Exploiting Load Testing and Profiling for Performance Antipattern Detection

Catia Trubiani[a,*], Alexander Bran[b], André van Hoorn[b], Alberto Avritzer[c], Holger Knoche[d]

[a]*Gran Sasso Science Institute, Italy*
[b]*University of Stuttgart, Germany*
[c]*Independent Consultant., USA*
[d]*Kiel University, Germany*

Abstract

Context: The performance assessment of complex software systems is not a trivial task since it depends on the design, code, and execution environment. All these factors may affect the system quality and generate negative consequences, such as delays and system failures. The identification of bad practices leading to performance flaws is of key relevance to avoid expensive rework in redesign, reimplementation, and redeployment.

Objective: The goal of this manuscript is to provide a systematic process, based on load testing and profiling data, to identify performance issues with runtime data. These performance issues represent an important source of knowledge as they are used to trigger the software refactoring process. Software characteristics and performance measurements are matched with well-known performance antipatterns to document common performance issues and their solutions.

Method: We execute load testing based on the characteristics of collected operational profile, thus to produce representative workloads. Performance data from the system under test is collected using a profiler tool to create profiler snapshots and get performance hotspot reports. From such data, performance issues are identified and matched with the specification of antipatterns. Software refactorings are then applied to solve these performance antipatterns.

*Corresponding author
Email address: `catia.trubiani@gssi.it` (Catia Trubiani)

Results: The approach has been applied to a real-world industrial case study and to a representative laboratory study. Experimental results demonstrate the effectiveness of our tool-supported approach that is able to automatically detect two performance antipatterns by exploiting the knowledge of domain experts. In addition, the software refactoring process achieves a significant performance gain at the operational stage in both case studies.

Conclusion: Performance antipatterns can be used to effectively support the identification of performance issues from load testing and profiling data. The detection process triggers an antipattern-based software refactoring that in our two case studies results in a substantial performance improvement.

Keywords: Software Performance Engineering, Software Performance Antipatterns, Empirical Data, Load Testing and Profiling

1. Introduction

In the software development process it is fundamental to understand if performance requirements are fulfilled, since they represent what end users expect from the software system, and their unfulfillment might produce critical consequences [1, 2]. The performance assessment of complex software systems is not a trivial task due to many variabilities, such as workload fluctuation and resource availability [3, 4], that may occur when the system is in operation and they inevitably introduce flaws affecting the overall system quality [5, 6, 7].

The evaluation of the system design is of key relevance in the software development, since the identification of bad practices and the application of software refactorings [8, 9, 10] aim to modify the internal structure of software systems while preserving their behavior [11]. As consequence of these modifications, it is necessary to continuously assess the performance of systems and put in place a set of methodologies that allow for the detection of violations of performance requirements, thus reporting the performance flaws to software developers [12].

In the context of software performance engineering [13, 14], the idea of integrating software development characteristics and monitored performance data raises new challenges: which performance data should be carried back-and-forth between runtime and design time, which feedback should be provided to developers to support them in the diagnosis of performance results. There is an obvious trade-off in the performance evaluation of early model abstractions (where detected problems are cheaper to fix but the amount of information is limited), and late performance monitoring on running artifacts (where the results are more accurate but several constraints have been added on the structural, behavioral, and deployment aspects of a software system) [15].

In the literature, several approaches have been proposed for the performance modeling and analysis of software systems [16, 17, 18]. However, the maintenance of running systems is still very complex due to runtime variabilities (e.g., workload fluctuation and resources availability) that inevitably affect the system performance characteristics. In this setting, it is crucial to specify the expectations on performance requirements (e.g., service response time, CPU, memory and I/O utilizations, etc.), to monitor the system resources and to detect the performance flaws. These flaws trigger the performance-driven software *refactoring* process that aims to apply software modifications suitable for satisfying performance requirements. However, solving performance problems is very complex and the applicability of refactorings is not guaranteed. An initial effort in this direction can be found in [19] where the software refactoring consists of parallelizing the code to improve the system performance.

In this paper we deal with performance-driven software refactoring, we focus on systems already deployed to production while acting at the source code level [11]. In essence, our goal is to improve the system performance by modifying the internal structure of software methods without altering their functionalities. To this end, we make use of software performance antipatterns [20, 21] that represent effective instruments for keeping performance metrics under control. Performance antipatterns have been used to document: (i) common bad practices leading to performance flaws and (ii) common best practices leading to performance improvements by means of software refactorings. In our previous work we demonstrated the effectiveness of performance antipatterns, specifically: (i) we formalized the representation of antipatterns by means of first-order logic rules that express a set of system properties under which an antipattern occurs [22]; (ii) we introduced a methodology to prioritize the detected antipatterns and solve the most promising ones [23]; (iii) we introduced a model-driven approach to detect and solve antipatterns within architectural description languages (ADL) [24, 25].

In this paper we move a step forward with respect to our previous work [22, 23, 24] since we make use of load testing and profiling data to detect and solve performance flaws by analyzing a system's actual runtime behavior. The novel contributions of this paper are: (i) an approach to specify performance antipatterns that is parameterized by application execution data derived from load testing and profiling; (ii) an approach to detect performance antipatterns that is parameterized by monitoring the application execution and collecting performance measurements that are used to identify the sources of performance flaws; (iii) performance antipattern solution includes the application of software refactorings aimed to improve the performance measurements under analysis; (iv) experimentation on a real-world industrial case study where domain experts provided detection rules and refactoring actions that are used in a further and representative case study based on distributed microservices [26].

The remainder of the paper is organized as follows. Section 2 provides background information on performance antipatterns and presents the specification on a subset of them that have been detected and solved by the domain experts in our case studies. Section 3 describes our approach of detecting and solving performance antipatterns, and provides some information on the developed tool that automates the detection. The applicability of our approach is assessed by running two case studies: (i) a real-world industrial case study is illustrated in Section 4; (ii) a microservices laboratory study is presented in Section 5. The experimentation shows the threats to validity of the approach that are discussed in Section 6. Section 7 reports the related works. Finally, Section 8 concludes the paper and provides future research directions.

2. Performance antipatterns

Bad smells are defined as signs of potential problems in code. Code refactorings are defined as behavior-preserving code transformations that can help to improve the system design. Specifically, code refactorings represent solutions to bad smells [11]. Antipatterns describe a commonly occurring solution to a problem that generates decidedly negative consequences [27]. Bad smells include low-level or local problems that are usually symptoms of more global design smells such as antipatterns [28]. This means that bad smells might be indicators of the possible presence of antipatterns. For instance duplicated code, long methods, large classes, and long parameter lists represent opportunities for refactorings. However, the applied refactorings are not guaranteed to be beneficial, for example breaking a data-intensive problem to be solved in parallel may lead to bottlenecks [29]. Performance antipatterns, as the name suggests, deal with the performance issues of software systems and their specification [20] includes different aspects of a software system referring to development-related concerns (e.g., a centralized component managing most of the application business), and/or operational ones (e.g., a high utilization of hardware nodes or a low throughput of a certain service).

A systematic approach to specify bad smells has been presented in [30], where detection metrics support the enforcement of object-oriented design heuristics. Following the discussion in [30], we present in the sequel of the section the specification of performance antipatterns that can be detected from profiler data along with possible refactorings to solve them.

In our previous work [22] we provided a static detection, i.e., a logic-based formalization of antipatterns supporting the detection and solution applied to architectural models. In particular, we classified development characteristics in three main categories: static (e.g., high number of connections among software components), behavioral (e.g., high number of exchanged messages), and deployment characteristics (e.g., few nodes hosting a high number of software components). In contrast, in this paper we investigate how the development properties defined in [22] can be adapted while looking at running systems. Load testing and profiling data are collected and we further analyze the performance *measures* observed while the system is running (e.g., high utilization, low throughput, high response time). We make use of load testing and profiling data since the conjunction of these two sources of information jointly contributes to identify performance flaws.

In the following we provide a structured specification of selected performance antipatterns including problem, solution, and unbalanced forces, similarly to [27]. Unbalanced forces identify the primal forces that are ignored, misused, or overused in antipatterns. Such forces include the management of: (i) functionality (i.e., meeting the functional requirements); (ii) performance (i.e., meeting the required execution speed), and this applies to all antipatterns considered in this paper; (iii) complexity (i.e., defining the abstractions); (iv) change (i.e., controlling the evolution of software); (v) resources (i.e., controlling the use and implementation of artifacts); (vi) technology transfer (i.e., controlling the technology changes) [27].

We also report the static detection rules that have been defined in [22], and we present the rules defined to enable the detection of antipatterns from load testing and profiling data. We also abstract the refactoring actions that have been applied in our case studies to provide some hints on how to solve the detected antipatterns and actually refactor the software systems. We focus on a subset of the performance antipatterns defined in [20], since we exploit the knowledge provided by domain experts who reported the detection and solution of two performance antipatterns in our two case studies (see Sections 4 and 5). In particular, our case studies include the experimentation of these two performance antipatterns: Circuitous Treasure Hunt (CTH) and Extensive Processing (EP).

Table 1 briefly reports the description of the selected performance antipatterns. It is structured as follows: the first column shows the antipattern names; the second column describes the *problem* part, i.e., the *development* characteristics belonging to the corresponding antipattern, and the load testing data

Table 1: Specification of performance antipatterns in the literature [20].

Performance Antipatterns	Problem: Development	Problem: Performance	Solution
Circuitous Treasure Hunt (CTH)	Occurs when an object must look in several places to find information	An excessive number of DB accesses that lead to high utilization	Refactor DB queries to reduce the number of DB accesses, or introduce an adapter
Extensive Processing (EP)	Occurs when a long running process monopolizes a processor	An excessive usage of hardware and high execution time	Delegate asynchronous processing steps to other processes

leading to flaws in *performance* measures; the third column reports the *solution* part, i.e., the refactoring actions to put in place for solving the identified bad practice. These sources of information are exploited to detect and solve the corresponding antipatterns, and the system is refactored accordingly. It is worth to remark that the specification of performance antipatterns relies on quantities that are deliberately left vague, such as *high* utilization or *excessive* number of DB accesses (see Table 1), since such quantities are domain and application-specific. However, in order to actually verify these statements, it is necessary to set threshold values that establish a concrete quantification of these characteristics. Further discussion on thresholds is presented in Section 6; some heuristics to set their numerical values are reported in Appendix A.

In the remainder of this section, we discuss the key features of the selected antipatterns and how we interpreted these high-level specifications to actually use them in the extraction of development and performance data from load testing and profiling of systems under production. Differently from [22], the focus of this paper is on the empirical analysis of running systems. Therefore, performance antipatterns are specified through formulas that are derived from the empirical data collected by our two case studies.

2.1. Circuitous Treasure Hunt (CTH)

Problem. CTH typically occurs in database applications when the computation requires a large number of database requests to retrieve the data. It happens that a software instance performs a sequence of queries to obtain some data and it exploits the information retrieved in a query to construct the next one, instead of properly constructing a single, even more complex, query that gets the required information in a faster way. The performance loss is due to the overhead introduced by the cost of database accesses, processing of queries, and the transmission of all intermediate results [20].

Solution. The solution to this antipattern is to restructure or improve the queries issued to the database. This may include rewriting queries (e.g., replacing a join inside the application by a database join) and tuning fetch behavior of object-relational mappers that use lazy loading in inappropriate places. Further information and patterns for refactoring databases can be found in [31]. Another solution is to reduce the performance loss by introducing an adapter entity. This consists of creating a single software instance that handles a part of the logics needed to perform a query, and providing an interface to other objects, thus to reduce the traffic between the objects and the database query handler [20].

Unbalanced Forces. Management of performance, complexity, resources.

Static detection. The logic-based formalization [22] of the CTH antipattern is:

$$\boxed{\begin{array}{l}\exists swE_x, swE_y \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid swE_y.isDB = true \wedge \\ F_{numDBmsgs}(swE_x, swE_y, S) \geq Th_{maxDBmsgs} \wedge F_{maxHwUtil}(P_{swE_y}, all) \geq \\ Th_{maxHwUtil} \wedge F_{maxHwUtil}(P_{swE_y}, disk) > F_{maxHwUtil}(P_{swE_y}, cpu)\end{array}}$$

where sw$\mathbb{E}$ represents the set of software entity instances, and $\mathbb{S}$ represents the set of services in the software system. The formula checks if the software instance is a database sending a high number of database messages and generating a high disk utilization on the machine where the database is deployed.

Detection from runtime data. The basic idea is directly derived from the definition of the antipattern itself and the analysis of load testing and profiling data of our case studies. In particular, besides considering database requests, we also analyze the system implementation code and we check all the software methods that must look in several other ones to retrieve the needed information. CTH is detected when the CPU or disk utilizations (here denoted as hardware utilization) is high due to the fact that a method calls several other methods within its execution. To this end, for each method we store the hardware utilization observed during its execution, and we count the number of invoked methods before it is entirely executed. The CTH is detected when the hardware utilization and the number of calls to other methods are both larger than some pre-defined threshold values. Differently from [22], we do not restrict the detection to database requests and we do not distinguish the type of hardware resources, both CPU(s) and disk(s) contribute to a high resource utilization.

We formalize this interpretation by means of first-order logics, specifically we define two basic predicates: (1) the BP_{dev} predicate whose elements refer to the development characteristics, and (2) the BP_{perf} predicate whose elements belong to the performance measurements extracted from profiling data.

BP_{dev} — There is a software `method`, e.g., swM_x, that has been developed to call an excessive number of other methods, e.g. $\{swM_1, \ldots, swM_n\}$. To formalize such interpretation we use the $F_{numCalls}$ function that retrieves the number of methods called by swM_x. The bad development practice of calling an excessive number of other methods can be verified by comparing the output value of the $F_{numCalls}$ function with the $Th_{numCalls}$ threshold:

$$F_{numCalls}(swM_x) \geq Th_{numCalls} \tag{1}$$

BP_{perf} — The `method` swM_x shows a heavy computation during its operation, that is the hardware resources exceed a certain threshold, namely $Th_{hwUtilization}$. For the formalization of this operational characteristic, we use the $F_{hwUtilization}$ function that returns the maximum CPU/disk utilization among all devices while the method is in operation, and we compare such value with the $Th_{hwUtilization}$ threshold:

$$F_{hwUtilization}(swM_x) \geq Th_{hwUtilization} \tag{2}$$

Summarizing, the CTH antipattern occurs when the following predicate is true:

$$\exists swM_x \in \text{sw}\mathbb{M} \mid \boxed{(1) \land (2)}$$

where sw$\mathbb{M}$ represents the set of `Methods` in the software system. Each swM_x instance is reported to the designer for further analysis, since it is detected as an occurrence of the CTH antipattern.

Refactoring actions. The solution proposed in [20] is to refactor DB queries to reduce the number of DB accesses, or introduce an adapter (see Table 1). This includes the software refactoring of the identified methods, specifically the number of calls to other methods needs to be reduced, thus to improve the hardware utilization. In our industrial case study, we found an occurrence of the CTH antipattern due to an inefficient method for lookup information in a large data set. The lookup consisted of two queries: a first query returned a large result set that was the basis for the second query to gather the required information. The refactoring action was to refine the first query to reduce the size of extracted result set. Consequently, the second query was faster because less data had to be analyzed.

2.2. Extensive Processing (EP)

Problem. EP occurs when a long running process monopolizes a processor and prevents a set of other jobs to be executed until it finishes its computation. The processor is removed from the pool, but unlike the pipe and filter, other work does not have to pass through this stage before proceeding. This is particularly problematic if the extensive processing is on the processing path that is executed for the most frequent workload [20].

Solution. The solution to this antipattern is to identify processing steps that may cause slowdowns and delegate those steps to processes that will not impede the fast path. A performance gain could be achieved by delegating processing steps which do not need a synchronous execution to other processes [20].

Unbalanced Forces. Management of performance, change, resources.

Static detection. The logic-based formalization [22] of the EP antipattern is:

$$\boxed{\begin{array}{l}\exists OpI_1, OpI_2 \in \mathbb{O}, S \in \mathbb{S} \mid \forall i : F_{resDemand}(Op_1)[i] \geq Th_{maxOpResDemand}[i] \wedge \\ \forall i : F_{resDemand}(Op_2)[i] < Th_{minOpResDemand}[i] \wedge F_{probExec}(S, OpI_1) + \\ F_{probExec}(S, OpI_2) = 1 \wedge (F_{RT}(S) > Th_{SrtReq} \vee F_{maxHwUtil}(P_{swE_x}, all) \geq \\ Th_{maxHwUtil})\end{array}}$$

where $\mathbb{O}$ represents the set of software operations, and $\mathbb{S}$ represents the set of services in the software system. The formula checks if a software operation (whose resource demands are higher than the ones of an other operation alternatively executed) generates a high response time for the provided service or a high hardware utilization.

Detection from runtime data. Similarly to the CTH antipattern, the main idea is derived from the definition of the antipattern itself and the analysis of load testing and profiling data of our case studies. The problem arises when a long running process monopolizes a processor. So, we first look for methods generating high computation for hardware resources. We need to identify the situation of some blocked threads that prevent the execution of a method as fast as it could. Therefore, EP is detected when a method has to compete for resources showing a high number of blocked threads. The method execution time is high since other methods are monopolizing the CPU and delaying part of its computation. All these characteristics (i.e., the excessive number of blocked threads and the high execution time) are regulated by some pre-defined threshold values. Differently from [22], we do not consider the resource demand of software operations and their probability of execution since here we focus on the actual runs of methods. Besides this, we are interested in knowing their execution time instead of their predicted response time, or the utilization of the device where the software operations are deployed, as required in [22].

Similarly to the CTH antipattern, we formalize this interpretation with the BP_{dev} and BP_{perf} basic predicates.

BP_{dev} — There is a software `method`, e.g., swM_x, that is sharing hardware resources with an excessive number of blocked threads, e.g., $\{blockedThread_1,$ $\ldots, blockedThread_n\}$ that are generated by m other methods, e.g., $\{swM_1, \ldots,$ $swM_m\}$. To formalize such interpretation we use the $F_{blockedThreads}$ function that retrieves the number of blocked threads within the running of a selected method (swM_x). The bad development practice of blocking an excessive number of threads can be verified by comparing the output value of the $F_{blockedThreads}$ function with the $Th_{blockedThreads}$ threshold:

$$F_{blockedThreads}(swM_x) \geq Th_{blockedThreads} \tag{3}$$

BP_{perf} — The identified swM_x `method` is in operation meanwhile other methods ($\{swM_1, \ldots, swM_m\}$) are responsible for a high number of blocked threads, hence its execution time is delayed and results to exceed a certain threshold, namely $Th_{execTime}$. We use the $F_{methodExecTime}$ function for the formalization of this characteristic; it returns the method execution time in operation, and we compare such value with the $Th_{methodExecTime}$ threshold:

$$F_{methodExecTime}(swM_x) \geq Th_{methodExecTime} \tag{4}$$

Summarizing, the EP antipattern occurs when the following predicate is true:

$$\exists swM_x \in \mathrm{sw}\mathbb{M} \mid \boxed{(3) \wedge (4)}$$

where sw$\mathbb{M}$ represents the set of `Method`s developed and running in the software system. Each swM_x instance is reported to the designer for further analysis, since it is detected as an occurrence of the EP antipattern.

Refactoring actions. The solution proposed in [20] is to delegate asynchronous processing steps to other processes (see Table 1). This includes the software refactoring of the methods identified as blocked threads leading to prevent its processing, thus to improve the corresponding method execution time. In our case studies, we found five occurrences of the EP antipattern. Different refactoring actions on code and architectural level have been applied to solve them.

As an example on code level, caching of results has been improved to avoid re-computations of previously obtained results. On the architectural level, a refactoring action was the introduction of flow control, i.e., a well-known performance design pattern [32] aimed at avoiding performance flaws caused by overload. Flow control is a technique to prevent throughput degradation and increased delay due to congestion, and it mainly consists of fair allocation of resources among competing users. For an overview of congestion states and flow control approaches please refer to [32].

3. Our approach

Figure 1 provides the high-level picture of our approach. Input/output data is represented by square boxes, operational steps are numbered and represented by rounded boxes. Automated operational steps are denoted by cogwheels. A few steps are semi-automated, where some manual instrumentation is needed. One step is manually executed.

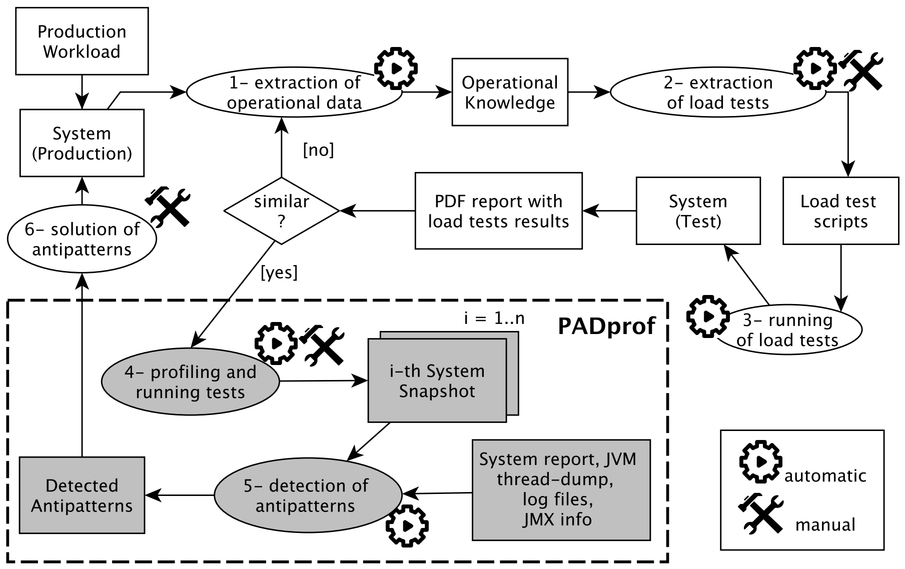
Production Workload
System (Production)
1- extraction of operational data
Operational Knowledge
2- extraction of load tests
Load test scripts
[no]
similar ?
PDF report with load tests results
System (Test)
6- solution of antipatterns
[yes]
3- running of load tests
PADprof
i = 1..n
4- profiling and running tests
i-th System Snapshot
5- detection of antipatterns
System report, JVM thread-dump, log files, JMX info
Detected Antipatterns
automatic
manual

Figure 1: Performance antipatterns detection and solution by exploiting load testing and profiling data.

Once the system is deployed to the production environment and accessed by its users, the resulting workload is referred to as the production workload or operational profile [33]. While the system is in its production use, operational data is automatically extracted (operational step 1 in our approach, see Figure 1). By operational data we mean the performance measurements obtained by Application Performance Management (APM) tools [34], e.g., HTTP request logs, application-internal execution traces, system-level measurements for CPU, memory utilization, etc. Operational knowledge is automatically obtained from the collected data, including the relevant information about the system's usage (e.g., arrival rates, usage patterns) and performance characteristics (e.g., throughput, response times, resource utilization). We use the Kieker tool [35] to derive workload and performance measurements from web server logs.

Like in the production environment, operational data is collected during the load test (operational step 2 in our approach, see Figure 1). The way the data is collected may vary. For instance, as opposed to the production environment, it is common to use fine-grained measurement tools like profilers. This step requires some manual intervention from performance testers that have to define the level of granularity for measurements. In fact, performance indices can be estimated at different levels of granularity (e.g., the response time index can be evaluated at the level of a CPU device, or at the level of a service that spans on different devices) and it is unrealistic to keep all indices at all levels of abstraction under control. After setting the granularity of measurements, load test scripts are automatically generated from the operational knowledge.

In this paper, the generation of load tests is based on our WESSBAS approach [36] that imports the logs of user requests captured during production (e.g., from the HTTP request logs of web or application servers). Each entry in the log describes an interaction of a user with the system. These logs are processed to create a probabilistic representation of the user behavior based on a domain-specific language. The WESSBAS framework allows to perform different transformations from the extracted models to load testing (LT) specifications that are executable scripts. These scripts are automatically executed with JMeter [37] that is the load driver used in this paper. Both WESSBAS and JMeter represent one possible implementation of the described operational steps. The approach itself is not limited to this specific tooling infrastructure. For instance, the extraction of load tests may rely on other workload characterization approaches that are revisited in [38], and alternative tools for running load tests are surveyed in [39].

The running of load tests (operational step 3 in our approach, see Figure 1) allows to automatically produce PDF reports with load tests results. These PDF reports are obtained by tailored scripts that process and merge the collected results of the load testing tool and the respective measurement tool (e.g., profiler). These results are automatically compared with the workload and performance characteristics of the production system using statistical means implemented in the WESSBAS framework [36]. If they are not similar, i.e., the collected data show an error percentage larger than a pre-defined error objective (e.g., 10%), then the load tests are refined to better approximate the production workload measurements. If they are similar, instead, we are able to replicate the running system and to proceed in the performance evaluation.

A major characteristic of our approach is that we use data collected during each load test from a profiler tool (operational step 4 in our approach, see Figure 1). Specifically, the profiler tool takes as input its agent that represents the probe in the application to collect the profiling data, e.g., stack traces. The profiler output results represent performance indicators, e.g., hotspots are methods that consumed the most time. In this paper, we use YourKit [40], alternative performance profiling tools are JVM Monitor [41], Oracle JMC [42], JProfiler [43], VisualVM [44]. The choice of YourKit is due to its capability of profiling applications at development stages. For each run, the profiler automatically generates a *snapshot* including the fine-grained profiling data results. This step also requires some manual intervention from performance testers that have to define the level of granularity for measurements. Through repeated load tests (e.g., with varying workload characteristics) and profiling we generate a set of n snapshots that are able to reproduce the system in production.

Table 2: Operational steps of our approach: tools, alternatives, and required expertise.

operational step	used tool	input data	output data	alterna-tives	user profile
extraction of operational data	Kieker [35]	web server logs	wkld. and perf. measurements	APM tools [34]	—
extraction of load tests	WESSBAS [36]	wkld. measurements	LT specific.	wkld. charac-terization [38]	perf. tester
running of load tests	JMeter [37]	LT specific.	LT results	LT tools [39]	—
profiling and running tests	YourKit [40], JMeter [37]	profiler agent, LT specific.	profiler results, e.g., hotspots	[41], [42], [43], [44]	perf. tester
detection of antipatterns	PADprof [45]	profiler results	PADprof report	—	—
solution of antipatterns	—	—	—	—	software developer

Our approach performs the *P*erformance *A*ntipattern *D*etection (operational step 5 in our approach, see Figure 1) based on *prof*iling data and the developed tool (PADprof), as described in Appendix B, is publicly available [45]. It focuses on how to use such snapshots, along with the system information report, JVM thread-dump, log files, and JMX information, to automatically detect antipatterns, as highlighted by the shaded boxes of Figure 1. The detected antipatterns are then manually solved (operational step 6 in our approach, see Figure 1) by providing refactoring actions to software developers that actually modify the system. Such modifications to the system are then analyzed in order to quantify the performance improvements, if any.

Table 2 provides a summary of the operational steps of our approach. In particular, in the table we report the used tools for the (semi-)automated steps, input and output data, the alternative approaches or tools that can be used, and the required profile of users for semi-automated and manual steps.

The two main activities related to the handling of performance antipatterns are: (i) the detection, i.e., the identification of bad practices during the development leading to performance flaws at the operational stage; (ii) the solution, i.e., the actual application of refactoring actions to fix bad practices thus to get performance improvements. The process of solving antipatterns is not automated, but the application of antipattern-based software refactorings shows a significant performance gain at the operational stage in our two case studies.

In the sequel of the section we describe the proposed approach to automatically detect performance antipatterns using profiler data. A high-level description of the developed tool is reported in Appendix B.

3.1. Automated detection of Circuitous Treasure Hunt (CTH)

The pseudo-code of the CTH detection procedure is reported in Algorithm 1. Firstly, the algorithm calculates the average hardware utilization values considering all snapshots and the identified problematic snapshot; in both calculations the algorithm filters all the utilization values lower than 5% (see lines 2–3 of Algorithm 1), as explained in Appendix A.

Algorithm 1 Detection of the Circuitous Treasure Hunt (CTH)

```
1: procedure CTHDETECTION($Th_{numCalls}$, $Th_{hwUtilization}$, $analysisOpt$)
2:     $avgHwUtilAllSnap \leftarrow getAvgHwUtil(allData, 5\%)$
3:     $avgHwUtilProbSnap \leftarrow getAvgHwUtil(probData, 5\%)$
4:
5:     if $avgHwUtilProbSnap > Th_{hwUtilization}$ then
6:         for all hotspotMethods do
7:             $avgMethodCount \leftarrow$ getMethodCount(allData, hotspotMethod, analysisOpt)
8:             $probMethodCount \leftarrow$ getMethodCount(probData, hotspotMethod, analysisOpt)
9:
10:            if $probMethodCount > Th_{numCalls}$ then
11:                display($hotspotMethod$ detected as Circuitous Treasure Hunt antipattern)
12:            end if
13:        end for
14:    end if
15: end procedure
```

To detect this antipattern, the algorithm checks two conditions. First, if the average hardware utilization of the identified problematic snapshot is higher than $Th_{hwUtilization}$ (see line 5 of Algorithm 1) that in our experimentation has been calculated as average over all snapshots plus 10% offset. If the first condition is verified, then it is necessary to extract the hotspot methods that might be the cause of such high hardware utilization. All hotspot methods are analyzed to identify the ones having a suspiciously high number of calls to other methods. For each hotspot method the algorithm extracts the number of method calls and there are multiple analysis options (analysisOpt) since the algorithm can store the maximum, the minimum, or the average counts for all the threads belonging to the same snapshot (see lines 7–8 of Algorithm 1). Once selected the preferred analysis option (max, min, avg) to be performed, the calculated values are then compared with the $Th_{numCalls}$ threshold (see line 10 of Algorithm 1). In our experimentation we use the average analysis option and the threshold has been calculated as the average value in all snapshots plus 25% offset. All methods fulfilling these two conditions are reported by our tool as CTH occurrences.

3.2. Automated detection of Extensive Processing (EP)

The pseudo-code of the EP detection procedure is reported in Algorithm 2. Similarly to the CTH antipattern, the algorithm firstly calculates the average blocked threads considering all snapshots and the identified problematic snapshot (see lines 2–3 of Algorithm 2).

Algorithm 2 Detection of the Extensive Processing (EP)

```
1: procedure EPDETECTION($Th_{blockedThreads}$, $Th_{methodExecTime}$)
2:     $blockedThreadsAllSnap$ ← getBlockedThreads(allData)
3:     $blockedThreadsProbSnap$ ← getBlockedThreads(probData)
4:
5:     if $blockedThreadsProb > Th_{blockedThreads}$ then
6:         for all hotspotMethods do
7:             $probMethodTime$ ← getMethodTimeInPercent(probData, hotspotMethod)
8:             $avgMethodTime$ ← getMethodTimeInPercent(allData, hotspotMethod)
9:
10:            if $probMethodTime > Th_{methodExecTime}$ then
11:                display($hotspotMethod$ detected as Extensive Processing antipattern)
12:            end if
13:        end for
14:    end if
15: end procedure
```

To detect this antipattern, the algorithm checks two conditions. First, the algorithm looks at the methods showing a number of blocked threads higher than the $Th_{blockedThreads}$ threshold (in our experimentation calculated as the average value across all snapshots plus 25% offset), see line 5 of Algorithm 2. Then, the algorithm checks for each hotspot method its execution time across the different snapshots. Since the snapshots may have a different duration, the method time is calculated in percentage looking at the complete hotspot execution time. In this way the method time is divided by the complete time of the profiling process, and it is not affecting the comparison across different snapshots (see lines 7–8 of Algorithm 2). The algorithm selects all the hotspot methods showing an execution time higher than the $Th_{methodExecTime}$ threshold (in our experimentation calculated as the average value across all snapshots plus 10% offset), see line 10 of Algorithm 2. All methods fulfilling these two conditions are reported by our tool as occurrences of the EP antipattern.

4. Industrial case study

In this section, we report our experience on the application of the proposed approach to a real-world industrial case study. It is worth to remark that this experimentation represents a preparation to automate the approach, in fact we make use of this experience to specify the antipatterns (see Section 2), and derive their detection algorithms (see Section 3). First, we describe the research goals, the methodology, and the case study setting in Section 4.1. We conduct two types of experiments and their results are presented in Sections 4.2 and 4.3. A discussion on the obtained results is reported in Section 4.4.

4.1. Goals, methodology, and case study setting

Our evaluation focuses on the detection and solution of well-known performance antipatterns on the basis of profiler snapshots generated from load tests. Particularly, we aim to answer the following research questions (RQs):

- *RQ_1:* How can known performance antipatterns be automatically detected from profiler data using detection rules?

- *RQ_2:* How effective are the software refactorings applied to solve the performance antipatterns?

- *RQ_3:* What is the quality of our antipattern-based detection rules?

The investigation of these three research questions is divided into two parts based on the following methodology:

1. First, we conduct a series of experiments in which we re-iterate the following steps: (i) load testing a version of the case study system; (ii) manually analyzing the performance problem; (iii) mapping the problem to a known performance antipattern; (iv) developing detection rules for this antipattern based on the profiler data; (v) solving the problem by applying a refactoring action; (vi) going back to the initial step while replacing the case study system under analysis with the refactored version. The process terminates when all the performance requirements are satisfied and no performance flaws are detected with the load testing.
2. Second, we assess the quality of the antipattern-based detection rules by applying them to profiler snapshots collected during the experiments.

The case study system is a popular and widely-used repository system provided by an innovative company operating in the open-source ecosystems domain. This repository system provides services to upload and download software artifacts via an HTTP-based API. In this paper, we distinguish only between GET and PUT operations. The performance requirement to be analyzed during the load tests is that the average response time for requests must not exceed the 100 milliseconds objective. The system is set up in an environment for executing performance tests which is deployed to Amazon Web Services. Apache JMeter [37] is used as the load driver, and YourKit [40] is used as the profiler tool.

4.2. The five experiment series

In this section we describe a series of five experiments conducted within this industrial case study. For each experiment we use subsequent software versions of the repository system resulting from an experiment conducted after the respectively previous experiment. As anticipated above, for each experiment we report the following steps:

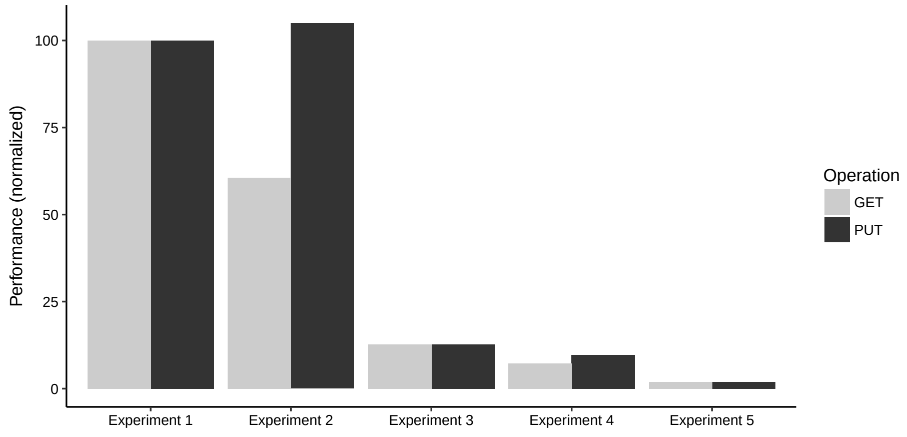

Figure 2: Industrial case study: performance results for the five experiment series.

1. Testing: execution of load tests to collect profiler snapshots and identify a snapshot showing a performance problem;
2. Analysis: manual analysis of the performance problem by a domain expert using the performance hotspots reported by the profiler;
3. Mapping: the performance problem is associated to a well-known antipattern by the domain expert;
4. Detection: development of a detection strategy for the mapped antipattern based on the profiler data. One or more profiler snapshot(s) of the improved system version are taken into account for this step;
5. Solution: development of the refactoring action(s) suitable to handle the detected performance problem by the domain expert;
6. If the performance requirements are not satisfied, repeat from step 1.

Figure 2 illustrates the normalized average response time results for each experiment. The output of each experiment is a YourKit profiler snapshot that is analyzed manually by the domain expert to map the identified performance slowdown to a known antipattern. Hereafter, we provide further details on these five experiments and we assign a descriptive label to each experiment in the respective section title, e.g., Exp1-CTH-Perm-Order for the first experiment presented in Section 4.2.1.

4.2.1. Experiment 1 — Exp1-CTH-Perm-Order

The experiment consists of running the baseline system.

- *Testing.* The average normalized performance (response time) was rated as 100 for the two analyzed operations: GETs and PUTs.

- *Analysis.* This average normalized performance metric was significantly larger than the required average delay.

- *Mapping.* This performance issue was manually mapped to the *Circuitous Treasure Hunt* (CTH) antipattern.

- *Detection.* The development of the detection strategy for the CTH antipattern has been performed as follows. The hardware utilization was the first data set to be evaluated for the detection of this antipattern, and values that represented no load conditions (i.e., below 5%) were filtered out, as explained in Appendix A. We have found that for the problem snapshot evidencing the CTH antipattern, the hardware utilization was the highest one when compared to other snapshots, attaining an average hardware utilization of 95%. In fact, the other collected snapshots reported hardware utilizations of 71%, 76%, and 18%. The maximum usage observed was 100% for the CTH snapshot, while the other values were 69% and 80%. The used threshold values are schematically reported in Table 3 where we can notice that for this antipattern the $Th_{hwUtilization}$ threshold is calculated as the average hardware utilization among all snapshots plus an offset set to 10%. The YourKit problem snapshot shows a hardware utilization higher than the threshold value. Therefore, the hardware utilization was confirmed by the domain expert as the metric to be evaluated for the automatic detection of the CTH antipattern.

Table 3: Thresholds values in the case studies.

Performance Antipatterns	Threshold	
	Name	Value
Circuitous Treasure Hunt (CTH)	$Th_{numCalls}$	$avg + 25\%$
	$Th_{hwUtilization}$	$avg + 10\%$
Extensive Processing (EP)	$Th_{blockedThreads}$	$avg + 25\%$
	$Th_{methodExecTime}$	$avg + 10\%$

Furthermore, the domain expert inspected the methods of the problem snapshot that were listed in the hotspot section and evaluated the associated method call count. We have found that for the problem snapshot evidencing the CTH antipattern, the method call count was higher when compared to the $Th_{numCalls}$ threshold (average plus 25% offset, see Table 3) in the YourKit problem snapshot. Therefore, the method calls count was also confirmed by the domain expert as the metric to be evaluated for the automatic detection of the CTH antipattern.

- *Solution.* Parsing of security permission expressions produces a large table, typically with over 5,000 rows for industrial customers, whereas user permissions contain a small number of expressions, usually less than 5. In the baseline version the large table was generated first for each security check. The action implemented to refactor the baseline version was to invert the order of parsing expressions and checking for permissions per user. Therefore, in the refactored version the list of user permissions is obtained first to create a small number of expressions parsing requests.

4.2.2. Experiment 2 — Exp2-EP-Perm-Cache

This experiment was run on the software version obtained as output from Exp1-CTH-Perm-Order where the order between parsing of expressions and checking for permissions has been inverted. This software refactoring has been applied to solve the CTH performance antipattern.

- *Testing.* The load test execution produced an average normalized performance that was rated as 60.60 for GETs and 104.93 for PUTs.

- *Analysis.* The GETs performance metric has improved significantly, but PUTs performance metric became worse.

- *Mapping.* This performance issue was manually mapped to the *Extensive Processing* (EP) antipattern.

- *Detection.* The development of the detection strategy for the EP antipattern has been performed as follows. The domain expert inspected the number of blocked threads for each snapshot and this is compared to the $Th_{blockedThreads}$ threshold (calculated as the average plus 25% offset, see Table 3). Therefore, the antipattern detection is triggered since the problem snapshot under analysis has more than 25% of blocked threads than the baseline average. Therefore, the blocked threads were confirmed by the domain expert as the metric to be evaluated for the automatic detection of the EP antipattern. Furthermore, the domain expert inspected the methods execution time and we have found that for the problem snapshot evidencing the EP antipattern, the method execution time was higher when compared to the $Th_{methodExecTime}$ threshold (calculated as average value plus 10% offset, see Table 3). Therefore, the method execution time was also confirmed by the domain expert as the metric to be evaluated for the detection of the EP antipattern.

- *Solution.* The action implemented for refactoring was a correction to a method implementing a role-permission solver cache. No additional information about this refactoring has been provided by the domain expert.

4.2.3. Experiment 3 — Exp3-EP-Reference-Type

This experiment was run on the software version obtained as output from Exp2-EP-Perm-Cache, i.e., by implementing a correction to the role-permission solver cache method. This software refactoring has been applied to solve the EP performance antipattern.

- *Testing.* The load test execution produced an average normalized performance that was rated as 12.70 for GETs and 12.63 for PUTs.

- *Analysis.* Both the GETs and PUTs performance metrics were significantly improving, but not satisfying the performance requirements.

- *Mapping.* This performance issue was mapped to the *Extensive Processing* (EP) antipattern.

- *Detection.* Our tool was able to automatically detect the Extensive Processing (EP) antipattern for one method.

- *Solution.* The applied refactoring action was a change in a method parameter specification from pass by value to pass by reference, which prevented the repeated execution of a serialization and de-serialization algorithm. More specifically, the method could use already-computed and cached objects passed by reference instead of conducting a re-computation for recurring input values.

4.2.4. Experiment 4 — Exp4-Unncessary-Method

This experiment was run on the software version obtained by Exp3-EP-Reference-Type, i.e., by implementing pass by reference instead of pass by value. This software refactoring has been applied to solve the EP performance antipattern.

- *Testing.* The load test execution produced an average normalized performance that was rated as 7.28 for GETs and 9.72 for PUTs.

- *Analysis.* Both the GETs and PUTs performance metrics were improving.

- *Mapping.* The problem was classified by the domain expert as an open ended performance issue, so we are not able to map it to any performance antipattern.

- *Solution.* The action implemented for refactoring was to remove a method that was deemed unnecessary by the domain expert.

4.2.5. Experiment 5 — Exp5-Final-Refactored-System

This experiment was run on the software version obtained from Exp4-Unncessary-Method, i.e., by removing software that was identified as unnecessary.

- *Testing.* The execution produced an average normalized performance that was rated as 1.96 for GETs and 1.89 for PUTs.

- *Analysis.* Both the GETs and PUTs performance metrics were improving, the performance objective is satisfied, no further experiments are needed.

In summary, we have conducted five experiments, we detected and solved three performance issues mapping them to known antipatterns. In particular, we detected one occurrence of the CTH antipattern and two occurrences of the EP antipattern. This experience of conducting the experiment series has been of key relevance to develop the rules to automatically detect known antipatterns based on the obtained profiler snapshots.

4.3. Analysis of the empirically derived rules

In this section, we present an analysis of the detection rules obtained from the case study by applying them to all snapshots obtained in the five experiment series. We use in our analysis the four *problem* snapshots that were used in experiments 1–4 (i.e., Exp1-CTH-Perm-Order, Exp2-EP-Perm-Cache, Exp3-EP-Reference-Type, and Exp4-Unncessary-Method) for the mapping to antipatterns. Moreover, we use three additional *baseline* snapshots and all the hotspot methods provided by YourKit.

For each detection rule, three scenarios are executed to investigate the quality of the developed rules. Each scenario is defined by a 3-tuple <*exp-hotspots, exp-snapshot-problem, exp-snapshot-baselines*>, where *exp-hotspots* refers to the experiment from which the hotspot methods are taken; *exp-snapshot-problem* is the problem snapshot evidencing a performance antipattern, and *exp-snapshots-baselines* refers to a set of baseline snapshots we compare to detect performance antipatterns.

Scenario 1 investigates if the manually identified performance antipattern is detected by the respective automatic detection rule. Therefore, the snapshot under study, using its hotspot methods, is compared to the snapshots recorded for earlier experiments. That is, the configuration for i is $< e_i, e_i, \{e_1, \ldots, e_{i-1}\} >$.

Scenario 2 investigates if the antipattern detection rule correctly detects the antipattern by comparing the problem snapshot evidencing the antipattern with snapshots that were recorded at a later time. The idea is to check that the antipattern is no longer occurring in snapshots stored for later experiments. That is, the configuration for i is $< e_i, e_i, \{e_{i+1}, \ldots, e_n\} >$.

Scenario 3 investigates whether the detection recognizes the positive impact on the refactoring applied to solve the performance antipattern, i.e., the same antipattern occurrence should not be detected in the refactored software version. Therefore, the snapshot under study, using its hotspot methods, is compared as the baseline to the snapshot of the refactored version that becomes the snapshot under study. That is, the configuration for i is $< e_i, e_{i+1}, e_i >$.

We categorize the evaluation results according to the performance antipatterns that are supported by the PADprof tool (i.e., CTH and EP) and for all the described evaluation scenarios.

Detection rules for Circuitous Treasure Hunt (CTH).

Table 4 shows the results of evaluating the detection rules of the CTH performance antipattern. In Scenario 1, the snapshot from Exp1-CTH-Perm-Order (Section 4.2.1) is compared to the three baseline snapshots. The hotspots are extracted from the snapshot of Exp1-CTH-Perm-Order. The PADprof output lists Method A[1] as CTH and EP antipattern. Method A is listed for the CTH with a deviation of 31.7% in call counts and 55.08% in hardware utilization. For the EP, the deviation is represented with 28.96% for the method time and 56.36% for the blocked threads count. Method A is the only method that was detected in this evaluation.

In Scenario 2, the Exp1-CTH-Perm-Order is compared to the three snapshots recorded at a later time, i.e., Exp2-EP-Perm-Cache, Exp3-EP-Reference-Type, and Exp4-Unncessary-Method. Two methods were detected in the analysis. The first is the same as in the first scenario with a deviation of 74.19% in method

[1]We use anonymized method names due to confidentiality reasons.

call counts and 30.18% in hardware utilization. For the other method, the call count deviation is 31.81% and the spike for hardware utilization is the same. No method was detected as the EP antipattern.

In Scenario 3, the Exp2-EP-Perm-Cache snapshot is selected as the problem instance and the Exp1-CTH-Perm-Order snapshot is used as comparison baseline. No method was detected as representative of any performance antipattern.

Evaluation Scenario	CTH	EP
1: Exp1-CTH-Perm-Order vs. Baseline	Method A	Method A
2: Exp1-CTH-Perm-Order vs. Exp2-EP-Perm-Cache, Exp3-EP-Reference-Type, Exp4-Unncessary-Method	Methods A, B	—
3: Exp2-EP-Perm-Cache vs. Exp1-CTH-Perm-Order	—	—-

Table 4: Evaluation results for the CTH detection rules.

Detection rules for Extensive Processing (EP).

Table 5 shows the results of evaluating the detection rules of the EP performance antipattern.

In Scenario 1, the evaluation is made by comparing the Exp2-EP-Perm-Cache and the Exp1-CTH-Perm-Order snapshots. The hotspots are extracted from the Exp2-EP-Perm-Cache snapshot. For the three methods categorized as the EP antipattern, the deviation for the blocked threads count was computed as 27.11%. The deviation does not change for the different methods, because the blocked thread count is a value that is dependent on the snapshot, not on the methods. The deviations for method times of the methods C, D and E were computed as 18.09%, 19.27% and 15.27%, respectively. Method C was assessed to be the root cause for the performance problem under study. No method is detected as occurrence of the CTH antipattern.

In Scenario 2, the Exp2-EP-Perm-Cache snapshot is compared to the snapshots taken from the refactored software versions labeled as Exp3-EP-Reference-Type and Exp4-Unncessary-Method. The hotspot section from the Exp2-EP-Perm-Cache shapshot data is used. Using the CTH antipattern detection rules nine methods are identified including the previously detected methods (i.e., C, D and E). Instead, using EP detection rules no methods are identified.

In Scenario 3, the snapshot from the experiment Exp3-EP-Reference-Type is compared to the snaphot from Exp2-EP-Perm-Cache. No method is detected for any of the software performance antipatterns under analysis.

Evaluation Scenario	CTH	EP
1: Exp2-EP-Perm-Cache vs. Exp1-CTH-Perm-Order	—	Methods C, D, E
2: Exp2-EP-Perm-Cache vs. Exp3-EP-Reference-Type, Exp4-Unncessary-Method	Methods C, D, E + 6 others	—
3: Exp3-EP-Reference-Type vs. Exp2-EP-Perm-Cache	—	—-

Table 5: Evaluation results for the EP detection rules.

4.4. Discussion

In this industrial case study we conducted a series of five experiments. Each experiment includes a load test and we collected YourKit profiler snapshots to identify hotspots methods that indicate performance problems.

The domain expert mapped the analyzed problem to a known performance antipattern, and we developed rules for the automated detection of the respective antipatterns from profiler snapshots. Refactoring actions have been applied by the domain expert, and the refactored system version is used for the subsequent experiment. Moreover, we also applied the developed rules to other profiler snapshots to evaluate their quality. With respect to our three research questions, we can answer as follows.

To answer RQ_1, we can consider the results of experiments 1–4 where performance slowdowns were identified and the domain expert mapped one of them to the CTH antipattern and two of them to the EP antipattern. For one experiment only, it was not appropriate to map it to an antipattern. This experience allowed us to develop the detection rules presented in Section 3, thus to automatically detect these two antipatterns from profiler data.

To answer RQ_2, we can consider the results of experiments 1–5 where the average normalized response time showed an improvement of about 50 times from Experiment 1 to Experiment 5, as shown in Figure 2. In addition, the final refactored system analyzed in Experiment 5 showed a normalized performance delay that was significantly better than the objective performance requirement. This achieved performance gain allowed the project to proceed to the next phase in the release process.

To answer RQ_3, we can consider the evaluation results reported in Section 4.3. The manually mapped antipatterns have been automatically detected by PADprof, i.e., the respective method that was identified as the root cause for the antipattern was detected for every snapshot evaluated in the respective scenario (i.e., scenario 1). Moreover, the antipatterns were correctly not reported when looking at the profiler snapshots of the refactored versions (scenarios 3). However, we could also detect false positives (scenario 2). In the snapshot with the manually mapped CTH antipattern (Exp1-CTH-Perm-Order), EP has been detected as well. This might be caused by the fact that the EP antipattern identified in the subsequent experiment is already present. In the evaluation of the EP rule, two other methods are assigned to this performance antipattern in addition to the method identified as the root cause.

To summarize, PADprof was able to automatically detect the *Circuitous Treasure Hunt* and the *Extensive Processing* antipatterns. The results indicate that performance signatures based on automated detection using hardware utilization, blocked threads, method calls and method execution times can be effectively applied to recommend software refactorings. However, application knowledge is required to judge and rank the recommendations since the results may include false positives. Additional performance gains were achieved using the domain expert knowledge of the detailed method implementations. These points represent threats to validity of the approach and a more detailed discussion is provided in Section 6.

5. Laboratory case study

In this section, we present the application of our approach to a representative microservice system that is a laboratory study. Following our industrial case study (see Section 4), we conduct a series of experiments comprising the following steps: load testing experiments, analysis of the resulting profiler data to detect antipatterns, solving the antipatterns by refactorings, and assessing the performance impact.

Section 5.1 presents the research questions of this second case study. The system under test is presented in Section 5.2, Section 5.3 describes the experiment series, and the discussion follows in Section 5.4.

5.1. Evaluation goals

Similarly to the industrial case study presented in Section 4, this case study evaluates the detection and solution of antipatterns to further assess our tool-based approach. The conducted experimentation is aimed to answer the following research questions (RQs):

- *RQ_1:* How effective are the detection rules derived from the industrial experience in this second case study?
- *RQ_2:* How effective are the software refactorings applied to solve the performance antipatterns in this second case study?
- *RQ_3:* Are there additional antipatterns and other procedures to detect them from profiler data?

5.2. System under test

As system under test, we present the Sock Shop[2] which is an open-source application representing an e-commerce web site. It has been designed to be a representative of a state-of-the-art distributed application based on the microservices architectural style [26]. Microservices are an emerging architectural style to implement large-scale distributed systems exploiting the features of cloud computing infrastructures. A recent survey highlights the Sock Shop as a candidate for a benchmark application for microservices [46]. Microservice architectures impose various challenges to software performance engineering, e.g.,

[2]https://microservices-demo.github.io/

due to the rapid release cycles and targeted short times between a code change and bringing this change into production.

The Sock Shop application includes six microservices focusing on the domain-specific functionality, namely order, payment, user, catalogue, cart, and shipping. In addition, the front-end microservice provides a UI to the application. Following the characteristics of microservice applications, different programming languages are used and the microservices have their own database. Each microservice is available as a Docker image, which is a state-of-the-art container-based virtualization technique.

Due to the limitations of the YourKit profiler used by our approach, we can only analyze the Java microservices, i.e., order, cart, and shipping. We have selected the cart microservice for our experiment. We have also modified the existing Docker scripts for deploying the application to include the YourKit profiler agent for the cart component. The other components are taken as-is from the repository server. The load driver JMeter is used to generate workload to the application via the UI service. The workload script focuses on the services involving the cart microservice, but other services are also monitored.

5.3. The three experiment series

In this section, we report on the process, refactorings, and performance results obtained for the three experiments conducted with the Sock Shop. Each experiment consisted of three load tests that were run with different load configurations. Tables 6, 7, and 8 describe, for each experiment run, the submitted load, the detected antipattern, the applied refactoring action, and the obtained performance results. The collected performance measurements include the system throughput, the cart component throughput, the cart component delay and the percentage of time used by the cart hotspot. In addition, for Tables 7 and 8 we also report on the hotspot method percentage of activation and degradation as compared to the experiment load test run baseline, which is run 3 of experiment 3.

Run	Antipattern/ Refactoring	Submitted Load	System Thro. (trans/sec)	Cart Thro. (trans/sec)	Cart Delay (ms)	Cart Hot-spot (%)
1	EP / no flow control	100 threads per second	30	1.4	64	N/A
2	EP / flow control	50 threads per second	51.5	1.6	186	73
3	EP / flow control	100 threads per second	102.2	3.2	218	47.2

Table 6: Microservices case study: Exp1 performance results.

Run	Antipattern/ Refactoring	Submitted Load	System Thro. (trans/sec)	Cart Thro. (trans/sec)	Cart Delay (ms)	Cart Hot-spot (%)	Degrada-tion (%)
1	EP / flow control	100 threads per second	133.7	34.2	244	49.3	17.9
2	EP / flow control	50 threads per second	67.0	17.2	224	66.8	35.5
3	EP / flow control	20 threads per second	27.3	7.3	395	53.5	22.13

Table 7: Microservices case study: Exp2 performance results.

Run	Antipattern/ Refactoring	Submitted Load	System Thro. (trans/sec)	Cart Thro. (trans/sec)	Cart Delay (ms)	Cart Hot-spot (%)	Degrada-tion (%)
1	EP / Hash	100 threads per second	131.3	33.5	114	48.9	17.6
2	EP / Hash	50 threads per second	66.8	16.9	188	59.3	27.9
3	EP / Hash	20 threads per second	26.5	6.8	219	31.3	baseline

Table 8: Microservices case study: Exp3 performance results.

5.3.1. Experiment Exp1-run1

- *Testing.* In experiment Exp1-run1, 100 threads were started with no overload control, i.e., no limitation on the submitted load. The system showed performance degradation in both system throughput and cart throughput, as shown in the first row of Table 6. For 100 threads with no overload control a total throughput of 30 transactions/sec was observed.

- *Analysis.* The manual analysis of the performance problem by the domain expert detected a congestion state. In such a state, throughput degradation occurs as load is increased. The first row of Table 6 shows that the achieved throughput of 30 trans/sec is significantly lower than the submitted load of 100 threads/sec, i.e., a symptom of congestion. Therefore, a need to add a flow control mechanism was identifed [32]. In the experiments that used the input rate flow control approach, the system was able to achieve a throughput of 50 trans/sec and 100 trans/sec, as shown in the second and third rows of Table 6.

- *Mapping.* This performance issue was mapped to the *Excessive Processing* (EP) antipattern.

- *Detection.* The detection approach used in this experiment employs the analysis of profiler data.

- *Solution.* The refactoring action applied to solve this antipattern is to adopt an input rate flow control approach [32], thus to ensure that the system demands are not allowed to exceed a specified rate. In this case, the rates were set to 50 trans/sec and 100 trans/sec, respectively.

5.3.2. Experiment Exp1-run2

- *Testing.* In experiment Exp1-run2, 50 threads were started with flow control. The system showed significant improvement in system and cart throughput, when compared to the previous experiment, as shown in the second row of Table 6. Experiment Exp1-run2 aims to maintain a constant offered load of one request/sec for each thread.

- *Analysis.* The manual analysis of the performance problem by the domain expert was executed by using the hotspots reported by the profiler data of run2. The method `works.weave.socks.cart.controllers.ItemsController.lambda$addToCart` has been identified as root cause of the performance problem.

- *Mapping.* This performance issue was mapped to the *Excessive Processing* (EP) antipattern.

- *Detection.* The detection approach used in this experiment was the use of the hotspot function of the YourKit tool, which showed excessive processing on the identified root cause method.

- *Solution.* The refactoring action applied to solve this antipattern was the replacement of the `addToCart` lambda function approach with a conventional hashtable lookup.

5.3.3. Experiment Exp1-run3

- *Testing.* In experiment Exp1-run3, 100 threads were started with flow control. A linear performance improvement is shown in the cart throughput when compared to the previous experiment, as shown in the third row of Table 6. Experiment Exp1-run3 aims to maintain a constant offered load of one request/sec for each thread.

- *Analysis.* The manual analysis of the performance problem by the domain expert was executed by using the hotspots reported by the profiler data of run3. The method `com.sun.proxy.$Proxy106.findByCustomerId` has been identified as root cause of the performance problem.

- *Mapping.* This performance issue was mapped to the *Excessive Processing* (EP) antipattern.

- *Detection.* The detection approach used in this experiment was the use of the hotspot function of the YourKit tool, which showed excessive processing on the identified method.

- *Solution.* Even though one method was detected as a hotspot, it was not refactored by the domain expert.

In the sequel of this section, we report on two further experiments (Exp2 and Exp3) that were run to assess the performance improvement achieved by the addition of the flow control and the lambda method refactorings.

5.3.4. Experiments Exp2 and Exp3

- *Testing.* In experiments Exp2 and Exp3, a new load test was used to produce the results shown in Tables 7 and 8 for the original system with flow control and the refactored system with flow control, respectively. The refactoring approach here employed was the replacement of the `addToCart` lambda function with the conventional hastable lookup approach.

- *Analysis.* Experiment Exp3-run3 was designated as the baseline experiment, and the manual analysis by the domain expert resulted in the following observations:

  (a) as shown in experiment Exp2-run3 (degradation column), the flow control method software version used 22.13% more method calls for the hotspot method (`com.sun.proxy.$Proxy106.findByCustomerId`) than the baseline version;

  (b) as shown in experiment Exp2-run1 and Exp3-run1, the refactored software version has a similar rate of the bottleneck method activation for both the original and the refactored versions, again shown in the degradation column;

  (c) as shown in experiments Exp2-run2 and Exp3-run2 (degradation column), the refactored version shows a decrease in the hotspot method activation of about 7.6%;

  (d) as shown in the Cart Delay column, Exp2 and Exp 3 experiments show performance improvements for the average cart delays for the refactored software version with respect to the original system.

5.4. Discussion

In this laboratory case study we conducted a series of three experiments. With respect to our three research questions, we can answer as follows.

To answer RQ_1, we have observed that the developed detection rules make use of the percentage of hotspot method activation and such rules can be effectively used to guide software refactorings. Further performance metrics affecting customer satisfaction (e.g., the system throughput) can be used as additional guidance for performance-based software refactorings.

To answer RQ_2, we have measured software performance antipattern refactoring in terms of hotspot method activation degradation (%) with respect to a software baseline. We have experienced for the same load conditions 22.13% of improvement in the rate of hotspot method activation.

To answer RQ_3, we have found overload as an additional antipattern that was solved by adding a flow control method. The extensive processing (EP) antipattern can be caused by an inefficient segment of code that needs to be refactored, or by an overload condition due to a burst of arrivals. Therefore, the overload condition antipattern can be classified as a further root cause of the excessive processing (EP) antipattern. Techniques to differentiate between performance degradation due to an overload condition or to an intrinsic software flaw have been presented in [4].

6. Threats to validity

Besides inheriting all limitations of the underlying software performance engineering, load testing and profiling analysis techniques [39, 47, 14], our approach exhibits the following threats to validity.

Efficiency and correctness from domain expertise. The process of detecting performance antipatterns is driven by the domain experts that mapped performance problems with known antipatterns and provided refactoring actions from their own experience. In our two case studies the manual architecture analysis and refactoring have been both performed in less than one day, but the efficiency of this process inevitably depends from architects' expertise. Furthermore, the correctness of detection and solution is not guaranteed; it becomes increasingly unlikely that it remains consistent, especially if other mappings/refactoring actions are provided by different domain experts. An important aspect of future work is to extend the specification of performance antipatterns by involving more than one domain expert for a case study, thus to further investigate the correctness of their knowledge. We can identify the mappings/refactoring actions that result not consistent between different domain experts, and provide support to collaboratively solve the detected inconsistencies.

Specification of threshold values. As discussed in [22], thresholds cannot be avoided in the specification of performance antipatterns. In our previous work [48] we showed that thresholds multiplicity and their estimation heavily influence both the detection and the refactoring activities. In fact, the assignment of concrete numerical values to thresholds is indeed a critical task, and many application-specific factors may contribute to set such values. In this paper, we provided some ad-hoc heuristics (see Table A.9) and we were driven by the case studies to set the concrete thresholds values. As a topic for further research, we plan to develop a methodology to dynamically set thresholds values, for example by adapting to the load found in baseline snapshots. Alternatively, we may also consider to use machine learning techniques [49] to set threshold values by inspecting the profiler performance data.

No guarantee of performance improvements. The detection and solution of one or more antipatterns does not guarantee (by itself) the absence of performance flaws. Any refactoring action coming from removing one or more antipatterns, generates a new system that has to be newly monitored in order to check whether the performance measures satisfy the stated requirements. In addition, the snapshots employed to detect antipatterns depend on specific system executions under specific workloads. Applying software refactorings for these specific cases may cause performance to worsen under different workloads in other execution scenarios. In essence, the antipattern-based refactoring approach requires a continuous process that cycles through analysis of snapshots, detection and solution of antipatterns followed by further measurements to check performance improvements, if any. In some cases antipattern solutions might even degrade the system performance, by exposing some previously unknown component interaction. For example, an antipattern solution may require to split a software method into multiple ones and execute them on different hardware platforms, however it may happen that some of these new methods need some data, thus the network can be exposed to an excessive message traffic that slowsdown the whole system application.

Reliability of antipattern-based detection. The approach used in this paper for antipattern-based detection relies on performance measurements that may be difficult to calculate. For instance, in our industrial case study we experienced some difficulties to measure the cache utilization, and this lack of knowledge inevitably affects the reliability in the detection of antipatterns. In the literature, some approaches have tackled the problem of measuring the cache usage [50, 51], but they are based on heuristic evaluations. To overcome this issue, as future work we plan to integrate these heuristics to complement our missing knowledge. This way, we aim to provide approximate estimations on the most difficult performance measurements with the goal of increasing the reliability of the antipattern-based detection. Further experimentation is needed to investigate this point and understand if heuristics are helpful in this domain.

Complexity of antipattern-based refactorings. The approach used in this paper makes use of long-running performance tests, and combines manual antipattern detection based on expert domain knowledge with rule-based automated antipattern detection. However, as experienced in our industrial case study (see Section 4), the results may include false positives and the software refactoring process is inevitably complex and largely affected by the domain experts. The performance gains we obtained from two case studies are promising and warrant the investment on additional experimentation to further validate antipattern-based software refactorings with a variety of other real-world systems.

7. Related work

Software performance antipatterns and strategies to avoid or mitigate performance flaws were first described by Smith and Williams in [21], and further refined in their later works [20, 52]. These antipatterns are technology-independent, hence they can be applied to many different implementation platforms. Java specific antipatterns are described in [53]. Antipatterns regarding specific Java technologies and aspects have also been described, as for example Java EE [54], Enterprise Java Beans [55], and Java multithreading [56]. The key characteristic of antipatterns is that their specification includes both the bad practices leading to performance flaws and the common solutions to overcome such flaws. In the following we discuss related works dealing with the identification of performance issues (Section 7.1) and their removal (Section 7.2).

7.1. Identification of performance flaws

Several approaches have been proposed in the literature to use static analysis to locate and fix bugs in software [57]. Static analysis tools like PMD [58] or FindBugs [59, 60] can be used to find potential root causes for performance antipatterns, such as resource leaks which may cause the *Ramp* performance antipattern. These tools are widely used in industrial continuous integration setups, and are thus of high practical relevance. Nistor et al. [61] present CARAMEL, i.e., a static analysis technique to detect performance bugs that waste processing time due to superfluous loop iterations.

However, most of the performance issues cannot be detected statically, but can only be observed at runtime. As noted in a study of real-world performance bugs analyzed from major open source systems [62], many bugs require inputs with both special features and sufficiently large scales to surface. This result is also supported by the findings presented in [63]. Therefore, the choice of appropriate input is crucial to discover performance issues.

Grechanik et al. [64] present an approach to detect performance bottlenecks in software systems by means of feedback-directed systematic experimentation. Their approach analyzes execution traces to generate test scripts which are likely to provoke computationally intensive executions of the system under test, thus allowing to proactively anticipate performance bottlenecks of the software. Systematic experiments are also used [65] to detect performance antipatterns in Java-based three-tier applications. The authors expose the system under test to varying workloads and observe changes to specific runtime metrics, thus to detect occurrences of performance antipatterns using a decision tree.

There are also other approaches using data to locate performance issues. Nguyen et al. [66, 67] use data from performance counters to automatically detect performance regressions. The authors use control charts to extract the relevant events from the performance counter data, and employ machine learning techniques to assign detected bottlenecks to possible causes. Avritzer and Weyuker [68] measure low-level metrics like rate of process context switching and the rates of several system calls to predict the performance impact of porting an existing application to a new operating system. Similarly to our approach, the authors rely on workloads derived from the operational use of the software system. The *diagnoseIT* approach described by Heger et al. [69] relies on trace data from application performance management (APM) tools to discover recurring performance issues including a mapping to performance antipatterns. For this purpose, knowledge from APM experts is formalized as rules which are applied to the trace data to locate instances of known performance issues. As opposed to the approach described in this paper, diagnoseIT focuses on data obtained from APM tools during production and does not consider load testing or profiling. Parsons and Murphy [70] also describe a rule-based approach to detect performance antipatterns from runtime traces and targets Java EE antipatterns.

An approach to detect performance issues in distributed systems such as web applications is presented by Sambasivan et al. [71]. The authors analyze request-flow graphs derived from end-to-end traces to detect performance regressions between different versions of an application. Wert et al. [72] focus on inter-component communication to identify communication antipatterns such as *Empty Semi-Trucks* and *Blob*, and provide a heuristic for the *Circuitous Treasure Hunt* antipattern.

7.2. Removal of performance flaws

In the recent trend of integrating development (Dev) and operations (Ops) teams, processes, and tools [73, 74], it is also important that the developers are aware of the performance consequences of their development decisions. If performance bugs or antipatterns are detected, the developers must be enabled to fix these issues. In order to increase this performance awareness, Horký et al. propose to utilize performance unit tests [75]. Recurring, automated performance tests are also endorsed in [76]. The authors illustrate how performance benchmarks can be incorporated into the continuous integration setups to ensure that performance regressions can be promptly discovered and fixed.

In [77] performance bugs found in well-known browsers (i.e., Mozilla Firefox and Google Chrome) are analyzed to learn how project members collaborate to detect and fix these bugs. Four main points are outlined: (i) techniques should be developed to improve the reproducibility of bugs; (ii) more optimized means to identify the root cause of performance bugs should be developed; (iii) collaborative root cause analysis process should be better supported; (iv) the impact of changes on performance should be analyzed, e.g., by linking automated performance test results to commits, thus to trace software changes and performance improvements. The approach presented in this paper is related to the work presented in [77], as both papers explore performance load testing results to understand the software refactorings that most likely contribute to performance gains. Nistor et al. present an empirical study on three popular code bases (i.e., Eclipse JDT, Eclipse SWT, and Mozilla) [78], which aims to investigate how performance and non-performance bugs are discovered, reported to developers, and fixed by developers. Three main findings are reported: (i) fixing performance bugs may introduce new functional bugs, similarly to fixing non-performance bugs; (ii) fixing performance bugs is more difficult than fixing non-performance bugs; (ii) unlike non-performance bugs, many performance bugs are found by code reasoning and profiling, not through direct observation of the bug's negative effects (e.g., slow code).

The issue of introducing new bugs by refactoring code is also addressed by [79]. The authors analyze a large number of refactoring operations, and results show that the percentage of faults likely induced by refactorings is relatively low (i.e., around 15%). However, there are some specific kinds of refactorings (i.e., Pull Up Method and Extract Subclass) that are very likely to induce bug fixes, in fact the percentage of fixes likely induced by such refactorings is around 40%. Our approach incrementally applies performance antipattern-based refactorings that are meant to preserve the system behavior, since they do not modify software functionalities [20]. However, as performance improvement is not guaranteed in advance, the refactored system is re-run, new performance measurements are obtained, and a new analysis is performed.

Further approaches have been proposed on refactoring for parallelism with the goal of improving the system performance, and recently the problem of parallelism in web applications is tackled [80]. Dig et al. [19, 81, 82, 83] present interactive tools to apply refactorings aimed to revise the code by converting sequential code and making use of parallel constructs that preserve the program's behavior. This goal is achieved by leveraging multi-core processors (e.g., converting sequential loops to parallel loops) or exploiting concurrency primitives introduced by recent Java versions. For instance, in [84] the authors present two tools for refactoring C# applications by converting lower-level parallel abstractions in better abstractions (e.g., from Thread-based usage to lightweight Task) and such conversions resulted in the reduction of the code bloat by 57%. Franklin et al. [85] present an approach to automatically refactor Java code by using the lambda expressions introduced with Java 8. This refactoring may improve performance as lambda expressions create further opportunities for parallelism, as for example, the optimized processing of collections. In [86] the authors propose a refactoring engine for Java programs by modifying Abstract Syntax Trees (ASTs) referring to database changes only. Such engine results an order of magnitude faster than comparable refactoring engines. Refactoring for performance has been tackled by Rieger et al. [87]. The authors report experimental results of refactoring C++ code with the goal to get performance improvements. Specifically, excessive object allocation is avoided by replacing wrapped primitives by native ones. Overbey et al. [88] aim to refactor Fortran code for high-performance computing. In particular, the proposed approach consists of manually applying performance optimizations, such as loop unrolling, to improve the system runtime performance.

Most of the approaches presented in this section operate statically on the implementation code, whereas our approach aims to identify the performance problems at runtime. In addition, the software refactoring approach presented in this paper focuses on fixing the software code identified as the performance hotspot at runtime, without analyzing the complete system implementation.

8. Conclusion

In this work, we introduced a systematic process to integrate profiling data derived from load testing results with the software refactorings required to alleviate performance flaws. The proposed approach has been applied to two case studies, i.e., an industrial case study and a lab study based on microservices. Experimental results demonstrated the effectiveness of our tool-supported approach. Performance problems were mapped, by domain experts, to two known antipatterns, i.e., Circuitous Treasure Hunt and Extensive Processing. The application of performance-based software refactorings driven by antipatterns' solution seems promising, since it showed a significant performance gain at the operational stage. This experience allowed us to investigate the application of performance antipatterns based on the analysis of load testing results and performance profiling data.

Beyond the open issues discussed as threats to validity, as future work we plan to extend our approach in the following directions. First, we aim to broaden the set of considered antipatterns, e.g., technology-specific ones [89, 90] that point out performance limitations of the underlying running environment. Second, we intend to study the usability of our approach by exposing the developed tool [45] to users with different levels of expertise. Third, we propose to apply our approach to other real-world case studies, thus to further investigate its effectiveness in multiple domains. Fourth, we plan to extend the automated antipattern detection tool with more detailed system architecture knowledge. For example, bottom-up system architecture recovery techniques based on Java traces [91] could be integrated into our approach to derive high-level performance models. Such models could then be used to guide automated software refactorings to address performance antipatterns.

9. Acknowledgements

We would like to thank the Guest Editors and all the three anonymous reviewers who provided constructive comments that helped us to considerably improve our paper. This work originated from the GI-Dagstuhl Seminar 16394 on *Software Performance Engineering in the DevOps World*, which took place in September 2016. This work has been partially supported by the German Federal Ministry of Education and Research (grant no. 01IS15004/diagnoseIT and 01IS17010/ContinuITy).

References

[1] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos, Non-functional requirements in software engineering, Vol. 5, Springer Science & Business Media, 2012.

[2] D. Petriu, M. Woodside, Analysing software requirements specifications for performance, in: Proceedings of the International Workshop on Software and Performance (WOSP), 2002, pp. 1–9.

[3] E. J. Weyuker, A. Avritzer, A metric to predict software scalability, in: Proceedings of the IEEE Symposium on Software Metrics (METRICS), 2002, pp. 152–158.

[4] A. Avritzer, A. Bondi, E. J. Weyuker, Ensuring stable performance for systems that degrade, in: Proceedings of the International Workshop on Software and Performance (WOSP), 2005, pp. 43–51.

[5] F. P. Marzullo, R. N. Porto, G. Z. da Silva, J. M. de Souza, J. R. Blaschek, An MDA approach for database profiling and performance assessment, in: Computer and Information Science, Vol. 131, Springer Science & Business Media, 2008, pp. 1–10.

[6] G. Blair, N. Bencomo, R. B. France, Models@ run.time, IEEE Computer 42 (10) (2009) 22–27.

[7] R. Calinescu, C. Ghezzi, M. Z. Kwiatkowska, R. Mirandola, Self-adaptive software needs quantitative verification at runtime, Communications of the ACM 55 (9) (2012) 69–77.

[8] T. Mens, T. Tourwé, A survey of software refactoring, IEEE Transactions on Software Engineering 30 (2) (2004) 126–139.

[9] M. V. Mäntylä, C. Lassenius, Drivers for software refactoring decisions, in: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering (ESEM), 2006, pp. 297–306.

[10] A. Ouni, M. Kessentini, H. Sahraoui, M. S. Hamdi, The use of development history in software refactoring using a multi-objective evolutionary algorithm, in: Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO), 2013, pp. 1461–1468.

[11] M. Fowler, K. Beck, Refactoring: improving the design of existing code, Addison-Wesley Professional, 1999.

[12] C. Heger, R. Heinrich, Deriving work plans for solving performance and scalability problems, in: Proceedings of the European Workshop on Performance Engineering (EPEW), 2014, pp. 104–118.

[13] C. U. Smith, L. G. Williams, Performance solutions: a practical guide to creating responsive, scalable software, Addison-Wesley Professional, 2001.

[14] C. M. Woodside, G. Franks, D. C. Petriu, The future of software performance engineering, in: Proceedings of the International Workshop on the Future of Software Engineering (FOSE), 2007, pp. 171–187.

[15] R. Mirandola, C. Trubiani, A deep investigation for QoS-based feedback at design time and runtime, in: Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 2012, pp. 147–156.

[16] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, I. Meedeniya, Software architecture optimization methods: A systematic literature review, IEEE Transactions on Software Engineering 39 (5) (2013) 658–683.

[17] V. Cortellessa, A. Di Marco, P. Inverardi, Model-Based Software Performance Analysis, Springer, 2011.

[18] H. Koziolek, Performance evaluation of component-based software systems: A survey, Performance Evaluation 67 (8) (2010) 634–658.

[19] D. Dig, A refactoring approach to parallelism, IEEE Software 28 (1) (2011) 17–22.

[20] C. U. Smith, L. G. Williams, More new software antipatterns: Even more ways to shoot yourself in the foot, in: Proceedings of the International Computer Measurement Group Conference (CMG), 2003, pp. 717–725.

[21] C. U. Smith, L. G. Williams, Software performance antipatterns, in: Proceedings of the Workshop on Software and Performance (WOSP), 2000, pp. 127–136.

[22] V. Cortellessa, A. Di Marco, C. Trubiani, An approach for modeling and detecting software performance antipatterns based on first-order logics, Software & Systems Modeling 13 (1) (2014) 391–432.

[23] C. Trubiani, A. Koziolek, V. Cortellessa, R. H. Reussner, Guilt-based handling of software performance antipatterns in Palladio architectural models, Journal of Systems and Software 95 (2014) 141–165.

[24] M. De Sanctis, C. Trubiani, V. Cortellessa, A. Di Marco, M. Flamminj, A model-driven approach to catch performance antipatterns in ADL specifications, Information & Software Technology 83 (2017) 35–54.

[25] V. Cortellessa, M. De Sanctis, A. Di Marco, C. Trubiani, Enabling performance antipatterns to arise from an adl-based software architecture,

in: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA), 2012, pp. 310–314.

[26] S. Newman, Building Microservices, O'Reilly Media, Inc., 2015.

[27] W. H. Brown, R. C. Malveau, H. W. McCormick, T. J. Mowbray, AntiPatterns: refactoring software, architectures, and projects in crisis, John Wiley & Sons, Inc., 1998.

[28] N. Moha, Y.-G. Gueheneuc, L. Duchien, A.-F. Le Meur, Decor: A method for the specification and detection of code and design smells, IEEE Transactions on Software Engineering 36 (1) (2010) 20–36.

[29] G. Pinto, A. Canino, F. Castor, G. Xu, Y. David Liu, Understanding and overcoming parallelism bottlenecks in forkjoin applications, in: Proceedings of the International Conference on Automated Software Engineering (ASE), 2017, pp. 765–775.

[30] M. Lanza, R. Marinescu, Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems, Springer Science & Business Media, 2007.

[31] S. Ambler, P. Sadalage, Refactoring Databases – Evolutionary Database Design, Addison Wesley, 2006.

[32] M. Gerla, L. Kleinrock, Flow control: A comparative survey, IEEE Transactions on Communications 28 (4) (1980) 553–574.

[33] J. D. Musa, Operational profiles in software-reliability engineering, IEEE Software 10 (2) (1993) 14–32.

[34] 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 ACM/SPEC International Conference on Performance Engineering (ICPE), 2017, pp. 429–432.

[35] A. van Hoorn, J. Waller, W. Hasselbring, Kieker: A framework for application performance monitoring and dynamic software analysis, in: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE), 2012, pp. 247–248.

[36] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, H. Krcmar, Wessbas: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems, Software & Systems Modeling (2016) 1–35 (In press. Online first: `http://dx.doi.org/10.1007/s10270-016-0566-5`).

[37] Apache JMeter, available online: `http://jmeter.apache.org`.

[38] M. C. Calzarossa, L. Massari, D. Tessera, Workload characterization: A survey revisited, ACM Computing Surveys 48 (3) (2016) 48:1–48:43.

[39] Z. M. Jiang, A. E. Hassan, A survey on load testing of large-scale software systems, IEEE Transactions on Software Engineering 41 (11) (2015) 1091–1118.

[40] YourKit Java profiler, available online: `https://www.yourkit.com`.

[41] Jvm monitor, available online: `http://jvmmonitor.org`.

[42] Oracle java mission control, available online: `http://www.oracle.com/technetwork/java/javaseproducts/mission-control/java-mission-control-1998576.html`.

[43] Jprofiler, available online: `https://www.ej-technologies.com/products/jprofiler/overview.html`.

[44] Visualvm, available online: `https://visualvm.github.io`.

[45] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, H. Knoche, PADprof open source project., available online: `https://github.com/diagnoseIT/padprof` (2017).

[46] C. M. Aderaldo, N. C. Mendonça, C. Pahl, P. Jamshidi, Benchmark requirements for microservices architecture research, in: Proceedings of the International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE), 2017, pp. 8–13.

[47] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering 38 (6) (2012) 1276–1304.

[48] D. Arcelli, V. Cortellessa, C. Trubiani, Experimenting the influence of numerical thresholds on model-based detection and refactoring of performance antipatterns, Electronic Communications of the EASST 59 (2013) 1–30.

[49] I. H. Witten, E. Frank, M. A. Hall, C. J. Pal, Data Mining: Practical machine learning tools and techniques, Morgan Kaufmann, 2016.

[50] K. Liu, G. Pinto, Y. D. Liu, Data-oriented characterization of application-level energy optimization., in: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE), 2015, pp. 316–331.

[51] G. Pinto, K. Liu, F. Castor, Y. D. Liu, A comprehensive study on the energy efficiency of java's thread-safe collections, in: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 20–31.

[52] C. U. Smith, L. G. Williams, New software performance antipatterns: More ways to shoot yourself in the foot, in: Proceedings of the International Computer Measurement Group Conference (CMG), 2002, pp. 667–674.

[53] B. Tate, Bitter Java, Manning Publications, 2002.

[54] B. Dudney, S. Asbury, J. Krozak, K. Wittkopf, J2EE AntiPatterns, Wiley Publishing, 2003.

[55] B. Tate, M. Clark, P. Linskey, Bitter EJB, Manning Publications, 2003.

[56] H. Hallal, E. Alikacem, W. Tunney, S. Boroday, A. Petrenko, Antipattern-based detection of deficiencies in Java multithreaded software, in: Proceedings of the International Conference on Quality Software (QSIC), 2004, pp. 258–267.

[57] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, W. Pugh, Using static analysis to find bugs, IEEE Software 25 (5) (2008) 22–29.

[58] N. Rutar, C. B. Almazan, J. S. Foster, A comparison of bug finding tools for Java, in: Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), 2004, pp. 245–256.

[59] 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 (PASTE), 2007, pp. 9–14.

[60] 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 (PASTE), 2007, pp. 1–8.

[61] A. Nistor, et al., CARAMEL: detecting and fixing performance problems that have non-intrusive fixes, in: Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE), 2015, pp. 902–912.

[62] G. Jin, et al., Understanding and detecting real-world performance bugs, ACM SIGPLAN Notices 47 (6) (2012) 77–88.

[63] Y. Liu, C. Xu, S.-C. Cheung, Characterizing and detecting performance bugs for smartphone applications, in: Proceedings of the International Conference on Software Engineering (ICSE), 2014, pp. 1013–1024.

[64] M. Grechanik, C. Fu, Q. Xie, Automatically Finding Performance Problems with Feedback-directed Learning Software Testing, in: Proceedings of the

International Conference on Software Engineering (ICSE), 2012, pp. 156–166.

[65] A. Wert, J. Happe, L. Happe, Supporting swift reaction: Automatically uncovering performance problems by systematic experiments, in: Proceedings of the International Conference on Software Engineering (ICSE), 2013, pp. 552–561.

[66] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, P. Flora, An industrial case study of automatically identifying performance regression-causes, in: Proceedings of the Working Conference on Mining Software Repositories (MSR), 2014, pp. 232–241.

[67] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, P. Flora, Automated detection of performance regressions using statistical process control techniques, in: Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE), 2012, pp. 299–310.

[68] A. Avritzer, E. J. Weyuker, Deriving workloads for performance testing, Software Practice and Experience 26 (6) (1996) 613–633.

[69] C. Heger, A. V. Hoorn, D. Okanović, S. Siegl, A. Wert, Expert-guided automatic diagnosis of performance problems in enterprise applications, in: Proceedings of the European Dependable Computing Conference (EDCC), 2016, pp. 185–188.

[70] T. Parsons, J. Murphy, Detecting performance antipatterns in component based enterprise systems, Journal of Object Technology 7 (3) (2008) 55–90.

[71] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, G. R. Ganger, Diagnosing performance changes by comparing request flows., in: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2011, pp. 43–56.

[72] A. Wert, M. Oehler, C. Heger, R. Farahbod, Automatic detection of performance anti-patterns in inter-component communications, in: Proceedings of the International ACM Sigsoft Conference on Quality of Software Architectures (QoSA), 2014, pp. 3–12.

[73] L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015.

[74] M. Hüttermann, DevOps for developers, Apress, 2012.

[75] V. Horký, P. Libič, L. Marek, A. Steinhauser, P. Tůma, Utilizing Performance Unit Tests to Increase Performance Awareness, in: Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE), 2015, pp. 289–300.

[76] J. Waller, N. C. Ehmke, W. Hasselbring, Including performance benchmarks into continuous integration to enable DevOps, SIGSOFT Softw. Eng. Notes 40 (2) (2015) 1–4.

[77] S. Zaman, B. Adams, A. E. Hassan, A qualitative study on performance bugs, in: Proceedings of the International Conference of Mining Software Repositories (MSR), 2012, pp. 199–208.

[78] A. Nistor, T. Jiang, L. Tan, Discovering, reporting, and fixing performance bugs, in: Proceedings of the International Working Conference on Mining Software Repositories (MSR), 2013, pp. 237–246.

[79] G. Bavota, et al., When does a refactoring induce bugs? An empirical study, in: Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM), 2012, pp. 104–113.

[80] C. Radoi, S. Herhut, J. Sreeram, D. Dig, Are web applications ready for parallelism?, in: Proceedings of the Hawaii International Conference on System Sciences (HICSS), 2017, pp. 6212–6221.

[81] C. Radoi, D. Dig, Effective techniques for static race detection in Java parallel loops, ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (4) (2015) 24.

[82] D. Dig, M. Tarce, C. Radoi, M. Minea, R. Johnson, Relooper: Refactoring for loop parallelism in Java, in: Proceedings of the ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA), 2009, pp. 793–794.

[83] D. Dig, J. Marrero, M. Ernst, Refactoring sequential Java code for concurrency via concurrent libraries, in: Proceedings of the International Conference on Software Engineering (ICSE), 2009, pp. 397–407.

[84] S. Okur, C. Erdogan, D. Dig, Converting parallel code from low-level abstractions to higher-level abstractions, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2014, pp. 515–540.

[85] L. Franklin, A. Gyori, J. Lahoda, D. Dig, LAMBDAFICATOR: from imperative to functional programming through automated refactoring, in: Proceedings of the International Conference on Software Engineering (ICSE), 2013, pp. 1287–1290.

[86] J. Kim, D. S. Batory, D. Dig, M. Azanza, Improving refactoring speed by 10x, in: Proceedings of the International Conference on Software Engineering (ICSE), 2016, pp. 1145–1156.

[87] M. Rieger, B. Van Rompaey, B. Du Bois, K. Meijfroidt, P. Olievier, Refactoring for performance: An experience report, Proc. Software Evolution 2 (9) (2007) 1–9.

[88] J. Overbey, S. Xanthos, R. Johnson, B. Foote, Refactorings for fortran and high-performance computing, in: Proceedings of the International Workshop on Software Engineering for High Performance Computing System Applications (HPCS), 2005, pp. 37–39.

[89] B. Tate, M. Clark, P. Linskey, Bitter EJB, Manning Publications Co., 2003.

[90] B. Dudney, S. Asbury, J. K. Krozak, K. Wittkopf, J2EE antipatterns, John Wiley & Sons, 2003.

[91] A. Hamou-Lhadj, E. Braun, D. Amyot, T. Lethbridge, Recovering behavioral design models from execution traces, in: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), 2005, pp. 112–121.

Appendix A. Heuristics to set threshold numerical values

Performance antipatterns include the specification of threshold values. In Table A.9 we report the heuristics that have been defined to assign concrete numerical values. These heuristics are applied when no other sources of knowledge are available to set these values.

For example, the Circuitous Treasure Hunt (CTH) antipattern includes in its specification two thresholds: (i) $Th_{numCalls}$ represents the maximum bound for the number of calls to other methods; it can be estimated as the average number of calls performed by considering profiling data, plus a certain percentage offset; (ii) $Th_{hwUtilization}$ threshold represents the maximum bound for the hardware utilization; it can be estimated as the average number of all the utilization values with reference to profiling data, plus a certain percentage offset. These offset values allow to refine thresholds from being pure average values, e.g., in our experimentation we set $Th_{numCalls}$ and $Th_{hwUtilization}$ with offsets equal to 25% and 10%, respectively.

It is worth to remark that the collection of runtime data implies that there might be irrelevant data due to the start-up of the system that invalidates some measurements. For instance, looking at the chart showing utilization values we found that the plots show an initial ramp-up. Therefore, while calculating thresholds values, the average utilization values are filtered by excluding all the ones below 5%. The obtained threshold value is hard-coded in the detection

Table A.9: Thresholds specification for detecting performance antipatterns.

Performance Antipatterns	Threshold		
	Name	Description	Heuristic
Circuitous Treasure Hunt (CTH)	$Th_{numCalls}$	Maximum bound for method calls	Average number of calls performed by a method in profiling data, plus a percentage offset
	$Th_{hwUtilization}$	Maximum bound for hardware utilization	Average utilization of hardware devices in profiling data, plus a percentage offset
Extensive Processing (EP)	$Th_{blockedThreads}$	Maximum bound for number of blocked threads	Average number of blocked threads in profiling data, plus a percentage offset
	$Th_{methodExecTime}$	Maximum bound for the execution time of methods	Average execution time of a method in profiling data, plus a percentage offset

tool, and it represents the filter applied to eliminate utilization irrelevant data. In fact, utilization values lower than 5% are considered part of the system warm-up and consequently irrelevant for the generation of performance flaws. This does not affect our analysis because when the system is tested under load, a significantly higher utilization of the system is experienced.

Appendix B. Implementation

Our antipattern detection approach is supported by a publicly available tool, namely PADprof [45]. It takes as input the profiler results provided by YourKit [40] that show features to profile CPU, memory, and threads[3].

The results are presented in respective views in YourKit's graphical user interface (GUI). The CPU view presents the results associated with different groupings of call trees and method lists (all threads together, by thread, etc.). The memory view presents statistics about heap and non-heap memory, objects, and garbage collection. The thread view shows the states (runnable, blocked, sleeping, or waiting) of the threads over time. In addition, the tread view con-

[3]`https://www.yourkit.com/java/profiler/features/`

tains, for each thread, the detailed call stack and monitors (synchronization). YourKit already includes some analysis in the respective views, e.g., the detection of *hotspots*, i.e., methods with suspiciously high CPU usage, detection of typical memory problems, and deadlocks.

The entire data set of a YourKit run can be stored in a so-called snapshot file, which can then be loaded for later analyses. Selected snapshot data can be exported from the GUI or via command line in several formats: XML, CSV, HTML, plain text, and zip files. YourKit does not perform the mapping from performance hotspots to known performance antipatterns.

PADprof reads and parses the input files, and it makes calls to the Java-based implementations of the antipattern detection rules presented in Section 3. PADprof is able to process XML and CSV files. After the analysis of the input files is conducted, a report is generated. The report includes, for each detected antipattern, statistics on the antipattern detection analysis including the suspicious methods. Further details on the PADprof implementation are publicly available [45].