On the Validity of Performance Antipatterns at Code Level

David Georg Reichelt
Universität Leipzig
dg.reichelt@uni-leipzig.de

Stefan Kühne
Universität Leipzig
kuehne@uni-leipzig.de

Wilhelm Hasselbring
Universität Kiel
hasselbring@email.uni-kiel.de

Abstract

Performance antipatterns at code level should be avoided to assure good application performance. Performance antipatterns avoidance is hard, since it requires up-to-date knowledge of these antipatterns. Common lists of antipatterns, like the performance rules of the static code checker PMD, only contain limited information about versions and circumstances where the performance antipatterns are valid.

We close this gap by prodiving a suite of 30 performance benchmarks. Each of this benchmarks checks whether the performance antipattern is measurable in Java 6, 7, 8, 11 and 12. We find that two of the 30 performance checks are not valid in the current OpenJDK 12.

1 Introduction

Performance antipatterns, i.e. patterns of non-optimal implementations at code level and their respective solutions, are frequently used in software development. By avoiding antipatterns and applying the solutions of the antipatterns, developers do not need to measure implementation alternatives. A prerequisite for this usage is that antipatterns are valid in each context. Therefore, a valid antipatterns non-optimal implementation can not be better than the antipatterns solution in any use case.

Unfortunately, the validity of antipatterns changes between Java versions: In versions prior to OpenJDK 6u21, converting a `List` to an array using `toArray(T[] a)` was faster when the array `a` had the size of the `List`. Starting with OpenJDK 6u21, due to internal optimizations inside of OpenJDK, calling `toArray` with an array `a` of length zero became faster. The `toArray`-change is well-known and is already widely discussed.[1]

Other performance antipatterns in Java have been less researched. This is a problem, since they may also change between versions. If performance antipattern are outdated, the developers will change their code without positive effect on performance. Moreover, if the non-optimal solution of an antipattern became optimal in some or all cases, like in the `toArray`-change, avoiding the antipattern reduces the overall performance.

The tool PMD contains 30 performance checks.[2] Only two of them contain version information: The discussed OptimizableToArrayCall and AvoidFileStream, which was introduced due to updates in the java.nio-API in Java 7.

In this paper, we present (1) a benchmark suite capable of validating PMDs performance antipatterns at code level and (2) the results of using this benchmark suite with current Java versions.

The remainder of this paper presents the benchmarks (Section 2), the measurement setup and results (Section 3) and related work (Section 4). Finally, a summary is given (Section 5).

2 Benchmarks

Our goal is to measure whether the performance of an antipattern's occurrence and the antipattern's solution occurrence differs. Therefore, we define *one* workload, which is executed once using the antipattern (method name `testBad`) and once without it (method name `testGood`). Our benchmarks hence do not test the full gamut of usages of the antipatterns.

We solely consider time consumption. Our benchmarks may find that (1) `testGood` is faster than `testBad`, (2) `testGood` and `testBad` are equally fast or (3) `testGood` is slower than `testBad`.

If we find that `testGood` is faster than `testBad` (3), the antipattern is invalid: Since there is *one* usage of the pattern which has good performance, it cannot be assumed in general that the pattern should be avoided. If we find that `testGood` is faster (1) than or equally fast (2) as `testBad`, we cannot make a definitive conclusion about the antipattern: There might be workloads which disprove the antipattern.

For our benchmarks, we need to define the measurement and the analysis of the measured values.

Measurement The benchmarks are implemented using the Java microbenchmarking framework JMH.[3] It automates the repetition of JVM starts and workload executions, which are necessary to obtain statistically reliable mearement results [2]. Listing 1 shows an example of an benchmark: The benchmark measuring the `AddEmptyString`-check. In the antipattern version, an `int` is converted to a `String` by adding

[1] https://shipilev.net/blog/2016/arrays-wisdom-ancients/

[2] https://pmd.github.io/latest/pmd_rules_java_performance.html

[3] Java Measurement Harness, https://openjdk.java.net/projects/code-tools/jmh/

an empty `String` to it. In the solution version, the conversion is done using `Integer.toString`.

Listing 1: Example Benchmark

```
@State(Scope.Benchmark)
public class AddEmptyStringBenchmark {
 [..] Random random=new Random();
 private int value = random.nextInt();
 @Benchmark
 public String testGood() {
  String s = Integer.toString(value);
  return s;
 }
 @Benchmark
 public String testBad() {
  String s = "" + value;
  return s;
 }
}
```

Analysis After the execution of the benchmarks, JMH outputs are saved as text and parsed. To derive whether there is a difference between two sets of JVM executions, we use the two-sided t-test, which is efficient for finding performance changes [9].

Our benchmarks are available in our repository.[4]

3 Results

We measured on 4 machines with i7-4770 CPU @ 3.40GHz, 16 GB RAM and Ubuntu 18.04 whether performance antipatterns persisted between OpenJDK 6.0_41, 7.0_201, 8.0_222, 11.0.4 and 12.0.2. We omitted Java 9 and 10, since these intermediary versions were supported less than a year and are not supported anymore. We used 99% confidence level for the two-sided t-test, 30 JVM forks and 5 (warmup) iterations á 10 seconds for every benchmark. To use repeatably the same execution environment, Docker containers are defined for every researched Java version. Since packaged versions of OpenJDK 6 and 7 are not maintained anymore, we use Ubuntu 12.04 and Ubuntu 14.04 for providing these OpenJDK versions.

Table 1 shows which performance antipatterns can be reproduced in different OpenJDK versions.[5] Results in parenthesis mark small differences. While they are considered a significant difference by t-test, the relative difference between the measurements is below 2%. Therefore, practical impact of using or removing the antipatterns may be low. Hyphen marks no difference by t-test.

Our results show that several optimizations have been or are still wrong, i.e. blindly applying them could slowdown a software. In the following, we will describe the two examples where antipatterns are more than 2% faster in latest OpenJDK 13: `ConsecutiveLiteralAppendsBenchmark` and `UseArrayListInsteadOfVector`.

Benchmark	6	7	8	11	12	13
AddEmptyString	×	×	×	×	√	√
AppendCharacterWithChar	√	√	√	√	(×)	√
AvoidFileStream		√	√	√	(×)	(√)
BigIntegerInstantiation	(√)	√	√	√	√	√
ConsecutiveLiteralAppends	√	√	√	×	×	×
OptimizableToArrayCall	×	√	√	√	√	√
RedundantFieldInitializer	(√)	(×)	(×)	(√)	(√)	(√)
SimplifyStartsWith	√	(√)	(√)	(×)	(√)	(√)
StringInstantiation	(√)	(√)	(√)	(√)	(×)	-
StringToString	√	√	(×)	(√)	(√)	-
TooFewBranchesForASwitch	(√)	(√)	(×)	(×)	(√)	(√)
UseArrayListInsteadOfVector	√	√	×	×	×	×
UseIndexOfChar	×	√	√	√	√	√

Table 1: List of not Fully Reproducibly Antipatterns in Different OpenJDK versions.

`ConsecutiveLiteralAppends`: The rule suggests to append `String` entities to an `StringBuilder` using one call to `append`, instead of using multiple calls for constructing the same `String`. For example, if the `String "Hello World"` should be added, this should be done by `.append("Hello World")` instead of `.append("Hello").append(" ").append("World")`. This example is also used in our benchmark.

In OpenJDK 11 and later, `byte` arrays are used to save data of a `StringBuffer`. When using only one `append` call, the `System.arraycopy` on the `byte` array is executed by calling the runtime stub `jbyte_disjoint_arraycopy`, which makes use of available CPU features for speeding up copying. When using multiple append calls, the used `String`s in our example are so small that they are moved directly by `mov`-calls. In contrary, for copying the larger `byte` array the runtime stub is used. In theory, the usage of the runtime stub should speed up the process; using our concrete parameters, it slows down the process. On a lower level, this can be reproduced by copying bytes: Copying eleven `bytes` (`"Hello World"`) from one array to another is slower than copying five `bytes` (`"Hello"`), than one `byte` (`" "`) and than five `bytes` (`"World"`) to an 16 `byte` sized array.

In the majority of cases, using one `append` is faster. Nevertheless, if the performance of `append` is crucial, its performance should be measured instead of blindly applying the antipatterns solution.

`UseArrayListInsteadOfVector`: The rule suggests to use `ArrayList` instead of `Vector`, since `Vector` is `synchronized` and therefore slower. If elements are added to `ArrayList` or `Vector` and the underlying array is too small, the array is grown using `System.arraycopy`. By factorial experiments, we found that if a class `ClassA` has a field with array type which is copied by `System.arraycopy`, the use of multiple instances of `ClassA` may result in performance degradation. This hence is no problem in `ArrayList` but an optimization problem in the JVM. Neverthe-

[4] https://github.com/DaGeRe/pmd-check

[5] All measurement results are available in *https://zenodo.org/record/3364562#.XU2LMvzRY5k*

less, replacing `Vector` by `ArrayList` may result in a performance regression if this effect is triggered.

Future implementations may change which implementation has the best performance again. Nevertheless, if an application should yield optimal performance in current OpenJDK, `ArrayList` should only be used instead of `Vector` after comparing the real world execution. A rule of thumb could be that elements are less often read than added.

4 Related Work

The definition of performance antipatterns have been researched in the context of (1) empirical found antipatterns, and (2) systematic analysis of (2a) software code and (2b) documentation repositories.

The work of Smith and Williams [1] provide an empirical collection of performance antipatterns. They focus on architectural level and are applicable to any programming language and paradigm, but illustrated by J2EE examples. Occurences of the problems may be found by systematic experiments [4]. Various blogs[6] provide Java antipatterns. Both, their validity and their up-to-dateness, are not researched.

Systematic analysis of software code repositories analyzes the code itself or executes measurement by analysis of the code. Analyzes of the code itself currently defines special antipattern types, e.g. concurrency issues [5, 6]. If measurements are executed, the workload needs to be defined. It is possible to use existing benchmarks [7], generate benchmarks [5] or transform existing unit tests [9].

Documentation analysis uses bug tracker and commit comments for defining performance antipattern. There are language-specific [8] and language-agnostic [3] works. After defining antipatterns, the works fix occurences of the bugs which were unknown before.

Both, work analyzing code repositories and work analyzing documentation repositories, define classes problems, which are mostly antipatterns. Some of them are language specific. Changes, which appear due to language updates, were not researched so far.

5 Summary and Future Work

We described a benchmark suite capable of evaluating the validity of performance antipatterns in different Java versions. We found that, among the 30 antipatterns defined in PMD, two are not valid in the newest OpenJDK version 12. To avoid wrong antipattern usage, we believe that a continous approach for validating performance antipatterns should be established. This could be done by using and extending our benchmark suite. It could further be used to benchmark the same antipatterns with other JVM implementations, e.g. Oracles HotSpot JVM.

Furthermore, developers should not rely solely on performance antipatterns. Instead, measuring the performance in every version using continous tests and evaluating the performance at code level, e.g. using PeASS [10], would facilitate using performance-optimal implementations.

[6] E.g. https://stackify.com/java-performance-tuning/

Acknowledgements

This work was funded by the German Federal Ministry of Education and Research within a PhD scholarship of Hanns Seidel Foundation. Computations for this work were done with resources of Leipzig University Computing Centre.

References

[1] C. U. Smith and L. G. Williams. "More new software performance antipatterns: Even more ways to shoot yourself in the foot". In: *CMG Conference*. Citeseer. 2003, pp. 717–725.

[2] A. Georges, D. Buytaert, and L. Eeckhout. "Statistically rigorous java performance evaluation". In: *ACM SIGPLAN Notices* 42.10 (2007).

[3] A. Nistor, T. Jiang, and L. Tan. "Discovering, reporting, and fixing performance bugs". In: *MSR 2013*. IEEE Press. 2013, pp. 237–246.

[4] A. Wert, J. Happe, and L. Happe. "Supporting swift reaction: Automatically uncovering performance problems by systematic experiments". In: *Proceedings of the 2013 ICSE*. IEEE Press. 2013, pp. 552–561.

[5] M. Pradel, M. Huggler, and T. R. Gross. "Performance regression testing of concurrent classes". In: *Proceedings of the 2014 ISSTA*. ACM. 2014.

[6] R. Gu et al. "What change history tells us about thread synchronization". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 426–438.

[7] J. P. Sandoval Alcocer, A. Bergel, and M. T. Valente. "Learning from Source Code History to Identify Performance Failures". In: *Proceedings of the 7th ACM/SPEC on ICPE*. ICPE '16. Delft, The Netherlands: ACM, 2016, pp. 37–48.

[8] M. Selakovic and M. Pradel. "Performance issues and optimizations in javascript: an empirical study". In: *Proceedings of the 38th ICSE*. ACM. 2016, pp. 61–72.

[9] D. G. Reichelt and S. Kühne. "How to Detect Performance Changes in Software History: Performance Analysis of Software System Versions". In: *Companion of the ACM/SPEC ICPE*. ACM, 2018, pp. 183–188.

[10] D. G. Reichelt, S. Kühne, and W. Hasselbring. "PeASS: A Tool for Identifying Performance Changes at Code Level". In: *Proceedings of the 33rd ACM/IEEE ASE*. (in press). ACM. 2019.