BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2018
http://excel.fit.vutbr.cz

Towards the detection of performance degradation

Jiří Pavela*, Šimon Stupinský**

Abstract
Current tools for project performance analysis focus mostly on detecting selected performance bugs in the source code. Although being useful, results of such tools cannot provide evaluation of project's overall performance which is often crucial for development of large applications. Building on our previous works, we aim to provide a solution to a long term monitoring of performance changes using the tool chain of *performance data collection*, *regression analysis*, and subsequent *automated detection of performance changes* performed. We evaluated our solution on series of artificial examples and we were able to detect about 85% of performance changes across different performance models and errors, and estimate their severity (e.g. constant, linear, etc). The proposed solution allows user to deploy new method of code review — possibly integrated into continuous integration — and reveal performance changes introduced by new versions of code in early development.

Keywords: Performance — Continuous integration — Automated degradation detection — Difference analysis — Clusterization — Regression analysis

Supplementary Material: Project repository

*xpavel32@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*
**xstupi00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

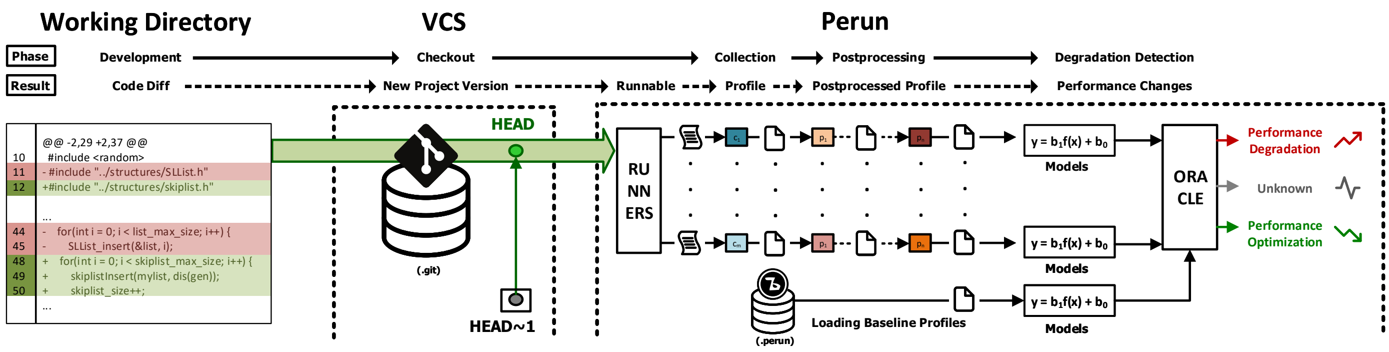
1. Introduction

Although a light-weight performance analysis can often be useful in determining the current performance of a program, the bigger picture of the performance evolution during the development can still remain hidden. We simply need more thorough approaches both for the collection and for the interpretation of the profiling data. Such methods were introduced in our previous works [2, 3, 4] and, moreover, were integrated in the existing performance management system [1] And indeed, we achieved effective control over the project performance.

However, performance degradation in a long term is often subtle. It can span across many versions or iterations of the code, and it can be especially tricky to discover. Since even with our already well-established methods of performance interpretation and its link to the corresponding code versions, the actual evaluation of performance changes across different code versions still has to be done manually by the user. And manually comparing the performance of different project versions is especially tedious, inaccurate and error-prone process. So in the end it will most likely be skipped by the user.

The automation of such process is a challenging task as it requires the analyzer to obtain information about project history, provide tools that are able to estimate dynamic time behavior of code fragments and lastly their comparison and evaluation across different versions — preferably with indicators that are easy to grasp and accurate. These tools have to co-operate and the whole process has to be fully automated.

Hence we exploited our previous results and proposed a framework for automated detection of performance changes based on models of performance behavior as obtained by the regression analysis [3]. To achieve a more precise results, we also remodeled the profiling data acquisition and updated post-processing methods. This way we can provide estimates of performance changes — be it degradation or optimization — during the code development for a broad range of programs. Moreover, our solution not only provides



```
@@ -2,29 +2,37 @@
 #include <random>
- #include "../structures/SLList.h"
+#include "../structures/skiplist.h"

...
-    for(int i = 0; i < list_max_size; i++) {
-        SLList_insert(&list, i);
+    for(int i = 0; i < skiplist_max_size; i++) {
+        skiplistInsert(mylist, dis(gen));
+        skiplist_size++;
...
```

Figure 1. Overview of the automatic detection method of performance changes over project repository. This paper focuses mainly on the degradation detection, collection and partly postprocessing methods [1]

the indication whether any performance change was detected, but also rough estimates the severity of these changes in form of a mathematical function describing the measured difference between two performance profiles and computes the confidence of the detection and classification process.

By integrating our method to the existing performance management system, we obtain a powerful difference analysis. Now, whenever we release a new version of our project, we can compare the newly computed regression models (so called *target*) with stored best models of previous stable version (so called *baseline*) and report a list of performance changes. And by providing three crucial aspects for each detected change — precise location, severity and confidence — we can achieve a high ratio of performance fixes. The main advantage of this approach is that it can be easily integrated into any development workflow (e.g. into continuous integration) serving as an additional code reviewer before any release is made and helping catching performance bugs early in the development.

Most of the current works in the code performance analysis field focuses on tools that are capable of detecting and locating performance-suspicious code fragments or hotspots. Many of these tools leverage *static code analysis* along with the artificial intelligence and machine learning techniques [5], detection of similar *memory-access patterns* in code [6] or *rule-based detection* which utilizes *efficiency-rules* extracted from performance bug trackers [7]. However, these tools operate predominantly on the source code (hence the static analysis) and thus can not provide the user with enough information about the actual impact of detected performance bugs in the production.

Others [8, 9] introduce test-like environments where performance bugs are discovered similarly to functional bugs — by running predefined set of tests on different volumes of data. This method is easily adoptable by users as it resembles familiar functional testing and can be in fact integrated into project's current test suite. On the other hand, the results may be too general (e.g. test finished under specified time threshold) and pinpointing the actual source of the problem must be done by the user.

In the end, none of the mentioned tools addresses the long term gradual development process of a program where code is being constantly written, changed or removed. Our proposed method and its integration in the Perun tool fills in the missing piece and provides a way to continuously monitor project's performance status by executing automated performance degradation analysis based on a collected performance profiles from various project versions. This helps user in identifying code changes that could introduce performance problems into the project's codebase or checking different code versions for subtle, long term performance degradation scenarios.

2. Automatic Detection of Degradation

We will first outline our framework of automated detection of performance changes on git repositories[1] as pictured on Figure 1. Note that this was proposed and developed together with the main coordinator of the Perun project [1]. Whenever the developer wants to release the new version of the project the following steps are executed in order to automatically detect performance changes:

1. User creates a set of changes, and checks them out in the project repository. A new commit of the project is thus created.
2. The commit triggers new build and we use our *trace collector* (see Sec. 3) over this newly build

[1]Note however, that our method is not dependent on the underlying version control system

binary and collect the raw performance data, in particular the time consumption of functions.

3. These raw data are first clusterized (see Sec. 4.1) and so for each resource, we have its time consumption and classified cluster. For these resources, we find a regression performance model (see Sec. 4.2), which predicts the values of runtime for different values of clusters. We call these resulting profiles *target profiles*.
4. We retrieve the profiles of the parent versions from the persistent storage of Perun. These profiles serve as a *baseline profiles* and provide the performance status estimates of the previous versions and will be compared against.
5. For each pair of target and baseline profiles, with the compatible configurations (i.e. profiles which were collected for the same execution of the binary), we perform the actual performance degradation detection method as described in 5.
6. The method returns set of located performance changes. For each such change we report its *severity* (i.e. how good or bad the change is), its location (i.e. where the change has happened) and our confidence (i.e. whether the change is most likely real).

3. Collecting the performance data

After the user commits new version of the project we first collect the performance data from the newly build project binary (or any other command or script). These performance data are acquired using the *trace collector* — binary profiler based on our last year's work [3] and authored by Jiří Pavela.

The previous version of our collector used code injections during the program compilation to gather the time and size related information about the function execution. However, this approach is inconvenient as it requires access to the project source files and modification of compilation process — requirements not satisfiable in many scenarios. Thus in order to eliminate both dependencies and allow the user to perform profiling on more broader range of executable files, we released an enhanced version of this profiler.

We decided to build the new collector on well-established existing solution that would support *tracing/probing* and dynamic code injection in executable files. These requirements were satisfied by *Perf* and *SystemTap* tools.

Perf [10] is a Linux utility program for instrumentation, tracing and probing of kernel and user processes. The probing is realized using the *kprobes* and *uprobes* [11] modules which gather timestamps whenever specified location is reached during program execution. Although *Perf* is available on the most of the Linux distributions and does not have external dependencies, its tracing capabilities are inferior to those of *SystemTap*.

SystemTap [12] is an infrastructure for detailed analysis of running programs with capabilities similar to those of *Perf*. Moreover, *SystemTap* offers advanced methods for the configuration of the probes and their behavior on firing, even though it uses the same *kprobes / uprobes* kernel modules as Perf does. *SystemTap* tracing and probing is controlled by developer-defined scripts which are subsequently translated into new kernel modules and then dynamically loaded into the kernel. The drawback is that *SystemTap* is a third-party software (although it is well-supported by the Linux systems) and requires the *dbgsym* version of kernel to operate.

Still, we choose to build our solution over *SystemTap* mainly for its flexibility with probes specification and scripting-like nature which allows precise control of the collection process.

The actual profiling data collection is realized by injecting probes at specified code locations (currently we support only function entries and exits). When the location is reached during the program execution, the probe fires and logs *timestamp* record stored into the kernel buffer and later saved to the output file.

Our collector then work in the following three step process described below and visualized in Figure 2:

1. The collector first loads the collection configuration, where user specifies which functions will be probed in the input binary. According to this configuration, we generate the SystemTap script with all required probes included.
2. The script is translated, loaded into the kernel and the probed binary is executed. The probed profiling data are stored in the output trace log.
3. After the successful collection, the trace log is loaded and each recorded timestamp is converted into the actual time spent in each traced function call.

The collector does not provide estimation of some independent variable required for degradation detection. The previous solution using user-defined code annotations was insufficient and thus we propose a heuristic called *clusterization* (see Section 4.1) to estimate this independent variable (preferably the input data size) and hence we only log the time consumption of the functions and the call order.

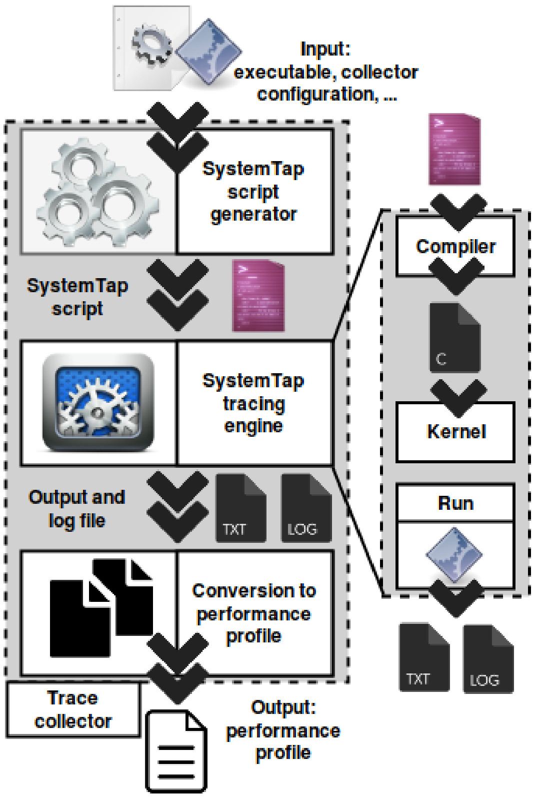


Figure 2. Schema of performance data collection process using the *trace collector*. Collection process requires executable file, collector configuration and provides output performance profile with records of time consumption of functions in the code.

4. Postprocessing the data

The raw performance data obtained from the previous collection phase must be further processed to provide data suitable for difference analysis and degradation detection. We postprocess profiles first by *Clusterization* and then by *Regression analysis* yielding a set of performance models for each function in the code.

Figure 3 shows an example of the resources post-processed by clusterizer and regression analysis. The actual resources were collected by *memory collector* [4], which logs amounts of allocated memory and were clustered w.r.t sort order of the allocated sizes. The bottom figure shows the allocations interpolated by selected models, with linear models being the best.

4.1 Clusterization

The clusterization postprocessing module is authored by the Perun's maintainer Tomáš Fiedor [1]. We include a brief description of this unit, since it is an essential part of the postprocessing chain.

The clusterizer is a simple heuristic and is used to classify the gathered resources to clusters by similarity of their values. We employ two simple strategies to cluster the resources — by *sliding window* or by *sort order of values*. The sliding window iterates the sorted values and performs aggregation of similar values into the standalone clusters according to the window dimensions. The other techniques works in straightforward way. Both techniques are especially useful in conjunction with regression analysis as they allow us to derive the *independent* variable data (see 4.2) for the collected resources and hence allow us to model performance.

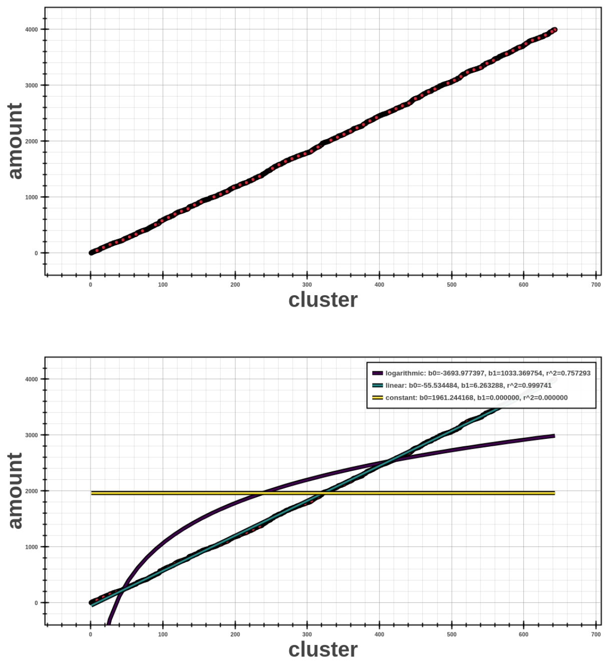


Figure 3. Result of clusterization method on data obtained from *memory collector*. Second image shows the usefulness of clusterization in conjunction with regression analysis — specifically the ability to infer the *independent variable* from collected data [1].

The trace collector cannot reliably collect the workload size (or any other independent variable) for each function call — information necessary to estimate the complexity. However, we assume that analyzed programs are deterministic and hence function's behavior and running time will be deterministic as well. With this assumption in mind, it is safe to perform the clusterization according to the running time of the function calls and estimate the runtime complexity.

4.2 Regression Analysis

Regression analysis (authored by Jiří Pavela) is a method used in statistics to deduce the relationship between *dependent* (e.g. function runtime) and *independent* variables (e.g. size of the underlying structures). This relationship is expressed as a *regression function* and can be used to predict the *dependent* variable values for every value of *independent* variable. In order to evaluate the goodness-of-fit of this function, i.e. how well does the model explain the relationship between regressed variables, we use the *coefficient of determination* (R^2). The higher values of R^2 signifies a better model of resources.

In our scenario we use the workload size (estimated by clusterizer) as the *independent* variable and the function time consumption as the *dependent* variable. The regression analysis then produces set of mathematical functions that describe the behavior of code functions in the program, i.e. consumed time in relation to the input data size.

The implementation of the regression analysis is already covered to great extent in our previous works [2, 3]. It is worth noting though that we made several changes since the release of the first version. In particular, we enhanced the following:

- We updated and optimized the computation process of regression models. Most models are now computed using general formulae 1 and 2 for coefficients $\hat{\beta}_0, \hat{\beta}_1$ of function model $f(z) = \hat{\beta}_0 + \hat{\beta}_1 z$. Currently only quadratic model uses the custom computation as it requires more coefficients. Also several fixes were made to correct computation errors.

$$\hat{\beta}_0 = \frac{\sum y_i - \hat{\beta}_1 \sum f(x_i)}{n} \tag{1}$$

$$\hat{\beta}_1 = \frac{n \sum f(x_i) y_i - \sum f(x_i) \sum y_i}{n \sum (f(x_i))^2 - (\sum f(x_i))^2} \tag{2}$$

- We introduced a new class of *derived* complexity models. These derived models are computed from the results of different, already computed models. The example of such model is the *constant* model, since R^2 can not be computed using standard formulae. Instead, as a heuristic we use the parameters of linear model to infer the goodness-of-fit for the constant model.

5. Detection of Changes

The last step in the whole process is the actual detection of performance changes (authored by Šimon Stupinský) according to the obtained regression models. The input of our method of detection is the pair of profiles, where the first is called the *target profile* and represents the performance of our new released version of project, while the latter is called the *baseline profile* and represents the stable base against which we will compare the *target*.

In addition to the actual detection of performance change, we will try to classify the *severity* of these changes. We represent the severity of performance

```
1  Method check ($\mathcal{M}_b$, $\mathcal{M}_t$)
2      foreach ($m_b$, $m_t$), ($m_b^n$, $m_t^n$) $\in \mathcal{M}_b, \mathcal{M}_t$ do
3          $\mathcal{M} = \{m_b, m_t, m_b^n, m_t^n\}$
4          $dp_{t-b} = m_t - m_b$
5          $dp_{t-b}^n = m_b^n - m_t^n$
6          $\varepsilon_{rel} = \sum dp_{t-b} \ / \ |dp_{t-b}|$
7          $\mathcal{M}$.add(FindPoly($dp_{t-b}$), 1)
8          for $deg = 0$ to MAX_DEG do
9              $\rho$ = FindPoly($dp_{t-b}^n$, $deg$)
10             if $\rho[residual] \leq \xi$ then
11                 break
12             $\mathcal{M}$.add($\rho$)
13         Classify($\mathcal{M}$, $\varepsilon_{rel}$)
```

Algorithm 1: Algorithm of detection and classification performance changes based on input baseline ($\mathcal{M}_b$) and target ($\mathcal{M}_t$) models and threshold ξ. We denote models by m, data points by dp and relative error by ε_{rel}.

changes by individual kind (such as linear) and by error rate, which denotes how big a change has occurred in comparison with the baseline. Moreover, we give a *confidence* rate to help developers decide whether the performance change really has occurred. In our methods we represent confidence by the minimum of coefficients of determination (R^2) of the best-fit models, which were used during the detection. This way we provide the user with the most accurate information about individual performance changes.

5.1 Algorithm of Detection

The base algorithm for detection of performance changes is depicted in Algorithm 1. We propose two variants which mainly differ in the actual classification of changes according to the computed data. These variants will be discussed in the corresponding subsections. Note, however, that for simplification we abstracted the algorithm, and in the implementation, some of the models are not computed for both methods.

The inputs of the main algorithm are sets of regression models of *baseline* and *target* profiles, which are denoted as $\mathcal{M}_b$ and $\mathcal{M}_t$ respectively. From both of these sets for each function we select its models with the highest value of R^2 (denoted as as m_b and m_t respectively)as well as its corresponding linear models (denoted as as m_b^n and m_t^n respectively).

For both pairs of best models (m_b and m_t) and linear models (m_b^n and m_t^n) we compute a set of data points (denoted as dp_{t-b} and dp_{t-b}^n respectively) by simple subtraction of these models (i.e. for each value of independent variable x we compute $m_t(x) - m_b(x)$). The intuition is that if error was injected to the baseline by the target version, then its model can be expressed as $m_t(x) = m_b(x) + m_e(x)$, where m_e denotes the model of the error. Hence we are trying to find such m_e.

Then we use regression analysis (see Section 4.2) to obtain a set of models for these subtracted data points. Moreover, for the first set of data points, corresponding to best-fit models, we compute the relative error, which serves as a pretty accurate check of performance change. We do not compute this relative error for linear models, since their R^2 can be quite low and hence the relative error can be too big.

We use the `numpy`[2] package to subsequently try to interleave the received data points using polynomials of various degrees (up to some `MAX_DEGREE`). `numpy` in this case returns a so called *residuals*, which signifies, whether increasing the degree of polynomial adds anything to the fitness of the data. Hence when the value of these residua are lesser than given threshold ξ, we break from the loop and stop.

All of these regressed models are then given to the concrete classify functions, which gradually returns detected degradations. Each method however uses different kinds of models.

5.2 Rate detection using coefficients

The first variant is a simple heuristic based on the results of *linear regression*, which models the relationship between independent variables x and dependent variables y as function $y = b_0 + b_1 \cdot x$.

In this variant we analyze only the linear models of target, baseline, and their subtraction. We compare the coefficients of linear model corresponding to the subtraction of the linear models with the coefficients of linear models of both original models. However, this check is executed only in the case of sufficiently high value of R^2. The observed properties of coefficients are depending on different possible changes between profiles. So to detect the constant change we compare intercepts (i.e. the coefficient b_0) of the linear models, and to detect linear change we compare slopes of the linear models (i.e. the coefficient b_1).

In the case we could not detect any change using linear models only, we use and analyze the subtraction of the best models.

5.3 Rate detection using polynomials

In the second variant we use *polynomial regression* to quantify the rate of the change, i.e. we represent the change in a form of nth degree polynomial function. The actual detection of the change is, however, detected according to the absolute and relative error of the best models of baseline and target profiles. We argue, that using using linear models can provide quite a fast classification rate, while having a worse detection rate. However, we compensate this by using it in combination with errors of subtraction of best models.

We think that well-fit interleaving of the data by polynomials of the certain degrees can pretty accurately classify how big change has occurred between profiles. In the main method we iteratively find such polynomial of smallest degree, that can interleave the data precisely.

6. Experimental Evaluation

We tested both methods on set of artificial examples consisting of profiles with selected types of models (constant, linear, logarithmic, quadratic, exponential and power) and profiles with injected errors, in particular profiles injected with constant, linear and quadratic changes. Moreover, we tested each type of the models to both favourable (optimization) and unfavourable (degradation) changes. In overall we tested six types of changes of 36 different pairs of profiles, and report the rate of detecting the change and also the rate of severity classification of the change.

Table 2 shows the results of our experimental evaluation. We denote with colours whether we correctly detected and classified the performance change. In case we misclassified the error, we report the result, i.e. that the performance change was more or less severe then in reality. Table 1 summarizes the detection and classification rate of both methods.

		$\pm c$	$\pm n$	$\pm n^2$	**overall**	**time [s]**
#1	d_r	80%	80%	100%	**86,67%**	5.26
	c_r	25%	55%	65%	**48,33%**	
#2	d_r	80%	100%	100%	**93,33%**	4.32
	c_r	80%	80%	0%	**53,33%**	

Table 1. Comparison of runtime, detection (d_r) and classification rate (c_r) of both methods for constant (c), linear (n) and quadratic (n^2) changes.

Constant Changes. We claim that detecting and classifying constant changes precisely is crucial for method evaluation, since these kinds of performance bugs are the most common and most likely to be overlooked. Detecting constant changes between versions were mostly successful with the exception of the logarithmic model. We believe this is caused by a small constant value in comparison with the big coefficients of logarithmic model. Hence, the shift of this type of model is not as obvious and detectable as opposed to the rest of models.

[2] https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html

The second method based on polynomials was in this case more successful and has proven all degradation marks with precise classification of the changes.

Non-constant changes. For non-constant changes, the results are quite diverse. The linear changes, where the absolute value of change increases with the value of independent variable, had similar results, with only problem recorded for power models. The quadratic error was correctly detected by all errors, however, it showed the highest misclassification rate.

#1 / #2	+c	-c	+n	-n	$+n^2$	$-n^2$
`cst`	OK / OK	n / OK	n^2 / c	OK / c	OK / n	n / c
`lin`	OK / OK	n / OK	c / OK	OK / OK	OK / n	n / n
`log`	NO / NO	NO / NO	OK / OK	OK / OK	OK / n	OK / n
`quad`	OK / OK	n / OK	n^2 / OK	OK / OK	OK / n	n / n
`exp`	n / OK	n / OK	OK / OK	OK / OK	OK / n	n / n
`pow`	n / OK	n / OK	NO / OK	NO / OK	OK / n	OK / n

Table 2. Comparison of two variants of performance change check for every model (rows) and change (columns). We denote with green if we correctly detected and classified the change, with yellow if we successfully detected but misclassified the change and with red if the check failed. If we misclassified the error, we report the actual classification result.

Overall the second method based on polynomials is better — it is slightly faster and with better detection and classification rate.

7. Conclusion

In this paper, we introduced a framework for long term and continuous performance change monitoring during the project development. We use the *version difference analysis* (integrated into performance profiling platform Perun [1]) to search for potential performance changes — along with their estimated classification, severity and certainty — against previous project versions. Using our methods, we were able to achieve approximately 90% of correct detection and 50% classification correctness on our set of examples.

Our future work will focus mainly on increasing the accuracy of our methods, improving the performance data collector and precision of pinpointing the actual problem source to prepare solution that could be effectively used in broad range of projects. Furthermore, we plan to evaluate our solution on existing projects and potentially detect real performance bugs.

Acknowledgements

We thank for the support received from Red Hat company and H2020 ECSEL project Aquas. We thank our supervisors and colleagues from VeriFIT performance team — Tomáš Fiedor, Hana Pluháčková, Adam Rogalewicz and Tomáš Vojnar.

References

[1] *Perun: Performance Version System.* https://github.com/tfiedor/perun. [Online; visited 2.4.2017].

[2] Podola Radim and Pavela Jiří. *Profilace bez komplikace: Profilovací knihovny pro programy v C/C++.* In *Excel@FIT'17.*

[3] Jiří Pavela. *Knihovna pro profilování datových struktur programů C/C++.* Bachelor's thesis, BUT, FIT, 2017.

[4] Radim Podola. *Knihovna pro profilování a vizualizaci spotřeby paměti programů C/C++.* Bachelor's thesis, BUT, FIT, 2017.

[5] Sokratis Tsakiltsidis, Andriy Miranskyy, and Elie Mazzawi. *Towards Automated Performance Bug Identification in Python. ISSREW'16.*

[6] Adrian Nistor. *Understanding, detecting and repairing performance bugs.* PhD thesis, University of Illinois, 2014.

[7] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. *Understanding and detecting real-world performance bugs. In Proc. of PLDI'18*, pages 77 – 88, 2012.

[8] *How to automate load testing.* http://blog.loadimpact.com/how-to-automate-load-testing. [Online; visited 2.4.2017].

[9] *Performance automation.* https://blackfire.io/docs/24-days/16-performance-automation. [Online; visited 2.4.2017].

[10] *Perf: Linux profiling with performance counters.* https://perf.wiki.kernel.org/index.php/Main_Page. [Online; visited 2.4.2017].

[11] *Kernel Probes.* https://www.kernel.org/doc/Documentation/kprobes.txt. [Online; visited 2.4.2017].

[12] *SystemTap.* https://sourceware.org/systemtap/. [Online; visited 2.4.2017].