following tests were conducted in order to evaluate the detection for the WCS anti-pattern. A summary of all results is shown in Table 5.3.

First Scenario

In the first scenario, the WCS snapshot is compared to the two snapshots that were recorded previously (CTH and EP). The hotspots are namely from the WCS snapshot.

Just one, the method D from Section 5.3.2, is detected as the Wrong Cache anti-pattern. The method time deviation is rated with 12.6% and the heap-memory usage difference is set to -2.92%. This was also the method that really caused the performance problem.

Second Scenario

In the second scenario, the current snapshot is compared to the one from the refactored version. Here, the WCS snapshot is compared to the UP one.

No method is detected for any of the SPAs.

Third Scenario

As before, the hotspots from the anti-pattern of this section are used, but for the problem data the snapshot from the newer and refactored version is selected. In this case, the UP snapshot is compared to the WCS one from which as well the exported hotspots are used.

One method is detected as the EP anti-pattern. Otherwise nothing more is detected, but four other methods could not be compared because too few data is available.

Evaluation Scenario	CTH	EP	WCS
Scenario 1: WCS vs. CTH, EP	—	—	Method D
Scenario 2: WCS vs. UP	—	—	—
Scenario 3: UP vs. WCS	—	Method F	—

Table 5.3.: Evaluation summary for the Wrong Cache analysis

5.4. Discussion of Results

5.4.1. First Scenario

The first scenario is the most important evaluation, because we evaluate if the anti-pattern that was manually detected in the snapshot is also detected with our tool. The scenario was conducted as the use cases of PADprof will probably be within the development process of a system. Snapshots from earlier development stages are compared to the current state of the system.

The respective method that caused the anti-pattern was detected in every snapshot which, regarding RQ1, was exactly the goal of the tool. Aside from that, in the Circuitous Treasure Hunt analysis, the same method that was detected as the CTH, is also listed in the EP results. This is the case because the blocked thread count in the snapshot is higher, as well as the method time. This could be related to the fact that the baseline data was recorded under lower load. The reason that the method time is higher in the CTH snapshot could be because the baseline data has a shorter recording (see Section 4.2) time or because the CTH anti-pattern also increases the method time besides the call count. The different analysis options for the method call count have no impact at all for the detection which does not need to be the case every time. But, if it is like that, the anti-pattern is present with a high probability.

In the Extensive Processing test, two other methods are assigned to this pattern aside from the real problem method. Both of the methods are also dedicated to the WCS anti-pattern. These two methods call each other in the call tree and are later responsible for the delay in the Wrong Cache snapshot. Hence, it seems very likely that the WCS is already present in this snapshot and the tool directly detects it.

In the WCS snapshot only one method is detected. Like explained before, it is one of the two methods that were already detected in the EP data. The reason why now only one method is detected, is that just this method is included in the hotspot section of the Wrong Cache data. But, the two others are associated, because they call one another which was approved by looking at the call tree. Otherwise, no method in this test is assigned to neither the CTH nor the EP anti-pattern which fulfills our goal.

5.4.2. Second Scenario

The second scenario was conducted in order to check, if the hotspot methods got better or worse in the newer iterations. In this case it is good when methods are detected as a certain anti-pattern, because it shows that these methods got better over time. The reason is because, if the performance variables are better in the newer version(s), then the current version is assigned to an anti-pattern, because it is worse than the newer ones.

In the Circuitous Treasure Hunt analysis the problem method was again tracked as the same anti-pattern, which is good, because that means that it had improved. Accordingly to the measurement, the method call count decreased by 43% and the CPU usage by 30% in average over the snapshots. A second method is also detected in this process. Also this method has improved over time. When selecting the other two analysis options, an additional method is detected. But because the call count is very low (4 vs. 3), the small difference is enough to fulfill the threshold of 25%. In this case, additional thresholds for a low call count or a boundary for a minimum difference could help.

In the Extensive Processing analysis, the three methods from the first test plus six other methods are surprisingly detected as the CTH anti-pattern. But by further investigation, it looks like the hotspot section from this snapshot includes methods that are on the very top of the call tree, because they execute more than a thousand other methods. This means, that if some of this called methods are improved then the call count decreases drastically because of this high amount. For example if every fifth method of the call tree calls 50% fewer methods, then a big amount of calls are saved. Hence, regarding RQ2, this result is surprising but also, with more detailed look, explainable.

In the Wrong Cache analysis no method is detected as any anti-pattern. But this does not say that there are no improvements made, because it is possible that the improvements are just not good enough for the default thresholds.

5.4.3. Third Scenario

The third scenario was made in order to exactly verify, if something got worse after refactoring. Therefore, the hotspots from the earlier analysis is used with a snapshot that is newer. This is then compared to the old one. If no anti-pattern is detected, then the performance for the hotspot methods did not got worse. Also, if no method is detected, it is positive for our tool, because normally, after the system was refactored, no performance problems should be there in comparison to the old state.

In the CTH as well as in the EP analysis, no method is detected for any anti-pattern which is what we have expected. However, in the Wrong Cache analysis, one method is assigned to the Extensive Processing anti-pattern. In the same analysis, four methods could not be analyzed because there was too few data. That, in addition to the short record length from the UP, can maybe explain why one method is detected. Because, when too few data is available, then a comparison is often not meaningful. Furthermore, the WCS snapshot is one of the longest recorded ones (see Section 4.2).

5.4.4. Conclusion

RQ1: The evaluation shows that our tool can automatically detect the anti-patterns that were discovered manually. Also it can do that earlier as with the manual detection as seen in the EP analysis where the Wrong Cache was also detected.

RQ2: Here and there more than just the problem method are detected, but most of the time a reasonable explanation could be found for that. There are some false positive results, but most of them are related to a circumstance. Because of this reason, it is important that the user also investigates the data output in order to find an answer for the detection like the the big deviation for the call count when comparing two very small values. Also, no problem method is missing in the detection wherefore the false negative rate is zero. This was expected, because we needed to use the same data as we did for the development of the tool. More about this threat is described in Section 5.5.

RQ3: In order to curtail some of the false positive results, the thresholds should be improved by adjusting them or adding more values for detection. This could help for example where the deviation in percent was high, but the amount of method calls was very low. Otherwise, we could not test PADprof with other data. Therefore the improvements are minimal because we build the detection process on this data. More about the threats to this evaluation is described in the next section.

5.5. Threats to Validity

Conclusion validity: All the analysis processes are based on the case study (Chapter 3). Hence, we had just one snapshot for each of the anti-patterns. That means that the evaluation is also made with the exact same data. Therefore, we can not give a conclusion if the anti-patterns would also be detected in other snapshots or whether it just works with the ones from the case study.

Internal validity: Because we have no control how the snapshots are recorded, it is possible that some deviations are not related to the anti-patterns itself. The different record lengths of the snapshots for examples were a problem, which was mentioned in Section 4.1.1. We also cannot ensure that every snapshot was generated with the exact same load test. Therefore, it is not possible to say, that all deviations that we have noticed in the different snapshot will occur in another one with the same anti-pattern. This circumstance is also because we have just one snapshot in order to detect special deviations in the data. The detection process is based on this aberration which means that it is possible that with the analysis we do not rely on deviations caused by the anti-pattern, instead it is based on some differences while creating the snapshots or something else.

Construct validity: This threat is very similar to the other ones. In order to develop the analysis strategies, we investigated the data we have and searched for differences between them. If now, a certain difference is not casual from the SPA, but we thought it is, then it is very likely that no other data would be detected correct. We cannot prove that our analysis is correct in any way, because we have no more data to look at.

External validity: Because we are just using YourKit as a profiler, we cannot ensure that other profilers would record data the same way. Therefore it is not possible to generalize our results for any available profiler.

The same can be said for the analysis process. The different anti-patterns are detected in the data we have, because they satisfy the thresholds we have set. But we cannot generalize it for other data, because we could not test our tool with other snapshots.

Chapter 6

Conclusion

This last chapter of the thesis, the research is summarized and then it is checked if the set goals are fulfilled. Also, some thoughts about future work are conducted. A short summary of the research is given in Section 6.1. The Retrospective (Section 6.2) deals with the goals of this research, and possible future work is presented in Section 6.3.

6.1. Summary

Throughout this thesis, we investigated the given snapshot data in YourKit in order to develop analysis strategies to detect anti-patterns in profiler data. We conducted different tests and in the end developed our tool — Performance Anti-pattern Detection from profiling data.

At first, we needed to understand what profiler data is and what we can do with it. Therefore, the profiler YourKit was investigated. There, we looked at all the data and compared the different snapshots that we had from the case study in order to detect deviations in the data. We had to investigate what is special about the different anti-pattern data. With this information, multiple tests were conducted in order to test if our ideas for the analysis can work or not.

After everything was sorted out, it was clear that not every anti-pattern can be detected with the data that we have. The Unnecessary Processing anti-pattern was directly excluded, because after looking at the data, we found no way of how it can be distinguished if a method is necessary or not. We also had our problems with the Wrong Cache anti-pattern, because all the ideas we had for analyzing failed. After all, we came up with analysis process for Circuitous Treasure Hunt, Extensive Processing and a more generalized approach for the Wrong Cache anti-pattern. We then started to formulate the rules for detection clearer in order to develop a tool that can read the exportable files from YourKit and analyzes them. The outcome of this was the tool Performance Anti-pattern Detection from profiling data. With our tool it is possible to read certain exported data from YourKit of different snapshots in order to compare them to each other. After the analysis, the tool lists methods that could have performance problems and to which of the three anti-pattern it is related to.

Now, because the tool was finished, we could evaluate how well the detection works. Therefore we thought of different test scenarios in order to cover as much as possible from the use cases of PADprof in the real world.

6.2. Retrospective

In Section 1.2, we described the goals for this thesis. Here, we discuss whether we have reached those goals.

The first goal was to evaluate YourKit and look what kind of data a snapshot holds. Additionally it should be investigated what and how the data can be exported in order to know, what can be used for the analysis in the future. In Section 3.3 all the information about the profiler are aggregated. There, Section 3.3.2 deals with the different data that can be exported and presents the various views of the GUI from YourKit. The export process is explained in Section 3.3.9. Furthermore, there is a section (3.3.10) included which describes what kind of analysis is already possible in YourKit itself.

The next goal was to research the different anti-patterns in order to get ideas for the analysis. The Software Performance Anti-Patterns which are investigated in the case study (Chapter 3) are introduced in Section 2.1. Furthermore, in the next section there are recommendations for other anti-patterns that could be supported in the future.

Based on the previous research, the next goal is about developing concepts for detecting the anti-patterns. The complete Chapter 4 deals with the ideas and analysis processes for each anti-pattern. There, for every anti-pattern a pseudo code is provided that describes the analysis. Also, all the difficulties while finding a reasonable way to detect the SPAs are explained.

The next goal uses all the findings of the previous one in order to implement our tool PADprof. It is presented in Section 4.4 where also a briefly user manual about the tool is located. The different strategies for the analysis are explained in Section 4.3.

The last goal, the evaluation of the implementation, is presented in Chapter 5. There, the research questions (Section 5.1) are presented, the results are shown (Section 5.3) and a discussion is made in order to rate the outcome (Section 5.4).

6.3. Future Work

In this section, possible future work is presented, that could not be done in the scope of this thesis.

The first recommended work is to test our tool with more, respectively other data. As explained in Section 5.5, we had not enough data in order to prove that our analysis works in every case. It was just possible to test the snapshots from which the strategies are developed. For a next step, ordinary data should be investigated to look if a system can be improved with the help of PADprof. Also, if possible, a similar case study should be made in order to investigate if the SPAs can also be detected in other data.

The second recommendation would be to first improve the analysis of the anti-patterns, especially the one for the Wrong Cache, because the approach there is fairly basic. The reasons for that are explained in Section 4.3.3. Maybe there are better ways of detecting the anti-patterns than we have done it. And secondly, extend the support to more anti-patterns. In our opinion the following ones could work, because the data should be available in YourKit [Tru11]:

- "Pipe and Filter" Architectures
- One Lane Bridge
- Excessive Dynamic Allocation
- The Ramp

For some of them, like the Excessive Dynamic Allocation, a memory snapshot could be useful. We mainly could not investigate this SPAs, because we had no example data for it in order to look for differences and to test an approach.

Another recommended work to do would be to add support for other export data than the one from the YourKit version of 2016. In the time this thesis was written, version 2017 came out which changed the structure of the XML/CSV data. Therefore, PADprof cannot read them properly. Also, some new functions were implemented, but after investigating the change log, these are no improvements that could help for our analysis. It could be also helpful for some developers to support older versions. Maybe, in the future, it is also possible to support data from other profilers.

Appendix A

Example data

In this chapter different kinds of exported data from YourKit is provided. Most of them, except “Monitor-Usage-Statistics.xml”, is generated by instrumenting PADprof.

A.1. Call-Tree-All-Threads-Together.xml

```
<view description="Call tree All threads together">
  <node call_tree="&lt;All threads&gt;" time_ms="12796" own_time_ms="">
    <node call_tree="java.lang.Thread.run()" time_ms="12781" own_time_ms="1921">
      <node call_tree="NativeMethodAccessorImpl.java (native)
           gui.GuiController.startClicked()" time_ms="7125" own_time_ms="0">
        <node call_tree="GuiController.java:207
             analysis.AnalysisExecution.&lt;init&gt;(File, List, List)" time_ms="5781"
             own_time_ms="0">
          <node call_tree="AnalysisExecution.java:42
               analysis.AnalysisExecution.generateObjects()" time_ms="5781" own_time_ms="15"/>
        </node>
        <node call_tree="GuiController.java:210
             analysis.AnalysisExecution.cthAnalysis(double, int, String)" time_ms="1000"
             own_time_ms="0"/>
        <node call_tree="GuiController.java:217 analysis.AnalysisExecution.wcAnalysis(int)"
             time_ms="187" own_time_ms="0"/>
        <node call_tree="GuiController.java:213
             analysis.AnalysisExecution.epAnalysis(double, int)" time_ms="156"
             own_time_ms="0"/>
      </node>
      <node call_tree="NativeMethodAccessorImpl.java (native)
           gui.GuiController.problemFileSelector()" time_ms="1078" own_time_ms="0">
        <node call_tree="GuiController.java:123 guiController.MainGui.getMultipleFiles()"
             time_ms="1062" own_time_ms="0">
```

```xml
          <node call_tree="MainGui.java:70
              javafx.stage.FileChooser.showOpenMultipleDialog(Window)" time_ms="1062"
              own_time_ms="1062"/>
        </node>
        <node call_tree="GuiController.java:131
            javafx.scene.control.TextInputControl.setText(String)" time_ms="15"
            own_time_ms="15"/>
      </node>
      <node call_tree="PlatformImpl.java:295 gui.GuiController$Console$$Lambda$267.run()"
          time_ms="968" own_time_ms="0">
        <node call_tree="gui.GuiController$Console.lambda$0(String)" time_ms="968"
            own_time_ms="0"/>
      </node>
      <node call_tree="NativeMethodAccessorImpl.java (native)
          gui.GuiController.comparisonFileSelector()" time_ms="859" own_time_ms="0"/>
      <node call_tree="NativeMethodAccessorImpl.java (native)
          gui.GuiController.hotspotFileSelector()" time_ms="828" own_time_ms="0"/>
    </node>
    <node call_tree="com.sun.javafx.tk.quantum.QuantumToolkit$$Lambda$41.run()"
        time_ms="15" own_time_ms="15"/>
  </node>
</view>
```

A.2. Call-Tree-By-Thread.xml

```xml
<view description="Call tree By thread">
  <node call_tree="JavaFX Application Thread group: &apos;main&apos;" time_ms="12656"
      own_time_ms="">
    <node call_tree="java.lang.Thread.run()" time_ms="12656" own_time_ms="1796">
      <node call_tree="NativeMethodAccessorImpl.java (native)
          gui.GuiController.startClicked()" time_ms="7125" own_time_ms="0">
        <node call_tree="GuiController.java:207
            analysis.AnalysisExecution.&lt;init&gt;(File, List, List)" time_ms="5781"
            own_time_ms="0"/>
        <node call_tree="GuiController.java:210
            analysis.AnalysisExecution.cthAnalysis(double, int, String)" time_ms="1000"
            own_time_ms="0"/>
        <node call_tree="GuiController.java:217 analysis.AnalysisExecution.wcAnalysis(int)"
            time_ms="187" own_time_ms="0"/>
        <node call_tree="GuiController.java:213
            analysis.AnalysisExecution.epAnalysis(double, int)" time_ms="156"
            own_time_ms="0"/>
      </node>
      <node call_tree="NativeMethodAccessorImpl.java (native)
          gui.GuiController.problemFileSelector()" time_ms="1078" own_time_ms="0">
        <node call_tree="GuiController.java:123 guiController.MainGui.getMultipleFiles()"
            time_ms="1062" own_time_ms="0"/>
    <node call_tree="GuiController.java:131
        javafx.scene.control.TextInputControl.setText(String)" time_ms="15"
        own_time_ms="15"/>
   </node>
   <node call_tree="PlatformImpl.java:295 gui.GuiController$Console$$Lambda$267.run()"
       time_ms="968" own_time_ms="0"/>
   <node call_tree="NativeMethodAccessorImpl.java (native)
       gui.GuiController.comparisonFileSelector()" time_ms="859" own_time_ms="0"/>
   <node call_tree="NativeMethodAccessorImpl.java (native)
       gui.GuiController.hotspotFileSelector()" time_ms="828" own_time_ms="0"/>
  </node>
 </node>
 <node call_tree="QuantumRenderer-0 group: &apos;main&apos; [DAEMON]" time_ms="125"
     own_time_ms="">
  <node call_tree="java.lang.Thread.run()" time_ms="125" own_time_ms="125"/>
 </node>
 <node call_tree="Thread-2 group: &apos;main&apos; [DAEMON]" time_ms="15" own_time_ms="">
  <node call_tree="com.sun.javafx.tk.quantum.QuantumToolkit$$Lambda$41.run()"
      time_ms="15" own_time_ms="15"/>
 </node>
</view>
```

A.3. CPU-Hot-Spots.xml

```
<view description="CPU hot spots">
 <node method="analysis.AnalysisExecution.generateObjects() AnalysisExecution.java"
     time_ms="5781"/>
 <node method="javax.xml.bind.helpers.AbstractUnmarshallerImpl.unmarshal(File)
     AbstractUnmarshallerImpl.java" time_ms="4437"/>
 <node method="javafx.stage.FileChooser.showOpenMultipleDialog(Window) FileChooser.java"
     time_ms="1921"/>
 <node method="javafx.scene.control.TextInputControl.appendText(String)
     TextInputControl.java" time_ms="968"/>
 <node method="javax.xml.bind.JAXBContext.newInstance(String) JAXBContext.java"
     time_ms="968"/>
 <node method="analysis.AnalysisExecution.getAverageMethodCount(ArrayList, String)
     AnalysisExecution.java" time_ms="921"/>
 <node method="analysis.CallTree.getMaxMethodCalls(String) CallTree.java" time_ms="890"/>
 <node method="javafx.stage.FileChooser.showOpenDialog(Window) FileChooser.java"
     time_ms="828"/>
 <node method="analysis.CallTree.returnAllSubnodes(Node) CallTree.java" time_ms="812"/>
</view>
```

A.4. Monitor-Usage-Statistics.xml

```
<view description="Monitor usage statistics">
 <node name="Blocked thread qtp782768455-352 native ID: 0x113C group: &apos;main&apos;"
     time_ms="1065" count="28"/>
```

```
<node name="Blocked thread qtp782768455-370 native ID: 0x116E group: &apos;main&apos;"
    time_ms="935" count="30"/>
<node name="Blocked thread qtp782768455-351 native ID: 0x113B group: &apos;main&apos;"
    time_ms="896" count="33"/>
<node name="Blocked thread qtp782768455-353 native ID: 0x113D group: &apos;main&apos;"
    time_ms="850" count="17"/>
<node name="Blocked thread qtp782768455-348 native ID: 0x1133 group: &apos;main&apos;"
    time_ms="734" count="14"/>
<node name="Blocked thread qtp782768455-362 native ID: 0x1147 group: &apos;main&apos;"
    time_ms="717" count="19"/>
<node name="Blocked thread qtp782768455-354 native ID: 0x113E group: &apos;main&apos;"
    time_ms="688" count="24"/>
<node name="Blocked thread qtp782768455-349 native ID: 0x1134 group: &apos;main&apos;"
    time_ms="657" count="32"/>
<node name="Blocked thread qtp782768455-347 native ID: 0x1132 group: &apos;main&apos;"
    time_ms="647" count="17"/>
<node name="Blocked thread qtp782768455-314 native ID: 0x1060 group: &apos;main&apos;"
    time_ms="626" count="17"/>
<node name="Blocked thread qtp782768455-346 native ID: 0x112D group: &apos;main&apos;"
    time_ms="607" count="27"/>
<node name="Blocked thread qtp782768455-157 native ID: 0xF82 group: &apos;main&apos;"
    time_ms="565" count="29"/>
<node name="Blocked thread qtp782768455-163 native ID: 0xF88 group: &apos;main&apos;"
    time_ms="526" count="15"/>
<node name="Blocked thread qtp782768455-342 native ID: 0x1129 group: &apos;main&apos;"
    time_ms="482" count="15"/>
<node name="Blocked thread qtp782768455-355 native ID: 0x113F group: &apos;main&apos;"
    time_ms="481" count="13"/>
<node name="Blocked thread qtp782768455-357 native ID: 0x1141 group: &apos;main&apos;"
    time_ms="470" count="15"/>
<node name="Blocked thread qtp782768455-358 native ID: 0x1142 group: &apos;main&apos;"
    time_ms="454" count="16"/>
<node name="Blocked thread qtp782768455-333 native ID: 0x1118 group: &apos;main&apos;"
    time_ms="370" count="16"/>
<node name="Blocked thread qtp782768455-361 native ID: 0x1145 group: &apos;main&apos;"
    time_ms="350" count="19"/>
<node name="Blocked thread qtp782768455-356 native ID: 0x1140 group: &apos;main&apos;"
    time_ms="329" count="14"/>
<node name="Blocked thread qtp782768455-160 native ID: 0xF85 group: &apos;main&apos;"
    time_ms="295" count="10"/>
<node name="Blocked thread qtp782768455-350 native ID: 0x1135 group: &apos;main&apos;"
    time_ms="281" count="14"/>
<node name="Blocked thread qtp782768455-159 native ID: 0xF84 group: &apos;main&apos;"
    time_ms="268" count="11"/>
<node name="Blocked thread qtp782768455-359 native ID: 0x1143 group: &apos;main&apos;"
    time_ms="256" count="15"/>
<node name="Blocked thread qtp782768455-332 native ID: 0x1117 group: &apos;main&apos;"
    time_ms="248" count="9"/>
```

```
  <node name="Blocked thread qtp782768455-316 native ID: 0x1062 group: &apos;main&apos;"
      time_ms="225" count="13"/>
  <node name="Blocked thread qtp782768455-398 native ID: 0x119F group: &apos;main&apos;"
      time_ms="190" count="4"/>
  <node name="Blocked thread qtp782768455-397 native ID: 0x119E group: &apos;main&apos;"
      time_ms="128" count="8"/>
</view>
```

A.5. Chart-CPU-Time.csv

"Uptime (ms)"	"CPU time (user + kernel)"	"CPU time (kernel)"	"Time spent in GC"
"166800"	"5"	"0"	"0"
"167800"	"0"	"0"	"0"
"168800"	"0"	"0"	"0"
"169800"	"0"	"0"	"0"
"170800"	"0"	"0"	"0"
"171800"	"0"	"0"	"0"
"172800"	"2"	"0"	"0"
"173800"	"0"	"0"	"0"
"174800"	"0"	"0"	"0"
"175800"	"1"	"1"	"0"
"176800"	"0"	"0"	"0"
"177800"	"0"	"0"	"0"
"178800"	"0"	"0"	"0"
"179800"	"2"	"0"	"0"
"180800"	"17"	"1"	"0"
"181800"	"27"	"0"	"1"
"182800"	"2"	"0"	"0"
"183800"	"1"	"0"	"0"
"184800"	"0"	"0"	"0"
"185800"	"1"	"0"	"0"
"186800"	"14"	"2"	"0"
"187800"	"33"	"2"	"2"
"188800"	"0"	"0"	"0"
"189800"	"26"	"3"	"1"
"190800"	"9"	"0"	"0"
"191800"	"1"	"0"	"0"
"192800"	"10"	"1"	"0"
"193800"	"26"	"3"	"1"

A.6. Chart-Heap-Memory.csv

"Uptime (ms)"	"CPU time (user + kernel)"	"CPU time (kernel)"	"Time spent in GC"
"166800"	"5"	"0"	"0"
"167800"	"0"	"0"	"0"
"168800"	"0"	"0"	"0"
"169800"	"0"	"0"	"0"
"170800"	"0"	"0"	"0"
"171800"	"0"	"0"	"0"
"172800"	"2"	"0"	"0"
"173800"	"0"	"0"	"0"
"174800"	"0"	"0"	"0"
"175800"	"1"	"1"	"0"
"176800"	"0"	"0"	"0"
"177800"	"0"	"0"	"0"
"178800"	"0"	"0"	"0"
"179800"	"2"	"0"	"0"
"180800"	"17"	"1"	"0"
"181800"	"27"	"0"	"1"
"182800"	"2"	"0"	"0"
"183800"	"1"	"0"	"0"
"184800"	"0"	"0"	"0"
"185800"	"1"	"0"	"0"
"186800"	"14"	"2"	"0"
"187800"	"33"	"2"	"2"
"188800"	"0"	"0"	"0"
"189800"	"26"	"3"	"1"
"190800"	"9"	"0"	"0"
"191800"	"1"	"0"	"0"
"192800"	"10"	"1"	"0"
"193800"	"26"	"3"	"1"

Appendix A

Bibliography

[AP10] N. Ayewah, W. Pugh. “The Google FindBugs Fixit.” In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. 2010, pp. 241–252 (cit. on p. 7).

[APM+07] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, Y. Zhou. “Evaluating static analysis defect warnings on production software.” In: *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM. 2007, pp. 1–8 (cit. on p. 6).

[BBC+10] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler. “A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World.” In: *Communications of the ACM* 53.2 (2010), pp. 66–75 (cit. on p. 7).

[CDT14] V. Cortellessa, A. Di Marco, C. Trubiani. “An approach for modeling and detecting software performance antipatterns based on first-order logics.” In: *Software and Systems Modeling* 13.1 (2014), pp. 391–432 (cit. on pp. 7, 27).

[HAT+04] H. Hallal, E. Alikacem, W. Tunney, S. Boroday, A. Petrenko. “Antipattern-based detection of deficiencies in Java multithreaded software.” In: *Proceedings of the Fourth International Conference on Quality Software (QSIC 2004)*. 2004, pp. 258–267 (cit. on p. 6).

[HG16] A. Hidiroglu, N. C. Gmbh. “Detecting Performance Anti-Patterns in Enterprise Applications by Analyzing Execution Traces Foundations and state of the art.” In: (2016), pp. 1–21 (cit. on p. 7).

[HHO+16] C. Heger, A. V. Hoorn, D. Okanović, S. Siegl, A. Wert. “Expert-Guided Automatic Diagnosis of Performance Problems in Enterprise Applications.” In: *2016 12th European Dependable Computing Conference (EDCC)*. 2016, pp. 185–188 (cit. on p. 7).

[HHOM17] C. Heger, A. van Hoorn, D. Okanović, M. Mann. “Application Performance Management: State of the Art and Challenges for the Future.” In: *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE ’17)*. ACM, 2017 (cit. on p. 1).

[HP07] D. Hovemeyer, W. Pugh. “Finding more null pointer bugs, but not too many.” In: *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM. 2007, pp. 9–14 (cit. on p. 6).

[Java documentation16] Oracle. *Memory in Java*. 2016. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html (cit. on p. 37).

[Par05] T. Parsons. “A framework for detecting performance design and deployment antipatterns in component based enterprise systems.” In: *Proceedings of the 2nd international doctoral symposium on Middleware - DSM ’05* (2005), pp. 1–5 (cit. on p. 7).

[profiler16] YourKit. *YourKit profiler*. 2016. URL: https://www.yourkit.com/ (cit. on pp. 2, 12).

[RAF04] N. Rutar, C. B. Almazan, J. S. Foster. “A comparison of bug finding tools for Java.” In: *International Symposium on Software Reliability Engineering*. 2004, pp. 245–256 (cit. on p. 6).

[SW00a] C. U. Smith, L. G. Williams. “Software Performance AntiPatterns.” In: *Computer Measurement Group Conference* December (2000), pp. 717–725 (cit. on p. 5).

[SW00b] C. U. Smith, L. G. Williams. “Software performance antipatterns.” In: *Workshop on Software and Performance*. 2000, pp. 127–136 (cit. on p. 6).

[SW02] C. U. Smith, L. G. Williams. “New software performance antipatterns: More ways to shoot yourself in the foot.” In: *Int. CMG Conference*. 2002, pp. 667–674 (cit. on p. 6).

[SW03a] C. U. Smith, L. G. Williams. “More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot.” In: *International Computer Measurement Group Conference*. 2003, pp. 717–725 (cit. on p. 6).

[SW03b] C. U. Smith, L. G. Williams. “New Software Performance AntiPatterns: EvenMore Ways to Shoot Yourself in the Foot.” In: *Computer Measurement Group Conference* (2003), pp. 717–725 (cit. on p. 6).

[TBH+17] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, H. Knoche. *PADprof open source project.* Available online: https://github.com/diagnoseIT/padprof. 2017 (cit. on pp. 4, 41, 46).

[TCL03] B. Tate, M. Clark, P. Linskey. *Bitter EJB*. Manning Publications, 2003 (cit. on p. 6).

[Tru11] C. Trubiani. “PhD Thesis in Computer Science Automated generation of architectural feedback from software performance analysis results Catia Trubiani.” In: (2011) (cit. on pp. 7, 29, 57).

[Wer13] A. Wert. “Performance problem diagnostics by systematic experimentation.” In: *Proceedings of the 18th international doctoral symposium on Components and architecture - WCOP ’13* (2013), p. 5 (cit. on pp. 2, 5, 6).

[WOHF14] A. Wert, M. Oehler, C. Heger, R. Farahbod. “Automatic detection of performance anti-patterns in inter-component communications.” In: *QoSA 2014 - Proceedings of the 10th International ACM SIGSOFT Conference on Quality of Software Architectures (Part of CompArch 2014)* (2014), pp. 3–12 (cit. on p. 7).

[YourKit Documentation] YourKit. *YourKit Java Profiler Help*. URL: https://www.yourkit.com/docs/java/help/ (cit. on pp. 12, 13, 18, 21–24, 28, 37).

All links were lastly followed on June 6, 2017.

Declaration