POLITECNICO DI TORINO
Repository ISTITUZIONALE

Quantitative Assessment of the Impact of Automatic Static Analysis Issues on Time Efficiency

Original
Quantitative Assessment of the Impact of Automatic Static Analysis Issues on Time Efficiency / Vetro', Antonio; Torchiano, Marco; Morisio, Maurizio. - ELETTRONICO. - (2011), pp. 1-8. (Intervento presentato al convegno Informatica Quantitativa 2011 tenutosi a Lipari, Isole Eolie (IT) nel 27-29 giugno 2011).

Availability:
This version is available at: 11583/2437375 since:

Publisher:

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

12 October 2024

Quantitative Assessment of the Impact of Automatic Static Analysis Issues on Time Efficiency

Antonio Vetro', Marco Torchiano and Maurizio Morisio
Dipartimento di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi, 24 - 10129 TORINO - ITALY
Email: name.surname@polito.it

***Abstract*—Background: Automatic Static Analysis (ASA) tools analyze source code and look for code patterns (aka smells) that might cause defective behavior or might degrade other dimensions of software quality, e.g. efficiency. There are many potentially negative code patterns, and ASA tools typically report a huge list of them even in small programs. Moreover, so far, little evidence is available about the negative impact on performance of code patterns identified by such tools. A consequence is that programmers cannot appreciate the benefits of ASA tools and tend not to include them in their workflow.**

Aims: Quantitatively assess the impact of issues signaled by ASA tools on time efficiency.

Method: We select 20 issues and for each of them we set up two source code fragments: one containing the issue and the corresponding refactored version, functionally identical but without the issue. We set up three different platforms, isolated from network and other user programs, then we execute the code fragments, and measure the execution time of both code versions.

Results: We find that eleven issues have an actual negative impact on performance. We also compute for each issue an estimation for the delay provoked by a single execution.

Conclusions: We produce a set of issues with a verified negative impact on performance. They can be checked easily with an analysis tool and code can be refactored to obtain a provably more efficient code. We also provide the estimated delay cost of each issue in the environments where we conduct the tests. These results can be improved with the help of other researchers: repeating the tests in several platforms would make it possible to build up a wider benchmark.

I. Introduction

Automatic static analysis tools analyze source code looking for violations of bug patterns that might cause defective behavior. However, finding defects is not the unique scope of these tools: developers can use them to verify code compliance to standards or to assess the internal quality of the developed system. In fact, ASA tools are also able to look for patterns of bad programming practices and suggest refactoring [1] tips. A positive side effect is that programmers are encouraged to improve specific software internal characteristics of their product. A drawback instead is the high number of warnings (issues) signaled to the users, among which many false alarms can be present: as a consequence, users could be discouraged to review all the warnings not to lose too much time, and managers give up the introduction of static analysis tools in the development process. However, this flip side can be limited through customization: ASA tools categorize and prioritize their warnings to help programmers in the customization and enable only those issues related to the quality dimension(s) of interest. Moreover, the research community is increasing its interest in assessing the value of static analysis, but this effort is addressed mainly in the prioritization of issues with respect to their relationship to defects in the code. Examples are the researches conducted by Kim and Ernst on open source software and defects found with static analysis ([2] and e.g. [3]), the studies of Wagner et al. on the type of bugs detectable by ASA tools (e.g. [4] and [5]) , whilst Ayewah and Pugh ([6]) and [7]) used FindBugs on production software to catch defects. We also conducted a similar study in university context [8] to discover which FindBugs issues were related to defects introduced by students in Java programs. The above studies represent research focused on the impact of ASA issues on defectiveness, conversely small effort is dedicated to other aspects of code quality, e.g. efficiency or maintainability, referring to the framework defined by the ISO/IEC 9126 product quality model [9]. The standard identifies 6 quality characteristics: Functionality, Reliability, Usability, Efficiency, Maintainability, Portability. This work is focused on Efficiency and in the use of ASA tools to improve it in Java programs.

II. Background and Related Work

Efficiency, in terms of time behavior, is measured by computation or approximation of the execution time of the software function under study. A fundamental concept in this computation is that the execution time is not deterministic, but it has a certain variation depending on the input data or on different contexts of the platform in which it is executed. For this reason, for a given code there is a best-case execution time (BCET), i.e. the shortest possible execution time, and a worst-case execution time (WCET), i.e. the longest possible execution time. Whilhelm et al. [10] identified in literature and industrial practices two approaches to determine the BCET and the WCET. The first approach is characterized by static methods: the code and the possible paths are analyzed and, combining different techniques, upper and lower bounds for

the execution time are provided. This methodology does not take into account the hardware and the environment on which the code is executed, hence the bounds overestimate the WCET and underestimate the BCET. The second methodology is measurement-based: the code, or a portion of it, is executed on a given hardware or a simulator for a set of inputs. Then, WCET and BCET are obtained from observation: these methods provide estimates and not bounds, and they usually underestimate the WCET and overestimate the BCET.

Despite the high number of techniques developed for both approaches, the problem for WCET analysis in the field of Java applications has not been deeply examined yet: Harmon and Klefstad conducted a survey of WCET analysis for Real-Time Java [11], but they were able to find fewer than twenty publications addressing the problem. Source code annotations as instructions to WCET tools ([12], [13]), low-level and high level analysis of bytecode ([14] [15]) and Java-native processors ([16]) are the most common solutions proposed, however it is very difficult to find a methodology to obtain precise and generalizable bounds for the WCET and the BCET in Java: the main motivation is the overhead and the variability introduced at execution time by VM services (e.g. automatic memory management) [17]. However, since the object of our experiment is very simple code, makes measurement simpler for us. For instance, we test code with only one possible path, thus we can exclude all static measurement-based approaches. Moreover, we are interested in the comparison of execution times: for this reason, we think it is considerable to abandon the usual concepts of WCET and BCET and adopt average values and confidence intervals. We focused on a specific ASA tool that is FindBugs v1.3.9 [18]. FindBugs uses analyzers called Bug Detectors to search for simple bug patterns. These bug detectors contain numerous heuristics to filter out or deprioritize warnings that may be inaccurate, or that may not represent serious problems in practice. FindBugs warnings are organized in 369 issues or bug patterns [1], grouped subsequently into Categories (Correctness, Performance, Security, etc) and priorities (high, medium, or low), based to the severity of the problem detected. Both categories and priorities are assigned by tool's authors, based on their wisdom and experience reviewing warnings in industrial and university contexts. A subset of FindBugs patterns is part of category Performance: they are supposed by tools' authors to have negative impact on Performance, i.e. the efficiency of the code.

III. Goal Definition

Let us consider a simple code fragment:

```
Collection<Integer> col =
                new LinkedList<Integer>();
...
col.removeAll(col);
```

If FindBugs were run on the above code it would signal the issue DMI USING REMOVEALL TO CLEAR COLLECTION. The description of the issues provided by the tool is: "*If you want to remove all elements from a collection* `c`*, use* `c.clear()`*, not* `c.removeAll(c)`". In response to such a notification from the tool the developer should refactor the code according to what is suggested by the issue description. i.e. replacing `col.removeAll(col);` with `col.clear();`.

[1]The full list of patterns is available at http://findbugs.sourceforge.net/bugDescriptions.html

TABLE I
Experiment Goal Definition

Evaluate	a subset of FindBugs issues
for the purpose	of assessing the actual impact compared to their removal by refactoring
with respect to	the time efficiency of code
from the viewpoint	of Java programmers
in the context of	archetypal ad-hoc code.

The research question we would like to answer with this study is whether the issue represents an actual threat to time efficiency. In other words: **does refactoring out the issue yields a code that exhibits an improved time efficiency?**

We define our experiment following the GQM template [19] for goal definition.

Object of study. The object of the study is represented by the issues signaled by FindBugs.

Purpose. The purpose of the experiment is to identify those issues that impact time efficiency and quantitatively assess the delay introduced. Having experimental evidence of this impact, programmers can be sure that if they refactor the code deleting these issues, the system will be faster.

Perspective. The perspective is from the point of view of programmers of Java applications that take care of performance issues in delivering their software.

Quality focus. Efficiency is the quality characteristic that we address. It is defined in ISO-IEC 9126 [9] as "the set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions". Efficiency can be specified both in terms of time and resource behavior: we focus our experiment on time behavior, i.e. the amount of time to perform one or more operations.

Context. The context is artificially developed code. We conduct our tests on code developed ad hoc to violate the issues that might impact negatively the efficiency-time behavior of the code.

Summarizing, our goal is defined by the scheme in Table I.

IV. Experiment Planning

A. Context and Variable Selection

Although one of the categories defined by the FindBugs tool is named Performance, we think that also issues belonging to other categories could actually affect the time efficiency

of code. For this reason, we select from FindBugs site a subsection of issues that might have a negative impact on the time efficiency with respect to the following set of criteria.

A Issue belongs to category performance

B Issue has a negative impact on performance with respect to expert judgment. The selection is made by the authors of this paper: two of them are professors of Java Programming course at Politecnico di Torino since more than ten years, whilst the first author is a second year PhD Student assisting the professors in the Java Course since four years. The experts read the description of the issues and for each of them classified them into one of the following categories (and implicitly assigned the relative score):

a) the issue impacts negatively the time efficiency of code (score: +1)
b) the issue does not impact negatively the time efficiency of code (score: -1)
c) no decision (score: 0)

Each issues is assigned a score that is the sum of the scores corresponding to the categories selected by the experts. Then an issue is selected for the experimentation when the total score is ≥ 2.

C Refactoring does not change functionality. For instance, the issue DLS OVERWRITTEN INCREMENT looks for code that performs an increment operation and then immediately overwrites it (e.g., `i++;` added in a for loop to skip an iteration). A possible code refactoring action would be to delete the offending increment: however, this action could change the functional behavior of the code, hence the issue is not selected.

D Efficiency does not depend on local (e.g. network) factors. For example, the issue DMI BLOCKING METHODS ON URL has a negative impact on performance because the $equals$ and $hashCode$ method of URL perform domain name resolution, thus this can result in a performance hit. However, this case is out of our interest because the cause of delay is the network and not the code. For the same reason we do not include in the experimentation DMI COLLECTION OF URLS .

E Identification of one issue per equivalence classes. The aim is to pick only one issue from each set of similar issues. In fact some issues are redundant, or one is a generalization of many others. For instance, consider the issue BC IMPOSSIBLE INSTANCEOF: it is signaled when the $instanceof$ operator will always return $false$, hence this is a useless operation that might lead to a delay. A similar issue is BC VACUOUS INSTANCEOF, that is complementary to the previous one, because it is signaled when $instanceof$ test will always return $true$. Therefore latter issue is representative also for the former one.

TABLE II
ISSUES SELECTED AS OBJECTS OF THE EXPERIMENT.

Code	Issue	A
1	BC VACUOUS INSTANCEOF	
2	BX BOXING IMMEDIATELY UNBOXED TO PERFORM COERCION	X
3	DLS DEAD LOCAL STORE	
4	DM BOOLEAN CTOR	X
5	DM NEW FOR GETCLASS	X
6	DM NUMBER CTOR	X
7	DM STRING CTOR	X
8	DM STRING TOSTRING	X
9	DMI RANDOM USED ONLY ONCE	
10	DMI USING REMOVEALL TO CLEAR COLLECTION	
11	ISC INSTANTIATE STATIC CLASS	
12	RCN REDUNDANT NULLCHECK OF NONNULL VALUE	
13	REC CATCH EXCEPTION	
14	SBSC USE STRINGBUFFER CONCATENATION	X
15	SIC INNER SHOULD BE STATIC	X
16	SS SHOULD BE STATIC	X
17	UM UNNECESSARY MATH	X
18	UPM UNCALLED PRIVATE METHOD	X
19	URF UNREAD FIELD	X
20	WMI WRONG MAP ITERATOR	X

An issue is selected if it satisfies the following combination of the five criteria: $(A \vee B) \wedge C \wedge D \wedge E$.

Table II shows all the issues selected as objects in the experiment. The first column indicates the numerical ID of the issue, the second column indicates its name, the last indicates whether the issue belongs to category performance or not (criterion A).

B. Variable selection and Hypotheses Formulation

Since the goal of the study is to evaluate the relationship of issues with time efficiency the only dependent variable is the execution time t. We will consider two variants of the same code: either containing the issue (I) or with the issue refactored out (R). Therefore the main factor we use is the code type, $C \in \{I, R\}$.

In addition we measure and control other independent variables:

- the specific issues ($Issue \in 1..20$) in the set of issues selected as described above,
- the platform (P), both hardware and software, where the experiment is conducted,
- the batch run (B) of each specific experiment.

Given our original research question and the selected variables we can formulate our null and alternative hypotheses, where the subscript indicate the level of the factor.

- $H_0 : t_I \geq t_R$
- $H_a : t_I < t_R$

C. Instrumentation and Experiment Design

The instrumentation required for our experiment is a software framework that allow the measurement of the execution times of the two different code fragments. Inspired by the JUnit[2] framework for automated software

[2]http://www.junit.org/

testing we developed a very simple framework. It consists of an abstract class, `Experiment`, that can be extended by concrete experimental classes. Each experimental class must provide two methods `performWithIssue()` and `performWithoutIssue()` that contain respectively the code including the issue and with the issue refactored out. In addition the method `setUp()` may be optionally redefined to prepare for the execution. For instance the experimental class for the issue DMI USING REMOVEALL TO CLEAR COLLECTION can be written as follows:

```
public class
DMI_USING_REMOVEALL_TO_CLEAR_COLLECTION
extends Experiment {
  Collection<Integer> col;
  private void setUp(){
    col = new LinkedList<Integer>();
    for(int i=0;i<1000;i++){
      col.add(Integer.valueOf(i));
    }
  }
  public void performWithIssue() {
    col.removeAll(col);
  }
  public void performWithoutIssue() {
    col.clear();
  }
}
```

The execution times of the methods `performWithIssue()` and `performWithoutIssue()` are expected to be in the order of nanoseconds. Unfortunately the standard measurement methods are not able to record precisely times at such order of magnitude. For this reason, the execution of each method is repeated consecutively a very high number of times (e.g. 1 million) to accumulate enough time to be detected by system APIs. We assume that each execution of the measured methods is independent on each other. This is true if no attribute is used except those initialized in the `setUp()` method.

The framework provides the method:

`perform(int nSamples , long nIter)`

that returns the results of the experiment in terms of the execution times. It takes as parameters two integers: the number of measurement samples to be generated (`nSamples`, set to 100 by default), and the number of iterations of the perform methods (`nIter`, set to 1 million by default). At the end of the experiment we will have `nSamples` samples, each of them representing the execution times of `nIter` iterations of both perform methods. We decide to have a batch of 6 runs of the basic experiment; each run was carried on at different random times during the day to compensate the possible confounding effect of periodical tasks performed by the operating system.

In addition, since the software and hardware platform is extremely relevant in terms of complexity when compared to the experimented code fragments, we decided to execute the experiment batch on three different platforms. Table III contains the characteristics of the platforms that hosted the experiments.

TABLE III
LIST OF PLATFORMS HOSTING THE EXPERIMENTS

Platform	U	W	M
Operating System	Ubuntu 10.10 kernel 2.6.35-25	Windows 7 Home Premium	Mac OS X 10.6.6 Darwin 10.6.0
Bits	64	64	64
Processors	2	2	2
Proc. Type	Intel Core 2 T5270	Pentium Dual Core T4500	Intel Core 2 Duo
Proc. Freq.	1.40 GhZ	2.30 GHz	2.66 GHz
Memory	2 GB	4 GB	4 GB
Java SE	1.6.0	1.6.0	1.6.0
build	_22-b04	_23-b05	24-b07

D. Analysis methodology

The goal of data analysis is to apply appropriate statistical tests to reject the null hypothesis. The analysis will be conducted separately for each issue in order to evaluate which one has an actual impact on time efficiency.

First of all we will test the null hypothesis H_0 for each issue across all platforms. Then we will analyze separately the different platforms.

Since we expect the values not to be normally distributed, we will adopt non parametric tests, in particular we selected the Mann-Whitney test [20]. Since the hypothesis is clearly directional the one-tailed variant of the test will be applied. We will draw conclusions from our tests based on a significance level $\alpha = 0.01$, that is we accept a 1% risk of type I error – i.e. rejecting the null hypothesis when it is actually true. Moreover, since we perform multiple tests on the same data – precisely twice: first overall and then by platform – we apply the Bonferroni correction to the significance level and we actually compare the test results versus a $\alpha_B = 0.01/2 = 0.005$.

After testing our experimental hypothesis, we will also check the potential confounding effect introduced by the co-factors: the platform and the different batch runs. Since the co-factors have more than two levels, we analyze the dependence of execution time on them using the Kruskall-Wallis rank sum test [20]. The null hypotheses we will attempt rejecting is that the co-factors have no effect on the dependent variable (time).

E. Validity evaluation

We identify two important threats to the validity of the experiment. The first threat affects the internal validity: experiments are executed inside an operating system, hence confounding factors could affect final results. Moreover, it is possible that the execution times for individual instructions are not independent from the execution history [10], because of caches and pipelines in processors, that could also cause the appearance of timing anomalies: therefore, we accept that the execution time of individual instructions may vary depending on the state of the processor in which they are executed, because we can not control the processor and avoid

the hardware-related problems. However, it is possible to take some counter measures to reduce the noise introduced by the upper levels (OS and VM): we repeat the experiment 6 times on three different operating systems and machines, obtaining overall 1800 samples for each version of the code, and we isolate as much as possible the environment in which the experiment program runs, disabling for instance network and network routines or avoiding to launch the program in the same time of operating system subroutines. Furthermore, the experiment is the only user program that runs in the machine. All these provisions do not delete the confounding factors, but limit them and let us to have a reduced noise on results.

The second threat is a construct threat: if a difference is found, we say that the cause of the difference is the refactoring action. However, the platform on which the code runs could affect results. Therefore, there are generalization problems: we try to control this threat by using three different platforms. Moreover, we make available on our website [3] the Eclipse project the experiment framework developed and we invite other researchers to repeat the experiment and compare the results with ours. In this way it is possible to build up a benchmark and make the empirical validation of the impact of issues on efficiency more reliable.

V. Analysis and Interpretation

The data collected during the experiments are summarized in table IV, which reports the average execution times expressed in milliseconds, for the three different platforms and separating the execution time of the code containing the issue (t_I) from the execution time of code with the issue refactored out (t_R).

We can immediately observe a wide variability of times and small differences mainly among different issues, but also to a smaller extent between platforms. In order to report in the same diagram such varying values we opted for the rest of this analysis to plot times using a logarithmic scale.

Columns p in table IV report the p-values of Mann-Whitney tests carried on overall and by platforms (W, U, M); statistically significant values are reported in bold face. The boxplot of figure 1 reports the execution times recorded in the experiment, divided by issue, in practice it add the dispersion to the information provided in the first four columns of the table. Execution times of code containing the issue is drawn in black, while for code with the issues refactored out (R) it is represented in red. A gray background is present corresponding to the issues for which we can reject the null hypothesis. The boxplot in figure 2 is similar but it reports the execution times recorded in each platform.

We can observe a range of patterns in terms of hypothesis rejection overall (figure 1) and for specific platforms (figure 2). On one side, the null hypothesis can be rejected both overall and for every tested platform for issues 2, 4, 6, 7, 9, 10, 11, 14, 16, 19, and 20. At the opposite side, the null hypothesis could not be rejected neither overall nor on any platform for issues 8, 12, and 18. Among the remaining issues: for issues 3 and 13 we rejected H_0 overall and on two out of three platforms, for issue 1 we could reject overall and on 1 platform, and for issues 5, 15, and 17 we could reject only on one platform.

[3] http://softeng.polito.it/vetro/confs/InfQ2011/EfficiencySmells.zip

TABLE V
P-values of Kruskal-Wallis test for co-factors

ID	Platform		Run	
1	$\leq$ 0.001	*	0.01	
2	$\leq$ 0.001	*	0.60	
3	$\leq$ 0.001	*	$\leq$ 0.001	*
4	$\leq$ 0.001	*	0.02	
5	$\leq$ 0.001	*	0.16	
6	$\leq$ 0.001	*	0.02	
7	$\leq$ 0.001	*	$\leq$ 0.001	*
8	$\leq$ 0.001	*	$\leq$ 0.001	*
9	$\leq$ 0.001	*	0.19	
10	$\leq$ 0.001	*	0.18	
11	$\leq$ 0.001	*	$\leq$ 0.001	*
12	$\leq$ 0.001	*	0.04	
13	$\leq$ 0.001	*	$\leq$ 0.001	*
14	$\leq$ 0.001	*	$\leq$ 0.001	*
15	$\leq$ 0.001	*	$\leq$ 0.001	*
16	$\leq$ 0.001	*	0.05	
17	$\leq$ 0.001	*	$\leq$ 0.001	*
18	$\leq$ 0.001	*	$\leq$ 0.001	*
19	$\leq$ 0.001	*	$\leq$ 0.001	*
20	$\leq$ 0.001	*	$\leq$ 0.001	*

The effect of co-factors on the main dependent variable has been checked with the Kruskal-Wallis test, whose results are reported in table V.

We observe that, concerning the Platform, the hypothesis can be rejected for all issues. While, the batch Run influenced the execution time of 11 out of 20 issues.

A. Discussion

Based on results in table IV, 11 of the issues selected have undoubtably a negative impact on time efficiency, since there is a statistically significant difference in all conditions. Such issues are:

2 A primitive boxed value is constructed and then immediately converted into a different primitive type (e.g., $newDouble(d).intValue()$) instead of performing direct primitive coercion (e.g., $(int)d$).

4 A method invokes a Boolean constructor, instead of using $Boolean.valueOf(...)$

6 Code uses $newInteger(int)$ whereas $Integer.valueOf(int)$ should be used, because it allows caching of values to be done by the compiler, class library, or JVM.

7 The $java.lang.String(String)$ constructor is used instead of $String$ parameter directly.

9 Code creates a $java.util.Random$ object, uses it to generate one random number, and then discards the $Random$ object. Subsequently, to generate a new random number, a new $java.util.Random$ object is created. Code should be refactored so that the $Random$ object is created once and saved to be invoked each time a new random number is needed.

TABLE IV
SUMMARY OF EXECUTION TIMES.

Platform:	all			M			U			W		
ID	t_I	t_R	p	t_I	t_R	p	t_I	t_R	p	t_I	t_R	p
1	34.72	34.48	$<$ **0.001**	55.41	55.32	0.01	47.04	47.13	1.00	1.70	1.00	$<$ **0.001**
2	8.39	2.78	$<$ **0.001**	10.03	3.67	$<$ **0.001**	12.70	2.73	$<$ **0.001**	2.45	1.93	$<$ **0.001**
3	68.10	35.27	$<$ **0.001**	109.43	54.32	$<$ **0.001**	90.78	46.99	$<$ **0.001**	4.10	4.51	1.00
4	9.81	5.43	$<$ **0.001**	10.57	4.19	$<$ **0.001**	13.63	7.18	$<$ **0.001**	5.24	4.93	$<$ **0.001**
5	180.20	183.31	1.00	167.03	155.49	$<$ **0.001**	237.84	242.74	1.00	135.72	151.69	1.00
6	9.65	4.78	$<$ **0.001**	10.64	3.07	$<$ **0.001**	13.77	7.12	$<$ **0.001**	4.53	4.16	$<$ **0.001**
7	14.72	5.15	$<$ **0.001**	17.29	4.20	$<$ **0.001**	18.88	7.09	$<$ **0.001**	7.99	4.16	$<$ **0.001**
8	84.16	88.54	1.00	75.29	75.79	1.00	113.92	121.16	1.00	63.26	68.67	1.00
9	2162.58	1117.11	$<$ **0.001**	326.57	164.02	$<$ **0.001**	3687.10	1901.80	$<$ **0.001**	2474.06	1285.49	$<$ **0.001**
10	468.66	213.77	$<$ **0.001**	411.45	210.21	$<$ **0.001**	728.38	278.32	$<$ **0.001**	266.16	152.78	$<$ **0.001**
11	8.70	5.08	$<$ **0.001**	8.24	4.17	$<$ **0.001**	13.15	6.84	$<$ **0.001**	4.70	4.22	$<$ **0.001**
12	591.41	592.08	0.74	80.33	80.22	0.42	1671.77	1673.89	1.00	22.14	22.14	0.47
13	35.81	35.47	$<$ **0.001**	55.64	55.08	$<$ **0.001**	47.25	47.31	1.00	4.54	4.03	$<$ **0.001**
14	561.95	302.34	$<$ **0.001**	455.71	268.63	$<$ **0.001**	767.91	409.92	$<$ **0.001**	462.24	228.46	$<$ **0.001**
15	6.98	7.04	0.08	5.28	5.66	1.00	8.82	9.07	1.00	6.83	6.39	$<$ **0.001**
16	9.71	8.62	$<$ **0.001**	10.78	8.35	$<$ **0.001**	13.77	13.45	$<$ **0.001**	4.57	4.05	$<$ **0.001**
17	592.05	594.41	0.55	3.84	4.15	1.00	1767.72	1775.06	0.01	4.59	4.01	$<$ **0.001**
18	537.67	544.22	1.00	462.41	462.53	1.00	707.32	716.04	1.00	443.28	454.10	1.00
19	11.80	11.04	$<$ **0.001**	13.94	13.89	$<$ **0.001**	16.23	13.93	$<$ **0.001**	5.22	5.30	$<$ **0.001**
20	582.91	539.86	$<$ **0.001**	558.12	514.13	$<$ **0.001**	668.55	633.61	$<$ **0.001**	522.05	471.84	$<$ **0.001**

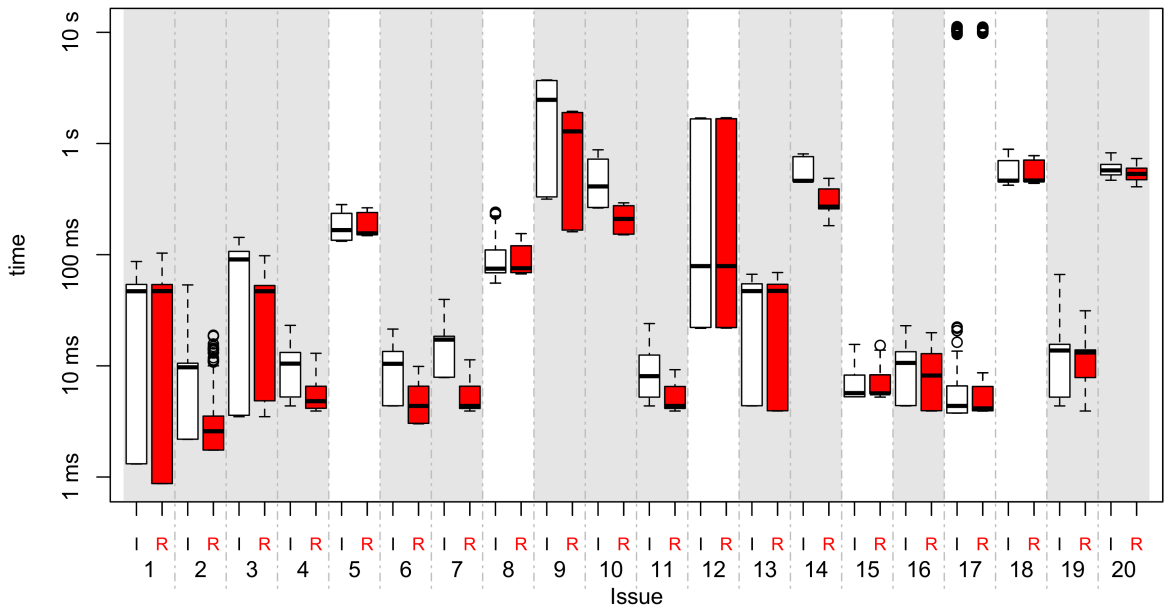

Fig. 1. Boxplot of execution times for all issues.

10 The code removes all elements from a collection c, using $c.removeAll(c)$ instead of $c.clear()$.

11 A class allocates an instance of a class that only supplies static methods. The refactoring action is to use the static methods directly using the class name as a qualifier.

14 Code builds a $String$ using concatenation in a loop instead of using $StringBuffer$.

16 A class contains an instance final field that is initialized to a compile-time static value. Since the field is immutable for each object of the class, it should be static.

19 A field which is never read

20 Code accesses the value of a Map entry, using a key that was retrieved from a keySet iterator. It is more efficient to use an iterator on the $entrySet$ of the map, to avoid the $Map.get(key)$ lookup.

More than half of the issues (nr 2, 4, 6, 7, 9, 11) concerns a useless creation of objects. The other issues are related to different problems, relating to inefficient, albeit functionally correct, set of operations.

Being known the number of times that the code containing the issues is invoked, it is possible to estimate the average delay that each of these issues bring to the code. The code fragments invoke issues only once to minimize the confounding factors: therefore the total number of invocations is 1 million times. However, issues 14 and 20 are inside a `for` cycle of respectively 5 and 10 iterations, thus they are executed

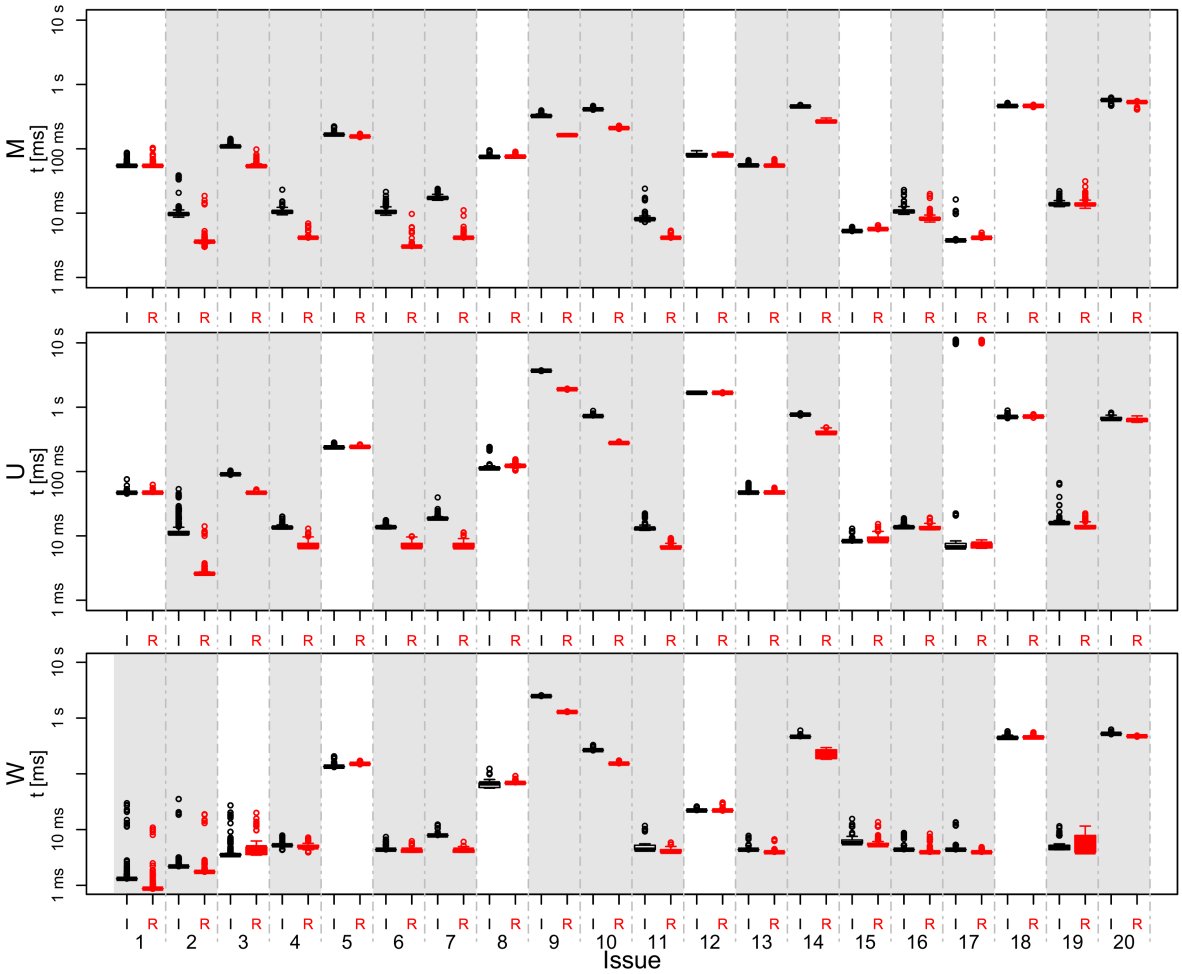

Fig. 2. Boxplot of execution times for all issues.

TABLE VI
MEAN EXPECTED DELAY $[ns]$ OF VERIFIED ISSUES

		Platform		
Issue	Iterations	U	W	M
2	1 M	8.23	0.43	6.14
4	1 M	6.43	0.43	6.29
6	1 M	6.67	0.43	7.40
7	1 M	11.81	3.91	13.02
9	1 M	1786.00	1185.80	159.60
10	1 M	449.20	113.00	201.10
11	1 M	6.29	0.43	3.90
14	5 M	74.74	51.16	37.81
16	1 M	0.40	0.43	2.36
19	1 M	2.21	0.44	0.11
20	10 M	4.36	4.96	4.07

5 and 10 millions times. Table VI contains the estimated delays, in nanoseconds, for each issue and platform, computed aggregating the measurements of the different batches.

All issues concerning the useless creation of objects (except nr 9) have similar unitary delays: few nanoseconds (3 to 11) in environments U and M, less than 1 in environment W. Similar delay is for the wrong map iteration (issue 20). The issue nr 9 (useless Random object) has the highest delay, that is in the order of magnitude of 1 μs. Also issue 10 (emptying the content of a collection) exhibits high delays (in the order of magnitude of some hundreds of ns), whereas issue 14 causes a delay of tens of ns. The smallest delays are those ones of issue 16, a field that should be static, and issue 19, a field that is never read. In real contexts these numbers can easily reach the order of ms and s: in real projects there are millions of lines of code where there can be millions of these simple issues or even billions if they are inside $for/while$ cycles. Moreover, these figures may be directly relevant in Real Time Applications, where the usage of Java is steadily increasing (for instance, Boeing has adopted real-time Java in drone aircrafts, and the United States Navy decided to use it in its next-generation battleships [11]).

On the basis of the above findings, we can assert that likely also the following issues, which were not object of our experiments, have impact on time efficiency, because related to the previous eleven: BX BOXING IMMEDIATELY UNBOXED, BX UNBOXED AND COERCED FOR TERNARY OPERATOR, DM BOXED PRIMITIVE TOSTRING, DM FP NUMBER CTOR (similar to issues 2, 6), DM STRING VOID

CTOR (similar to issue 7), issue UUF UNUSED FIELD (similar to issue 19).

Moreover, we observe that issues 8, 12 and 18, instead, do not have any negative impact on performance. We further investigate this fact computing the estimated differences between the two set of execution times ($t_I - t_R$). Issue DM STRING TOSTRING (nr 8, call $toString()$ on a $String$) has negative differences both overall and in every platform; issue RCN REDUNDANT NULLCHECK OF NONNULL VALUE (nr 12, check of a known null value) has a significant difference only in platform U, whilst UPM UNCALLED PRIVATE METHOD (nr 18, a private method never used) is significantly different under all conditions. This data shows that the refactored code of these three issues perform worst than the original code: this is an unexpected result. Nevertheless, it is probable that optimization enforced by the compiler and/or the hardware deletes the negative effect of the three issues at run time. All the three issues concern useless operations and bad programming practice: even if they do not impact the efficiency, refactor the code is worthy to increase its maintainability or decrease its complexity. Similar issues in Findbugs, not selected for our experiment, are : RCN REDUNDANT (COMPARISON OF NULL AND NONNULL VALUE, COMPARISON TWO NULL VALUES, NULLCHECK OF NULL VALUE, NULLCHECK WOULD HAVE BEEN A NPE) that are similar to issue nr 12, and UMAC UNCALLABLE METHOD OF ANONYMOUS CLASS that is similar to issue 18.

VI. Summary and conclusions

We set up an experiment to quantitatively assess the impact of selected FindBugs issues on time efficiency. We selected, through expert judgments, 20 representative issues and for each one we compared the average execution time of a code fragment containing that issue against the same code with the issue refactored out. The measurements were conducted on three different platforms.

Experts' examination of issues revealed that, mainly because FindBugs issue taxonomy is exclusive, a few issues having a potential impact on performance in fact do not belong to the Performance category. Moreover experiment revealed that 3 out of 11 verified issues do not belong to the Performance category, while 2 out of 3 unverified issues belong – apparently without justification – to that category. Overall, based on our findings we can select 11 issues that have a proved and quantified (table VI) impact on time efficiency

We provide these results to programmers that develop their applications in environments similar to those ones we used in the experiments, enabling them to estimate the delay provoked by code that violates the same issues we identified, or a subset.

We plan to investigate deeper the issues without proved impact on efficiency and to repeat the experiment for other issues, in different platforms. We are also conducting similar experiments in industrial contexts. Finally, we make available the code of the experimental Java framework developed to enable other researchers repeating the tests on their own platforms: we will be grateful to collect results and build a benchmark.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[2] S. Kim and M. D. Ernst, "Which warnings should i fix first?" in *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2007, pp. 45–54.

[3] ——, "Prioritizing warning categories by analyzing software history," in *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007, p. 27.

[4] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb, "An evaluation of two bug pattern tools for java," in *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 248–257.

[5] S. Wagner, J. Jrjens, C. Koller, P. Trischberger, and T. U. Mnchen, "Comparing bug finding tools with reviews and tests," pp. 40–55, 2005.

[6] N. Ayewah and W. Pugh, "The Google FindBugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, pp. 241–252.

[7] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM, 2007, pp. 1–8.

[8] A. Vetro', M. Torchiano, and M. Morisio, "Assessing the precision of findbugs by mining java projects developed at a university," in *Proceedings of MSR 2010*, I. C. Press, Ed., 2010, pp. 110–113.

[9] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[10] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[11] T. Harmon and R. Klefstad, "A survey of worst-case execution time analysis for real-time java," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 232, 2007.

[12] G. Bernat, A. Burns, and A. Wellings, "Portable worst-case execution time analysis using java byte code," in *In Proc. 12th Euromicro International Conference on Real-Time Systems*, 2000, pp. 81–88.

[13] E. Y.-S. Hu, G. Bernat, and A. Wellings, "Addressing dynamic dispatching issues in wcet analysis for object-oriented hard real-time systems," *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, vol. 0, p. 0109, 2002.

[14] I. Bate, G. Bernat, G. Murphy, and P. Puschner, "Low-level analysis of a portable java byte code wcet analysis framework," *Real-Time Computing Systems and Applications, International Workshop on*, vol. 0, p. 39, 2000.

[15] I. Bate, G. Bernat, and P. Puschner, "Java virtual-machine support for portable worst-case execution-time analysis," in *In Proc. 5th IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing*, 2002, pp. 83–90.

[16] D. S. Hardin, "Real-Time Objects on the Bare Metal: An Efficient Hardware Realization of the JavaTM Virtual Machine," *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, pp. 0053+.

[17] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, "A survey of adaptive optimization in virtual machines," *Proceedings of the IEEE*, no. 2, pp. 449–466, Feb.

[18] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2004, pp. 132–136.

[19] R. Van Solingen and E. Berghout, *Goal/Question/Metric Method*. McGraw-Hill Inc.,US, January.

[20] M. H. . D. A. Wolfe, *Nonparametric Statistical Methods*. New York: John Wiley & Sons, 1973.