

Automated Selection of Software Refactorings that Improve Performance

Nikolai Moesus¹, Matthias Scholze¹, Sebastian Schlesinger² and Paula Herber²

¹*QMETHODS – Business & IT Consulting GmbH, Berlin, Germany*

²*Software and Embedded Systems Engineering, Technische Universität Berlin, Germany*

Keywords: Software Refactoring, Performance, Static Analysis, Profiling.

Abstract: Performance is a critical property of a program. While there exist refactorings that have the potential to significantly increase the performance of a program, it is hard to decide which refactorings effectively yield improvements. In this paper, we present a novel approach for the automated detection and selection of refactorings that are promising candidates to improve performance. Our key idea is to provide a heuristics that utilizes software properties determined by both static code analyses and dynamic software analyses to compile a list of concrete refactorings sorted by their assessed potential to improve performance. The expected performance improvement of a concrete refactoring depends on two factors: the execution frequency of the respective piece of code, and the effectiveness of the refactoring itself. To assess the latter, namely the general effectiveness of a given set of refactorings, we have implemented a set of micro benchmarks and measured the effect of each refactoring on computation time and memory consumption. We demonstrate the practical applicability of our overall approach with experimental results.

1 INTRODUCTION

Performance issues are a common reason why software needs refactoring. However, due to the variety of possible causes for performance issues, finding an appropriate refactoring is a complicated task. To tackle this problem, antipattern detection as well as measurement-based performance engineering have been proposed. Antipattern detection is a static analysis that aims at detecting code flaws and violations against good practice (Louridas, 2006). While antipattern detection often succeeds in detecting critical code sections that cause performance bottlenecks, it tends to yield a great number of proposed refactorings of which only a very small fraction can be considered relevant for performance. To resolve all proposed issues is therefore neither efficient nor feasible. Additionally, a bad performing piece of code that gets only rarely called usually is not the cause of severe performance problems. Measurement-based performance engineering relies on dynamic analysis techniques that are applied when the program under development is running (Woodside et al., 2007). Those analyses generate huge amounts of heterogeneous data like response times, function call durations, stack traces, memory footprints or hardware counters. To find the important chunks of information that help

solving performance issues takes time and also requires skill and experience. Hence, manually searching for an appropriate refactoring is an expensive task.

In this paper, we present a novel approach for the automated selection of refactorings that are promising candidates to improve performance. The key idea of our approach is to use both static and dynamic analyses and combine the results to generate a heuristics that determines those refactorings that are most promising with respect to performance. Our major contributions are twofold: First, to quantify the expected effect of a refactoring, we present a novel rating function that incorporates the analysis data from both static and dynamic analysis, and thus enables us to heuristically assess the effectiveness of concrete refactorings. The output is a list of proposed refactorings sorted by their potential to yield a strong positive effect on performance. The reasoning behind our heuristics is to assess which portions of source code get executed frequently, such that a refactoring there pays off more than anywhere else. We combine this with a factor that provides an estimate for the general effectiveness of a given refactoring, independent of its position in the code. Second, we present an evaluation of the general effectiveness of a given set of refactorings that are generally assumed to improve performance, independent of their use in a concrete

program. To achieve this, we have implemented micro benchmarks and measured their effectiveness with respect to the execution time and memory consumption. We use the results from our micro benchmarks together with static and dynamic code properties in our ranking function to provide a heuristics that automatically assesses the expected effect of a concrete refactoring in a given program.

To demonstrate the practical applicability of our approach, we present the results from two experiments. In the first experiment, we have intentionally manipulated a given example program such that it contains typical antipatterns and investigate the impact on performance and how high the rule violations are rated by our heuristics. In the second experiment, we apply our rating function to a given program, implement the top rated refactorings and examine the performance improvement.

The rest of this paper is structured as follows: In Section 2, we introduce the preliminaries that are necessary to understand the remainder of this paper. In Section 3, we discuss related work. In Section 4, we present our approach for the automated selection of refactorings. In Section 5, we present our micro benchmark results as well as our case studies and experimental results. We conclude in Section 6.

2 BACKGROUND

In this section, we introduce the preliminaries that are necessary to understand the remainder of this paper, namely software antipatterns and refactorings.

2.1 Software Antipatterns

Software design patterns describe good solutions to recurring problems in an abstract and reusable way. A software antipattern is very similar, only that it describes a bad, unfavored solution (Long, 2001). The motivation to write those down is to prevent their use and to provide appropriate refactorings into better designs. An example for an antipattern described in (Smith and Williams, 2002) is *The Ramp*, where tasks have an increasing execution time due to a growing list that has to be searched but is never cleaned up.

Performance antipatterns are the class of patterns that lead to bad performance. As an example for a performance antipattern in the programming language Java, consider Listing 1. In this example, strings are concatenated with the '+' operator. The reason why using the + operator as shown is considered an antipattern is based on the internal implementation in Java. The + operator is natively overloaded for

String, although in Java operator overloading in general is not possible. However, this piece of syntactic sugar brings along a disadvantage concerning performance. Because objects of String are immutable, the additional characters cannot simply be appended. Instead, internally a Java StringBuilder object is allocated, concatenates the strings in its char buffer and returns the new immutable string. If such procedure is repeated in a loop as shown in Listing 1, each iteration allocates and dismisses a StringBuilder and an intermediate string. It is veiled from the programmer that there lies an inefficiency in the simple + syntax.

Listing 1: Example Performance Antipattern.

```

1 String [] p = { "These", "are", "
   separate", "parts" };
2 String str = p[0];
3 for (int i = 1; i < p.length; ++i)
4     {
5         str = str + " " + p[i];
6     }

```

2.2 Software Refactorings

A software refactoring is a change in source code that keeps the external behavior of a software unaffected but yet improves the internal design or other non-functional properties (Fowler and Beck, 1999). Examples for refactorings are splitting up large classes into multiple units, increasing encapsulation of classes or replacing inefficient operations. Refactoring is a structured process with specified steps and a defined goal. Due to the structured procedure, refactorings are an elegant way of improving software.

A refactoring may be a large scale operation that affects several units and takes much effort to fully implement. In this paper, we focus on micro refactorings (Owen, 2016), which affect only a few lines of code and are realizable in a short time or even automatically. Concerning performance, micro refactorings can have noticeable benefits, especially in often called functions or inside frequently executed loops. Therefore, micro refactorings have the potential of an excellent cost-benefit ratio.

The proposed performance refactoring for the example in Listing 1 is to allocate only one StringBuilder outside the loop and use it instead of the + operator, such that no temporary objects accumulate.

3 RELATED WORK

In (Tsantalis et al., 2006; Washizaki et al., 2009), the authors present approaches to automatically detect

software design patterns based on static information extracted from Java bytecode. However, they focus on the detection of classical design patterns (Gamma et al., 1995) and are not concerned with their effect on performance. Similarly, in (Wierda et al., 2007), the authors examine class relations in a formal concept analysis to detect repeating patterns without the need of prior pattern knowledge. The analysis is expensive, though, and is not feasible for a large code base. Additionally, the impact of the detected patterns on performance is not considered.

According to the survey on design pattern detection presented in (Rasool and Streitfeldt, 2011), the majority of published approaches combines structural and behavioral analyses of the software. Although behavioral analyses are not necessarily implemented as dynamic analyses, for example, the approach introduced in (Heuzeroth et al., 2003) uses static and dynamic analyses similar to how we use them: The static analysis provides a set of design pattern candidates, which is narrowed down in a dynamic analysis. For each design pattern they prepare a set of rules concerning the interaction of classes and discard every candidate that violates any of the rules. In (Wendehals, 2003), the authors search both abstract syntax graphs and call graphs for design patterns and rate each candidate. The combination of both ratings helps to determine actual design patterns. Although both approaches combine static and dynamic analyses, they again only detect patterns and are not concerned with their effect on performance.

In (Bernardi et al., 2013), the authors detect design patterns in a graph representation according to a meta model. They develop a domain specific language that allows precise definitions of patterns with inheritance between them to ease the creation of variants. They achieve a high detection precision but again, they are not concerned with the effect on performance.

An approach to detect performance antipatterns and suggest refactorings is proposed in (Arcelli et al., 2012; Arcelli et al., 2015). However, they work on software architectural models, while we focus on implementations.

In (Cortellessa et al., 2010), the authors achieve a rating of performance antipatterns based on their so called guiltiness. The algorithm requires a complete set of antipatterns and the set of performance requirements for the system as input. Each antipattern and requirement is associated to one or more system entities, e.g. a processor. Depending on to what extend a requirement is not fulfilled the associated system entities spread the guilt among all their associated antipatterns while taking into account the antipattern's estimated impact on the respective system entity. Alt-

hough this approach succeeds in selecting the most effective performance antipatterns, it requires an expensive modeling step to capture the component model.

In (Djoudi et al., 2005), the authors propose assembly code optimization by means of static antipattern detection and dynamic value profiling. They use a knowledge database for assembly antipatterns that have shown bad performance in micro benchmarks and attempt to find those with a static analyzer. The dynamic analysis benefits from very low instrumentation cost on the assembly level and captures data like cache miss rate. Although this approach is closely related to ours in many ways, e.g. the focus on micro refactorings, it utilizes the dynamic analysis as independent addition instead of combining its yield with the results from static analyses, and it does not target a high-level programming language, which is often preferable for software evolution and maintenance.

In (Luo et al., 2017), the authors present a machine learning system that examines execution traces of the software under test and calculates new input values for the next execution that are most promising to uncover a performance bottleneck. Finally, an analysis of the captured execution traces is carried out and a ranked list with presumed performance bottlenecks is compiled. Even though in our approach we utilize very different techniques, the result, namely an ordered list of specific performance issues, is similar. However, they do not automatically propose a solution to the detected performance issue.

In (Fontana and Zanoni, 2017), the authors use supervised learning to train a model that finds antipatterns and rates their severity. This relieves them from the necessity to formalize the antipatterns in order to perform the detection. However, they rely on external detection algorithms to support the generation of training data, which is tedious work. Additionally, they do not focus on performance and therefore propose no measure to determine the impact of antipatterns on performance.

To the best of our knowledge, no existing approach enables the automatic selection of refactorings that are most promising to increase performance.

4 AUTOMATED SELECTION OF REFACTORINGS

Static code analyses for antipattern detection issue too many alleged defects in a not prioritized fashion, rendering the information hard to work with efficiently. Dynamic software analyses, on the other hand, yield a lot of heterogeneous data which is not easy to inter-

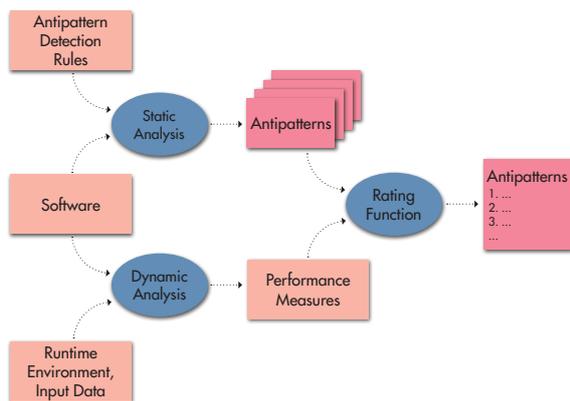


Figure 1: Automated Selection Approach.

pret and will not directly lead to a refactoring proposition in the source code. Apparently, both techniques have their individual disadvantages.

To overcome these problems, we propose an approach for the automated selection of refactorings that utilizes software properties determined by both static code analyses and dynamic software analyses. By combining the best out of both worlds into one heuristics, we compile a list of concrete refactorings sorted by their assessed potential to improve performance.

Our key idea is that with the help of dynamically retrieved runtime information we rank statically detected antipatterns by their importance regarding the expected impact on performance. By connecting runtime data with specific antipatterns in the code, we derive a precise recommendation which refactorings are most promising to improve the performance.

Figure 1 shows our overall approach. In the top left, a static analysis takes the software source code and a set of antipattern detection rules as input to produce an unordered list of antipatterns, e.g. the undesired use of the '+' operator. In the bottom left, a dynamic analysis examines the software while it is executed in its runtime environment consuming some input data. Various performance measures are the output of this process, e.g. the execution time. In the final step, we introduce a rating function, which uses the dynamic performance measures together with a factor that measures the general effectiveness of a given refactoring to assign a severity value to each statically detected antipattern. As a result, we get an ordered list where the top entries represent the antipatterns along with their proposed refactorings that have the highest potential of improving performance. Thus, there is no more need to manually handle neither the performance measures nor the huge amount of antipatterns. Instead, it is possible to deal with the most promising refactorings and defer the revision of the others.

4.1 Assumptions and Requirements

Our approach is applicable to all kinds of programming languages for which the corresponding static and dynamic analyses are available. There are, however, differences in how complicated it is to obtain runtime information and how many sophisticated tools there are for a certain technology. Thus, for practical reasons, we decide to tailor our approach to the widely used programming language Java.

Our approach relies on finding antipatterns with a static analysis and therefore is limited to what can be found this way. To be able to analyze even large code bases we are restricted to detection tools that are very fast. Characteristic for antipatterns found by those tools is that they are relatively simple and often concern only a single operation that is empirically known to be less efficient than some other operation, e.g. the + operator that concatenates strings. Significant savings are expected especially if antipatterns occur in loops or frequently called functions. Note that more complex causes for performance issues, like memory leaks, inefficient or unnecessarily large database requests or too frequent remote service calls, are hardly detectable by a static analyses in a fail-safe fashion. Expensive techniques, e.g. symbolic execution, would be required, but they still cover only a small part of the considered domain. Additionally, a high rate of false positives must be prevented because the acceptance of a tool and the confidence in its well-functioning would diminish rapidly.

Considering this, we expect our approach to work best for applications where the same source code is executed very often and thus the achievable performance improvement of refactoring antipatterns is high. This decisive criterion is assumed to hold for large business applications, e.g. server software or micro services that get thousands of similar request a second. Nevertheless, our approach works for other software as well, just with smaller performance gains.

Note that for the static antipattern detection, we require access to the source code, while for the dynamic analysis a runtime environment, and realistic input data must be available.

4.2 Rating Criteria

We aim at ordering the detected performance antipatterns according to their severity, i.e., their potential of improving performance. To achieve this, we determine some rating criteria. Those may be either static properties, e.g., the location in the source code where an antipattern is detected together with its loop depth, or properties that can be obtained through dynamic

analysis, e.g., the execution time and frequency of the surrounding method.

Static Properties

The core of our static analysis is the static antipattern detection, which provides a list of antipatterns together with their location in the code. As an additional static property we use the loop depth at the corresponding program location as a rating criterion, i.e. within how many layers of loops a specific antipattern is nested. We choose this property because source code within loops has the potential of being executed very often. If an antipattern represents an inefficiency, the many repeated executions give it a higher impact on performance and therefore it is more important to refactor.

Note that we do not use the actual amount of executions of a loop or the nesting depth of a given antipattern. To determine the actual amount of executions of a loop for arbitrary inputs is an undecidable problem, thus we cannot use this information as rating criterion. How deep an antipattern is nested in arbitrary control flow structures, i.e., the nesting depth, is easy to determine. One could argue that source code within, e.g., an `if`-statement is executed less often. However, we have no evidence that control flow structures other than loops form a reliable correlation that can be used as basis for a rating.

Runtime Properties

An important runtime property is the total execution time of a method. It is defined as the sum of all execution times of a method in a given program run. Therefore, it gives an impression on how much time the program spends in a specific method. The time spent in subroutines is counted towards the respective subroutine but not the calling method. Thereby, we get a correlation between the time spent and a very limited number of code lines. A high total execution time indicates that either some very expensive operations are performed or the number of executions must be high, e.g. due to a loop. In the second case an antipattern in this method has a higher impact on performance.

Another interesting property is the call count, i.e. how often a method is called during runtime. Using the same reasoning as above, we consider antipatterns in frequently called methods to have a higher impact on performance.

The third runtime property is the memory consumption of a method. To capture the memory consumption of a given method in Java, we use the suspension count. As Java is a memory-managed language, the garbage collector suspends the currently

executed method from time to time. The suspension count tells how often the garbage collector suspended a certain method to perform a collection. We choose this property as indicator for high memory consumption with the reasoning that if a method suffers suspensions disproportionately often, it probably allocates a lot of memory. The claimed correlation is based on the assumption that a garbage collection takes place whenever all memory is used up, which statistically happens more often in allocation intensive methods. Although different implementations of garbage collectors behave very differently in many ways, the assumption that more suspensions by the garbage collector indicate a higher memory consumption presumably holds. Note that for many other languages there exist profiling tools like Google's `gperftools` for C or the Memory Profiler for Python, which report the memory consumption of each method in a given program. Our rating function can easily be adapted to include these measures instead of the suspension count.

A runtime property that we leave out is the increase of execution time under increasing load. If a method takes significantly more time just because the system is under load, this indicates that the method contains some operation that impairs the performance. Often, the problem is about waiting time that is spent e.g. for synchronization between multiple threads (Grabner, 2009). We still do not consider this property for two reasons. First, none of our antipatterns causes waiting times. Second, measuring the increase of execution time under increasing load in a completely automated fashion is very complicated, e.g., because a dedicated testing environment for the measured software is required. For the same reasons we do not utilize synchronization and waiting times. Neither do we consider the API breakdown, because the information which component takes the most time is not detailed enough to form a connection with specific antipattern occurrences.

Antipattern Properties

As a further important rating criterion, we use the properties of the antipattern itself. We expect some antipatterns to bring high performance gains through refactoring while others yield only small improvements. To assess the general effectiveness of a given set of refactorings, we have implemented a micro benchmark for each class of antipattern and its refactored counterpart in a before-afterwards fashion (cf. Section 5). In doing so, we evaluate the effectiveness of each refactoring and thus can derive meaningful weights for our rating function.

Note that we have the choice to utilize either the

relative improvement after the refactoring or the absolute improvement. As we are mostly interested in a positive effect on the performance of the whole software it makes sense to consider the absolute gain. The relative improvement is only of limited meaning because an operation that takes quasi no time has few saving potential even if it can be made faster by a factor of 50. Therefore, we select the absolute effectiveness of refactorings as rating criterion.

4.3 Rating Function

Our final goal is to provide a heuristics for the prioritization and selection of antipatterns regarding their negative impact on performance for a given program. To achieve this, we present a novel rating function that can be used as a heuristics to estimate the severity of a detected antipattern in terms of the expected effect of refactoring the antipattern on performance. Our rating function is based on the various criteria discussed above and forms the heart of innovation in our approach as it actually combines the statically and dynamically obtained data.

We define our rating function, which determines the expected effectiveness of refactoring a given antipattern AP , as follows:

$$\text{severity} = \text{exec} \cdot (\text{calls} + b \cdot \text{loop}) \cdot f_{t,AP} + (\beta \cdot \text{susp} + 1) \cdot (\text{calls} + b \cdot \text{loop}) \cdot f_{m,AP}$$

where exec is the total execution time, susp the suspension count, calls the call count, loop the loop depth, $f_{t,AP}$ an antipattern time factor and $f_{m,AP}$ an antipattern memory factor. The antipattern time and memory factors $f_{t,AP}$ and $f_{m,AP}$ capture the general effectiveness of refactoring the antipattern AP . We have determined these factors using micro benchmarks. The idea behind this is as follows: If the execution time of a single piece of code only consisting of a given antipattern can be reduced by a factor of 10 using the proposed refactoring for this antipattern, we assess the general effectiveness of this refactorings to have a time factor of $f_{t,AP} = 10$. If, for the same experiment, the memory consumption is reduced to 50%, the memory factor of this refactoring is $f_{m,AP} = 2$. We describe our micro benchmarks to determine the time and memory factors for a given set of antipatterns in Section 5 and present the resulting factors in Table 2. To fine-tune our rating function, we introduce the weighting factors b and β , where b weights the relative relevance of the loop depth compared to the call count, and β the relative relevance of memory consumption to the execution time.

The ratio behind our rating function is to identify antipatterns that are at locations in the source code

that are executed very often. Refactorings at those locations have a larger potential of improving performance than elsewhere and should receive a higher rating. If, e.g., the execution time of a method is high, it is probable that this method is either called very often or contains a loop with many runs. If an antipattern is located in such a method with a high call count, we assume that it is executed often. Consequently, in this case the term $\text{exec} \cdot \text{calls}$ becomes large and leads to a higher rating. If on the other hand the call count is low but the antipattern lives within a loop, we likewise assume many executions. This time the term $\text{exec} \cdot \text{loop}$ becomes large and again leads to a higher rating. When merged together under the premise that either of the two cases should result in a higher rating, we get the term $\text{exec} \cdot (\text{calls} + b \cdot \text{loop})$ in the rating function. The weighting factor b can be used to normalize the loop depth with respect to the call count (the loop depth is typically between 0 and 4, while the call count has much larger numbers), and to express a domain- or application-specific relevance of loop depth and call count. In applications or domains where the loop depth is not expected to significantly influence the performance, b can be reduced, and vice versa. The antipattern time factor $f_{t,AP}$ gives an estimate for the general impact of the antipattern on execution time and is therefore multiplied. Altogether, this yields the first summand of the rating function.

The derivation of the second summand is very similar. For methods that get frequently suspended for garbage collection we assume a higher memory consumption. This leads to the term $\text{susp} \cdot (\text{calls} + b \cdot \text{loop})$. We again multiply with the antipattern memory factor $f_{m,AP}$ to take the general impact of the antipattern on memory consumption into account. A particularity of the suspension count is that for most methods it simply is zero, since overall garbage collection kicks in relatively seldom. Because we do not want to zero out the whole impact on memory, we add the constant one and get $(\text{susp} + 1)$.

Finally, we put together the two terms, each representing an independent indication of a high impact on performance. Since execution times can easily grow large while the suspension count keeps low, the weighting factor β should be used to align the magnitude. In addition, the software developer can increase β to search for refactorings that are promising to reduce the memory consumption, and decrease β to focus on refactorings that are promising to reduce the overall execution time. Note that in $(\beta \cdot \text{susp} + 1)$ the constant one is not scaled by β . The reasoning is that in case of zero suspensions the summand should have little influence and only break the tie between otherwise similarly rated antipatterns. If scaled, the second

summand could grow significantly large, although the suspension count is zero and there actually is no evidence for a high memory consumption.

4.4 Weight Determination

To normalize the loop depth with respect to the call count and the memory suspensions with respect to the execution time, we determine initial values of the weighting factors b and β . As mentioned above, they can be adjusted to fine-tune the ranking function to certain domains or to a desired performance goal.

The weighting factor b defines the relation between call count and loop depth. To scale the loop depth range of 0 to 4 such that it matches the magnitude of call counts common for the current antipattern selection process, we choose

$$b = \frac{1}{N_m} \sum_{methods} calls$$

with N_m as the number of methods. In other words, we choose b such that it equals the average call count of a method. In our experiments, the call count average is a multiple of the median. Thus, the loop depth has an adequate effect on the antipatterns rating but the extreme call counts still surpass the loop depth in effect. Note that b has to be calculated once in an antipattern selection process.

The weighting factor β defines the relation between execution time and suspension count:

$$\beta = \alpha \cdot \frac{\sum_{methods} exec}{\sum_{methods} susp}$$

In other words, β equals the total execution time divided by the total suspension count. The idea is to calculate the average of how much time corresponds to one suspension and scale the suspension count accordingly. The factor α can be used to reduce the suspension counts influence because it is suspected to be less reliable and accurate than the execution time, as the numbers generally are very low and a proper statistical distribution sets in very late. In our experiments, we use $\alpha = 0.2$. The weighting factor β has to be calculated once per antipattern selection process.

5 EVALUATION

We have implemented our approach in Java. We perform the static code analysis with PMD (Copeland and Le Vouch, 2017), which uses detection rules to find patterns in the source code. Compared to other tools like Checkstyle (Burn, 2017) and FindBugs

(Pugh and Hovemeyer, 2017), PMD has more rules for performance antipatterns and thus is best suited for our approach. For the recording of runtime properties, we use the monitoring tool Dynatrace AppMon (Dynatrace, 2017). It is widely used in practice and provides all the dynamic properties we need.

In this section, we first present our micro benchmarks and experimental evaluation of the general effectiveness of a given set of refactorings independent of a concrete program. Then, we demonstrate two experiments we have conducted in order to evaluate our approach for the automated selection of refactorings that are promising to have a high impact on the performance of a given program. As a case study, we use STATE, a SystemC to Timed Automata Transformation Engine written in Java and developed at TU Berlin (Herber et al., 2015; Herber et al., 2008). Although this is not the class of software our approach is designed for and thus the performance gain is small, the obtained findings demonstrate the practical applicability of our approach. Furthermore, we show some example output and give a first impression of the potential of our approach.

We carried out all experiments on an Intel(R) Core(TM) i7-2620M CPU @ 2.7 GHZ, 2 Core with 8 GB RAM running the Microsoft Windows 10 Pro operating system. We use the Oracle JVM version 8.

5.1 Micro Benchmarks

We have implemented micro benchmarks to determine the general effectiveness of a given set of antipatterns independent of a concrete program. We measure the effectiveness in terms of time and memory factors $f_{t,AP}$ and $f_{m,AP}$, which represent the relative severity of the various antipatterns. Since we are interested in the relative effectiveness of performance refactorings, only the relation between the performance of code containing the antipattern and code containing the refactored version is important and the absolute results do not matter.

Challenges of Java Micro Benchmarks

Micro benchmarks are not easy to design, especially in a language like Java. There are some general pitfalls and some that stem from how Java and its virtual machine work (Goetz, 2005).

The first thing to go wrong is that something completely different is measured than what was intended. A naive example is a benchmark to measure some arithmetical operation that writes each result to the console. What impacts the performance in such a setting is mostly the output and not the actual arithmetic

tics. To avoid this, we put only the absolutely necessary operations into the measurement code.

The accuracy of the CPU clock is by far not high enough to capture times in the magnitude of CPU cycles. Thus, we nest the operations we want to measure into a simple, repeating loop. The repetition count must be high enough to reach overall computation times where the accuracy is sufficiently good.

A Java specific pitfall is the just-in-time compiler (JIT compiler), which automatically compiles frequently executed portions of the program while leaving the rest for interpretation as usual. This can corrupt a measurement because half of the executions are interpreted and the other half compiled. We encounter this challenge with a so called warm-up. This means that we run the benchmark code 20000 times before the actual measurement is started. This guarantees that the JIT compiler translates the benchmark code and we measure only the compiled version.

Another general difficulty is the compiler optimization in benchmarks. As the executed code does not fulfill any contentual purpose the compiler may find out and optimize it away, rendering the whole benchmark useless. To solve this problem, we always return a number that in some way contains values involved in the measured code. Like this, we avoid optimization with a negligible overhead.

The garbage collection in Java is another mechanism we take into account with our micro benchmark design. If, for example, a garbage collection is performed during a benchmark the execution time increases significantly. Hence, before each measurement, we demand a garbage collection to happen to achieve similar starting conditions. Actually, the JVM cannot be forced to carry out a garbage collection but according to our experiments it always obeys. Thus, if a garbage collection takes place, it is because so much memory was consumed.

When taking a measurement, the result is subject to deviations. Therefore, we repeat the measurement several times. In a series there probably are outliers, e.g., due to some irregular background process on the machine that executes the benchmarks. For this reason we discard the extreme values and take the average over the remaining as final outcome.

Micro Benchmark Implementation

We have implemented the micro benchmarks as an Apache Tomcat servlet (Foundation, 2017). This allows a user friendly control in the web browser through a simple HTML interface. To cover all PMD performance rules, we have implemented 20 benchmark pairs, each consisting of one benchmark for the antipattern and one for its refactored version. The

core of each benchmark is a specific function that executes an antipattern or its refactored counterpart in a loop with a certain repetition count N , in our case $N = 100000$. Around this benchmark function the measurement process is built up. One benchmark measurement consists of two parts of which the first measures memory consumption and the second measures execution time.

For the memory consumption, we take the difference in heap size of the JVM before and after the benchmark function. Because in Java garbage collection can happen at any time, it has to be considered for the measurement, otherwise the alleged memory consumption even may become negative. To solve this problem, we use a callback function, which is triggered by each garbage collection run and which records how much space got cleared. We use this to calculate the memory consumption as follows:

$$memConsumption = (heapAfter + \sum_{GCRuns} collected) - heapBefore$$

We repeated this process 50 times before taking the average, which we consider the true memory consumption. First experiments showed that the callback function does not reliably execute timely before the measurement in which it was triggered is over. Therefore, after every measurement we schedule a wait of 100ms to catch late coming garbage collections and assign the numbers to the appropriate measurement.

For the execution time measurement, we take the difference in the system time before and after the benchmark function. To achieve a reliable value we calculate an average over 50 measurements, where we discard the lowest and highest four values beforehand. In order to accomplish an even more reliable result we calculate such an average for ten different cases, where in each case the repetition count is modified according to $repCount = i \cdot N$ with $i = 1, 2, \dots, 10$. In doing so, we get a series of supposedly equidistant execution time averages. We calculate the distance between every two successive results, which represents the increase in time for another N runs. Over these distances we take the average and finally consider it the true execution time of N runs of the benchmarked antipattern or its refactored counterpart. The deviation of the minimal distance and the maximal distance from the average indicate the quality of the measurement, with a low deviation confirming the outcome.

Results and Interpretation

The results from our micro benchmarks are shown in Table 2. Note that benchmarks marked with * were

executed with $N = 1000$ due to long execution times. The table comprises short, descriptive names for the benchmarks and the corresponding average times and the average memory consumptions as described above. The time factor describes by what factor the execution time changed in the refactored version compared to the antipattern, with a high value indicating that the refactoring is effective. The time saving describes the absolute gain in execution time achieved by the refactoring. Analogously, the values are calculated for memory consumption.

The time factors are in a range of 0.8 to 21497.92, i.e., some refactorings even have a negative impact while others are incredibly effective. The time savings are in a range of -0.16 to 4275.11 milliseconds per 100k repetitions and it is notable that a high factor does not necessarily appear together with a high absolute saving. Regarding memory consumption, the factor range is 0.21 to 10086.67 and the savings range is -0.09 to 198.17 MB per 100k repetitions. For those pairs where the refactored version consumes no memory the factor becomes NaN.

The results fit well with the expectations we had based on the antipattern description. For example, we now have evidence that performing arithmetics with the `short` type takes additional execution time due to the internal type casts but saves memory. In conclusion, we are very confident that our effort to design good benchmarks payed off by providing useful results that we can use in the rating function to distinguish between more or less severe antipatterns.

The measurements taken with the micro benchmarks are not only good for determining the antipattern factors used in the rating function. They have a value in themselves because the effectiveness of refactoring the antipatterns gets quantified in a relative and an absolute way. The results show that there are several refactorings that reduce execution time or memory consumption by large factors. At the same time, we get an overview of how much can be saved through refactoring this kind of antipatterns. While for some the savings are close to zero, for others they are multiple seconds per 100k calls. Those values suggest that in general our approach of refactoring this kind of statically detectable antipatterns has some effect, as long as not every occurrence is considered but only systematically selected ones. The results from our micro benchmarks also provide a valuable insight to the general effectiveness of performance refactorings and can be used by further research on antipatterns and performance refactorings.

Table 1: Experiments with STATE.

	Average Time	Absolute Diff	Relative Impact
Original	2232.47 ms	-	-
Injected	2242.03 ms	+9.56 ms	0.428 %
Refactored	2218.07 ms	-14.4 ms	0.645 %

5.2 Experimental Evaluation

We have evaluated our approach with the software STATE (Herber et al., 2015; Herber et al., 2008) in version STATE-2.1. It is licensed as open source under the GNU General Public License version 3 and consists of approx. 30,000 lines of code in 285 classes. We chose one of the shipped examples from STATE to do our measurements, namely `b.transport`.

Experiment 1: Antipattern Injection. In our first experiment, we have injected some antipatterns in the source code of STATE, measured their impact on performance and evaluated how they get rated by our ranking tool. To achieve this, we have duplicated the STATE source code. Then, in one copy we have manipulated two methods by replacing all occurrences of `StringBuilder` with the less efficient `+` operator. In this process, we have altered about 60 lines of code and replaced in total 49 calls to `append`. The rest of the source code remains unchanged.

Our expectation is that the manipulated copy runs slower, i.e. the measured execution times are increased. We base this expectation on the benchmark results where the string concatenation antipattern showed strong impact on performance. Another expectation is that the introduced antipatterns get ranked high in a follow up analysis of the manipulated copy, because they were injected into a prominent method and have large antipattern factors.

The upper two rows of Table 1 show the execution times of the original STATE version compared to the worsened version where we have injected antipatterns. The average execution time of the original STATE software is 2232.47 ms. The average execution time of the worsened version with antipatterns injected is 2242.03 ms, resulting in an absolute difference of 9.56 ms. Thus, the refactoring of the injected antipatterns, i.e. the restoration of the original state, achieves a performance improvement of 0.428%. The subsequent rating tool analysis reveals that the injected antipatterns are found by our tool. The 24 occurrences appear among the 26 top rated antipatterns.

According to our expectation, the STATE version with antipatterns injected shows worse performance

than the original. The execution time difference of about half a percent is small, but we have to keep in mind that STATE has very different characteristics to a large-scale server software or micro service, where our approach is supposed to exploit its full potential. Considering this, half a percent is already a good result, especially in relation to the very low effort it takes to implement some simple, local refactorings.

Experiment 2: Performance Refactorings. In our second experiment, we have performed a preliminary analysis of STATE with the rating tool and subsequently refactored the top rated antipatterns. Afterwards, we measured if the performance was improved by the refactorings. With our approach, we get a list of detected antipatterns sorted according to their assigned ratings. Figure 2 shows the first five lines of the output file slightly shortened. The large number in square brackets is the rating assigned to the antipattern. The other information helps to comprehend the rating and to find the antipattern in the source code.

In our experiment, we have implemented the proposed refactorings for 17 of the top rated 19 antipatterns. Two antipatterns remain untreated. One is an unavoidable object instantiation inside a loop and the other would require a `StringBuilder` to prepend text, which it is not intended for. Apart from the 17 refactorings the source code remains unchanged.

Our expectation is that the refactored version runs faster than the original, i.e. the measured execution times are reduced. Although the analyzed software is not in our target domain of large-scale server software, this experiment shows exactly how our approach is meant to be used in practice.

The last row of Table 1 shows the results for our second experiment. The average execution time of the refactored version of STATE is 2218.07 ms. Compared to the original version, this results in a difference of 14.40 ms. Thus, the refactoring of the 17 top rated antipatterns achieves an overall performance improvement of 0.645%.

Overall, we can see our expectation satisfied, since the refactored version effectively executes faster than the original STATE. Again, slightly more than half a percent is a small performance improvement but the same argumentation as above holds and we still consider the result a success. It shows that the rating tool succeeds in proposing refactorings that improve performance and suggests that its application on a server software or micro service can yield great performance gains with a small refactoring effort. Note that a PMD analysis of the STATE source code already restricted to the performance antipatterns yields 843 issues. Of those, only 339 received

a rating greater than zero and promise a positive effect on performance through refactoring at all. Note also that using solely Dynatrace AppMon to capture runtime properties leaves one with lots of information without any concrete instruction on what to do.

6 CONCLUSION

In this paper, we have proposed a novel approach to combine static and dynamic software analyses to automatically select refactorings that improve the performance of a given program. Our major contributions are twofold: First, we have presented a rating function for antipatterns, which assesses their respective potential to improve performance through refactoring based on both static and dynamic properties. Second, we have implemented micro benchmarks that assess the general effectiveness of a given set of performance antipatterns independent of a specific program. Our benchmarks clearly show that the antipatterns actually have an effect on performance, although the effects vary. Due to the mostly small savings, in the majority of cases a refactoring is only reasonable if the antipattern is executed frequently, e.g. in a loop or frequently called method. This illustrates the importance of a feasible approach to select the most effective refactorings in a given program.

We have implemented our approach for the automated selection of refactorings that are most promising to improve the performance of a given program using PMD (Copeland and Le Vourch, 2017) for static code analyses and Dynatrace AppMon (Dynatrace, 2017) for dynamic software analysis to capture performance measures. The result is a list of recommended refactorings ordered by effectiveness.

We have demonstrated the practical applicability of our approach with a sample software that consists of 30,000 lines of code. Although our approach works best for large scale server software, it still yields some improvement for our much smaller case study from a totally different domain. We therefore assess the potential in a large scale server software as high, especially due to the good cost-benefit ratio.

Our approach enables us to select only the most important antipatterns out of the huge amount of antipatterns that are typically provided by static antipattern detection tools. At the same time, it drastically reduces the cost of interpreting data delivered by dynamic analyses. Due to the automated interpretation and the precisely recommended refactorings, little expertise is required.

In future work, we plan to carry out a field experiment in which our approach is used to improve the

[3280529]	exec= 97.00 susp= 0 calls=13595 loop=0	Antipattern 'AppendCharacterWithChar' in method 'toString' at line 288 in file Location.java
[2592057]	exec= 1674.51 susp= 3 calls= 5 loop=1	Antipattern 'AvoidInstantiatingObjectsInLoops' in method 'parseParallel' at line 123 in file UppaalXMLManager.java
[1548647]	exec= 1504.23 susp= 0 calls= 5 loop=1	Antipattern 'AppendCharacterWithChar' in method 'embed' at line 103 in file ParallelUppaalXMLEmbedder.java
[1280611]	exec= 58.58 susp= 0 calls= 8770 loop=0	Antipattern 'AppendCharacterWithChar' in method 'toString' at line 221 in file Transition.java
[1280611]	exec= 58.58 susp= 0 calls= 8770 loop=0	Antipattern 'AppendCharacterWithChar' in method 'toString' at line 222 in file Transition.java

Figure 2: Extract of Rating Tool Results for STATE.

Table 2: Micro Benchmarks for Antipatterns.

Micro Benchmark	Avg Time [ms]	Avg Mem [kB]	Time Factor [1]	Time Saving [ms/100k]	Mem Factor [1]	Mem Saving [MB/100k]
StringBuilder using equals("")	1.30	4133				
StringBuilder using length() == 0	0.04	140	30.62	1.24	24.52	3.96
Concatenate 10 strings with plus operator	74.42	246622				
Concatenate 10 strings with StringBuilder	37.21	48900	2.09	40.70	5.06	198.17
Multiple append in multiple statements	9.22	60202				
Multiple append in only one statement	9.28	60215	0.99	-0.05	1.00	-0.09
Instantiate Boolean object	0.05	140				
Reference pooled Boolean	0.04	0	1.25	0.01	NaN	0.14
Arithmetics with short	0.05	17				
Arithmetics with integer	0.04	82	1.25	0.01	0.21	-0.07
Instantiate object with final member *	23.45	553				
Instantiate object with static final member *	23.18	544	1.01	22.95	1.02	0.01
Call toArray with empty array	0.86	6471				
Call toArray with correctly sized array	1.02	6471	0.84	-0.16	1.00	0.00
Create many small objects	0.44	2740				
Create separate data arrays	0.21	802	2.10	0.23	3.42	1.94
Check first char with startsWith	0.04	0				
Check first char with charAt(0)	0.05	0	0.80	-0.01	NaN	0.00
Copy array iteratively into List *	50.30	40539				
Wrap array with asList	0.39	2623	10962.82	4275.11	1313.69	37.92
Copy array iteratively into array	10.02	15				
Copy array with copyarray	9.86	14	1.02	0.16	1.07	0.00
Convert to string with + ""	4.12	12732				
Convert to string with toString	3.59	5510	1.15	0.53	2.31	7.22
Instantiate with explicitly initialized member *	27.47	570				
Instantiate with implicitly initialized member *	25.32	565	1.08	182.75	1.01	0.01
Throw an exception *	30.35	1068				
Set flag and check if it's set	0.12	9	21497.92	2579.63	10086.67	1.06
Create string with new	0.57	1876				
Create pooled string with quotes	0.04	0	14.25	0.53	NaN	1.88
Check string equality casting both upper case	12.70	20872				
Check string equality ignoring case	4.60	0	2.76	8.10	NaN	20.87
Append character with double quotes	4.17	2366				
Append character with single quotes	1.69	1629	2.47	2.48	1.45	0.74
Search character with double quotes	0.93	0				
Search character with single quotes	0.91	0	1.02	0.02	NaN	0.00
Check if string is empty with trim	1.00	2408				
Check is string is empty with loop	0.04	0	25.00	0.96	NaN	2.41
Initialize StringBuilder too short	4.46	33252				
Initialize StringBuilder sufficiently large	3.82	28163	1.17	0.64	1.18	5.09

performance of a large scale server software. Furthermore, we plan to investigate more complex refactorings. For this, we plan to include static analyses

that detect performance bottlenecks, for example, via symbolic execution.

REFERENCES

- Arcelli, D., Berardinelli, L., and Trubiani, C. (2015). Performance antipattern detection through fuml model library. In *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development*, pages 23–28. ACM.
- Arcelli, D., Cortellessa, V., and Trubiani, C. (2012). Antipattern-based model refactoring for software performance improvement. In *Quality of Software Architectures*, pages 33–42. ACM.
- Bernardi, M. L., Cimitile, M., and Di Lucca, G. A. (2013). A model-driven graph-matching approach for design pattern detection. In *Working Conf. on Reverse Engineering*, pages 172–181. IEEE.
- Burn, O. (2017). Checkstyle. <http://checkstyle.sourceforge.net/index.html>.
- Copeland, T. and Le Vourch, X. (2017). PMD. <https://pmd.github.io/>.
- Cortellessa, V., Martens, A., Reussner, R., and Trubiani, C. (2010). A process to effectively identify guilty performance antipatterns. In *International Conference on Fundamental Approaches to Software Engineering*, pages 368–382. Springer.
- Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.-T., and Jalby, W. (2005). Exploring application performance: a new tool for a static/dynamic approach. In *Proceedings of the 6th LACSI Symposium*. Los Alamos Computer Science Institute.
- Dynatrace (2017). Dynatrace appmon. <https://www.dynatrace.com/>.
- Fontana, F. A. and Zanoni, M. (2017). Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, 128:43–58.
- Foundation, A. S. (2017). Apache tomcat. <http://tomcat.apache.org/>.
- Fowler, M. and Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Goetz, B. (2005). Anatomy of a flawed microbenchmark. <https://www.ibm.com/developerworks/java/library/j-jtp02225/>.
- Grabner, A. (2009). Performance analysis: How to identify synchronization issues under load? <https://www.dynatrace.com/blog/performance-analysis-how-to-identify-synchronization-issues-under-load/>.
- Herber, P., Fellmuth, J., and Glesner, S. (2008). Model Checking SystemC Designs Using Timed Automata. In *International Conference on Hardware/Software Codesign and Integrated System Synthesis (CODES+ISSS)*, pages 131–136. ACM press.
- Herber, P., Pockrandt, M., and Glesner, S. (2015). STATE – A SystemC to Timed Automata Transformation Engine. In *International Conferen on Embedded Software and Systems (ICESSE)*, pages 1074–1077. IEEE.
- Heuzeroth, D., Holl, T., Hogstrom, G., and Lowe, W. (2003). Automatic design pattern detection. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 94–103. IEEE.
- Long, J. (2001). Software reuse antipatterns. *ACM SIGSOFT Software Engineering Notes*, 26(4):68–76.
- Louridas, P. (2006). Static code analysis. *IEEE Software*, 23(4):58–61.
- Luo, Q., Nair, A., Grechanik, M., and Poshvanyk, D. (2017). Forepost: Finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering*, 22(1):6–56.
- Owen, K. (2016). Improve the smell of your code with microrefactorings. <https://www.sitepoint.com/improve-the-smell-of-your-code-with-microrefactorings/>.
- Pugh, B. and Hovemeyer, D. (2017). Findbugs. <http://findbugs.sourceforge.net/>.
- Rasool, G. and Streitfdert, D. (2011). A survey on design pattern recovery techniques. *IJCSI International Journal of Computer Science Issues*, 8(2):251–260.
- Smith, C. U. and Williams, L. G. (2002). New software performance antipatterns: More ways to shoot yourself in the foot. In *Int. CMG Conf.*, pages 667–674.
- Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halikidis, S. T. (2006). Design pattern detection using similarity scoring. *IEEE transactions on software engineering*, 32(11).
- Washizaki, H., Fukaya, K., Kubo, A., and Fukazawa, Y. (2009). Detecting design patterns using source code of before applying design patterns. In *Intl. Conf. on Computer and Information Science*, pages 933–938. IEEE.
- Wendehals, L. (2003). Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, pages 29–32.
- Wierda, A., Dortmans, E., and Somers, L. J. (2007). Detecting patterns in object-oriented source code - a case study. In *ICSOFT (SE)*, pages 13–24. INSTICC Press.
- Woodside, M., Franks, G., and Petriu, D. C. (2007). The future of software performance engineering. In *Future of Software Engineering, 2007. FOSE'07*, pages 171–187. IEEE.