BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

# AUTOMATIC DETECTION OF PERFORMANCE DEGRADATION

AUTOMATICKÁ DETEKCE DEGRADACE VÝKONU

PROJECT PRACTICE
PROJEKTOVÁ PRAXE

AUTHOR ŠIMON STUPINSKÝ
AUTOR PRÁCE

SUPERVISOR Doc. Mgr. ADAM ROGALEWICZ, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2018

# Contents

**1 Introduction** **2**

**2 Automatic Detection of Degradation** **3**

**3 Collecting and Postprocessing the Data** **5**

3.1 Collecting the Performance Data . . . . . 5

3.2 Clusterization and Regression Analysis . . . . . 6

**4 Detection of Changes** **7**

4.1 Algorithm of Detection . . . . . 7

4.2 Rate detection using regression analysis . . . . . 8

4.3 Rate detection using coefficients . . . . . 8

4.4 Rate detection using polynomials . . . . . 10

4.5 Close Evaluation . . . . . 10

**5 Experimental Evaluation** **12**

**6 Conclusion** **14**

**Bibliography** **15**

# Chapter 1

# Introduction

Current tools for project performance analysis focus mostly on detecting selected performance bugs in the source code. 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.

This paper builds on our previous work [1], which describes methods for detection of the performance change between two profiles. There we have demonstrated individual types of errors injected on selected types of models and their detection on series of experiments. This work focuses on the precise formalization of those algorithms for automatic detection of performance changes based on the explored statistical and mathematical methods and its implementation. This is an extension of our preliminary results published in [2].

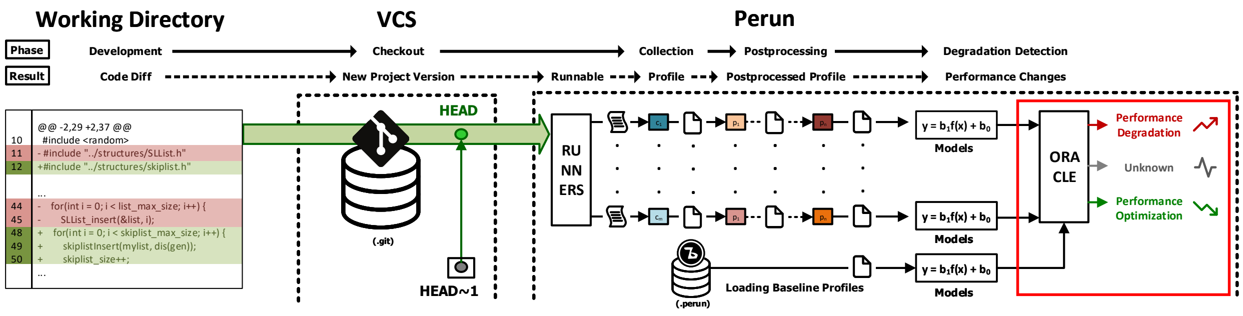
Moreover, by integrating these methods into the existing performance management system [3], we obtain a powerful difference analysis. Now, whenever we release a new version of our project, we can compare the newly computed regression models (see Sec. 3.2), the so-called *target profile*, with stored best models of previous stable version, the so-called *baseline profiles*, and report a list of performance changes. The details of the process of change detection also provides the following three crucial aspects of each detected change — the precise location, the severity and the confidence — which will be introduced in Chapter 4.

We evaluated our solution on series of artificial examples (see Ch. 5) 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 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.

# Chapter 2

# Automatic Detection of Degradation

We will first outline our framework for automated detection of performance changes on git repositories[1] as pictured in Figure 2.1. However, note that this framework was proposed and developed together with the main coordinator of the Perun project [3]. Whenever the developer wants to release the new version of the project the following steps are executed in order to automatically detect performance changes (note that this report focuses on solving of the points **5** and **6**):



```
@@ -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 2.1: Overview of the automatic detection method of performance changes over project repository. This paper focuses mainly on the degradation detection [3].

1. The 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 the new build and we use our *trace collector* (see Sec. 3.1) over this newly build binary and collect the raw performance data, in particular, the time consumption of functions.

3. These raw data are first clusterized (see Sec. 3.2) and so for each resource, we get its time consumption and classified cluster. For these resources, we estimate a regression performance model (see Sec. 3.2), which predicts the values of runtime for different values of clusters. We call these resulting profiles *target profiles.*

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

4. We retrieve the profiles of the parent versions from the persistent storage of Perun. These profiles serve as *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 4.

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).

Chapter 3

# Collecting and Postprocessing the Data

The input for our detection methods are two profiles, that represents the individual versions of the project. In the following sections we will briefly introduce the relevant methods for their collection. The first phase is the *collecting* the data by individual kind of the collector. This phase contains the steps 1 and 2 from the Itm. 2. After the collecting the data follows its processing by *clusterization* and by the *Regression Analysis*, these steps were mentioned in the steps 3 and 4 from the Itm. 2.

## 3.1 Collecting the Performance Data

After the user commits the new version of the project we first collect the performance data from the newly build project binary (or any other command or script). The actual profiling data collection is realized by injecting probes at specified code locations (currently we support only functions entries and exits).

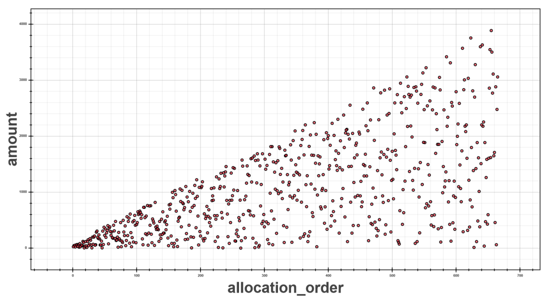


Figure 3.1: The example of the visualization of the collecting data by *memory collector*, which monitors the size of memory allocation at runtime of the program.

The **collector** (authored by Jiří Pavela) first loads the collection configuration, where the user specifies which functions will be probed in the input binary. According to this configuration, we generate the required script. In the next step, this script is translated, loaded into the kernel and the probed binary is executed. The probed profiling data are stored in the output trace log. The last phase of this process is loading the trace log and for each recorded timestamp is conversed into the actual time spent in each traced function call.

## 3.2 Clusterization and Regression Analysis

However, 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.

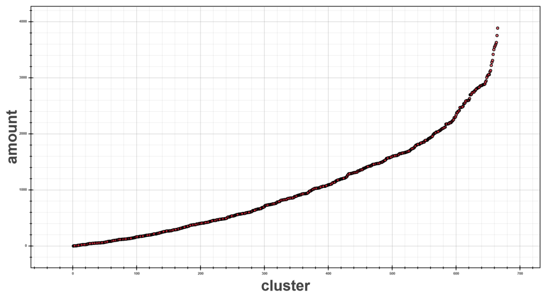


Figure 3.2: The example of visualization of the collecting data, which were processed by *clusterizer*.

The **clusterizer** (authored by Tomáš Fiedor) 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*. Both techniques are especially useful in conjunction with regression analysis as they allow us to derive the *independent* variable data for the collected resources and hence allow us to model performance.

**Regression Analysis** (authored by Jiří Pavela) is a method used in statistics to deduce the relationship between *dependent* and *independent* variables. 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 [4, 5].

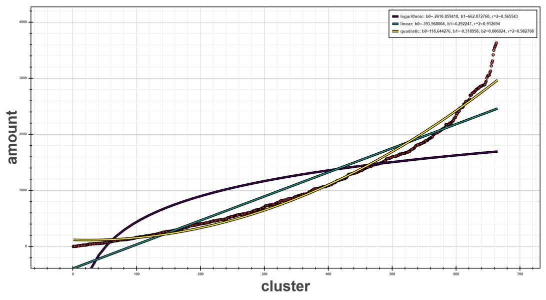


Figure 3.3: The example of the visualization of the collecting data, which were processed by *clusterizer* and subsequently were processed by *regression analysis*. In the figure you can see the individual models, which translate the collecting data.

# Chapter 4

# 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 newly released version of the 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 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[1] ($R^2$) 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.

## 4.1 Algorithm of Detection

The base algorithm for detection of performance changes is depicted in Algorithm 4. We propose two main variants (see Sec. 4.3, 4.4) and one variant for quick control of changes (see Sec. 4.2), 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 all 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 $m_b$ and $m_t$ respectively)as well as its corresponding linear models (denoted 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 the 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 $m_e$.

[1] https://en.wikipedia.org/wiki/Coefficient_of_determination

```
Method general_check($\mathcal{M}_b$, $\mathcal{M}_t$, $\mathcal{U}_m$)
    foreach $(m_b, m_t), (m_b^n, m_t^n) \in \mathcal{M}_b, \mathcal{M}_t$ do
        $\mathcal{M} = \{m_b, m_t, m_b^n, m_t^n\}$
        $dp_{t-b} = m_t - m_b$
        $dp_{t-b}^n = m_b^n - m_t^n$
        $\varepsilon_{rel} = \sum dp_{t-b} \ / \ |dp_{t-b}|$
        if NoChange $(m_b, m_t)$ then
            $\mathcal{M}$.add($\theta$)
        else
            if $\mathcal{U}_m == \mathcal{P}_n$ then
                PolyRegression ($\mathcal{M}$, $dp_{t-b}^n$)
            else if $\mathcal{U}_m == \mathcal{N}$ then
                LinRegression ($\mathcal{M}$, $dp_{t-b}^n$)
            else if $\mathcal{U}_m == \mathcal{R}_n$ then
                FindModels ($\mathcal{M}$, $dp_{t-b}$)
        end
        Evaluate($\mathcal{M}$, $\sum dp_{t-b}$, $\varepsilon_{rel}$, $\Delta$)
```

**Algorithm 1:** General algorithm of detection and classification performance changes based on input baseline ($\mathcal{M}_b$) and target ($\mathcal{M}_t$). We denote models by $m$, data points by $dp$ and relative error by $\varepsilon_{rel}$. The third input of the algorithm ($\mathcal{U}_m$) represents the individual modes of methods of detection performance changes. The specific modes are denoted as follows: *Polynomial Regression* ($\mathcal{P}_n$), *Linear Regression* ($\mathcal{N}$) and *Regression Analysis* implemented in Perun ($\mathcal{R}_n$).

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. The results obtained from all classification methods are then given to the concrete evaluation function, which gradually evaluates the mentioned aspects of performance changes, based on the supplied data.

## 4.2 Rate detection using regression analysis

This simple method is built on the subtraction of best-fit models and subsequently interleaving these data by newer models. We use regression analysis (see Section 3.2) to obtain a set of models for these subtracted data points. The analysis by this process does not serve the users for obtaining more accurate information about performance changes. Its main objective is to gain insight into the possibility of performance changes between individual profiles. The users can use two basic methods for the obtaining the more accurate information (see Sec. 4.3, 4.4).

## 4.3 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 the 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 the 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 $b_1$).

```
Method LinRegression($\mathcal{M}$, $dp^n_{t-b}$)
    $b_0, b_1, r$ = Stats.Linregress($\mathcal{M}_{m_b}$, $dp^n_{t-b}$)
    if $\xi \leq b_1 \leq \xi \wedge \xi * b_0 \geq \Delta_{b0} \leq \xi * b_0$ then
        $\Delta = \mathcal{C}$
    else if $|\xi * b_1| \geq \Delta^n_{b1} \leq |\xi * b_1| \wedge r^2 > \xi$ then
        $\Delta = \mathcal{N}$
    $\mathcal{E}$ = FindModels($dp^n_{t-b}$)
    if $\mathcal{E}_{n^2}(R^2) > \xi \wedge \mathcal{E}_{n^2}(R^2) > \mathcal{E}_n(R^2)$ then
        $\Delta = \mathcal{N}^2$
    if $\Delta == \emptyset$ then
        $\Delta$ = FindBestModel($\mathcal{E}$)
    $\mathcal{M}$.add($\Delta$)
```

**Algorithm 2:** The algorithm of classification performance changes based on *Linear Regression*. The bases of the algorithm are *linear* models, which are part of the set of models ($\mathcal{M}$). The analysis is performed on the basis of the obtained parameters from the library method `Linregress` from the package `scipy` and its subpackage `stats`[3]. We denote *gradient* by $\nabla$, *intercept* by $\mathfrak{y}$ and *r_value* by $\mathfrak{r}$. The individual performance changes we denote as follows: *Constant* ($\mathcal{C}$), *Linear* ($\mathcal{N}$) and *Quadratic* ($\mathcal{N}^2$). The set of models which represented the profile constructed from the linear standard error is denoted by $\mathcal{E}$.

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

The first change, which is checked is the **constant** change. The first assumption for this change is, that the value of the **slope** is approximately the same as the value of the original slope or within the range of the given threshold. Another assumption is, that the value of the difference of intercepts of *baseline* and *target* profiles is within the acceptable range of **intercept** of the change profile. Simply put, the model was shifted and its slope was not changed. In this case, we classify the change as **constant**. Let us have model $y = 42x + 5.2$ for the error, then considering slope $b_1$ is 42 and intercept 5.2, this algorithm will classify this as **constant** error.

In another case, the situation is somewhat opposite, concretely in the check of **linear** change. Is assumed that the value of **intercept** did not change too much and that the value of the difference of *slopes* of the linear model from both compared profiles is within in the acceptable range of the value of the slope of the change profile. In this case one more condition must be fulfilled, namely that the value of **coefficient of determination** of the change profile must be must be sufficiently high. Under these assumptions, the change is subsequently classified as the **linear** change. Let us have model $y = 5.2x+42$ for the change model, then considering slope $b_1$ is 5.2 and intercept $b_0$ 42, this algorithm will classify this as **linear** change.

In the last concretely check it is **quadratic** change. There must be fulfilled one assumption and it, that the value of the **coefficient of determination** of the *quadratic* model must be higher as the **coefficient of determination** of the *linear* model and must be higher as the given *threshold*. Then we classify the change as **quadratic**. Described assumptions mean, that can occur the situation when the $R^2$ of the *quadratic* model will be higher than $R^2$ of the *linear* model but if its value will be lower than the value of the *threshold*, the change not be classified such as *quadratic*.

## 4.4 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 $n$th 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 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.

```
Method PolyRegression($\mathcal{M}$, $dp^n_{t-b}$)
    $\mathcal{M}$.add(FindPoly($dp^n_{t-b}$), 1)
    for $deg = 0$ to MAX_DEG do
        $\rho$ = FindPoly($dp^n_{t-b}$, $deg$)
        if $\rho[residual] \leq \xi$ then
            break
        $\mathcal{M}$.add($\rho$)
```

**Algorithm 3:** The algorithm of classification performance changes based on *Polynomial Regression*. The bases of the algorithm are *linear models*, which are part of the set of models ($\mathcal{M}$). The threshold to compare of the suitability of interleaved polynomial we denote as $\xi$ and *linear absolute error* computed from the mentioned linear models we denote as $dp^n_{t-b}$.

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 this method we iteratively find such polynomial of smallest degree, that can interleave the data precisely. We use the `numpy`[4] 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 *residual*, 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 $\xi$, we break from the loop and stop.

## 4.5 Close Evaluation

The final phase in the evaluation of the performance changes is the processing of the received data. The input of this method is set of models, which contains the individual models above which were performed detection and classification. Another input parameter is the *absolute error*, whose value serves for resolution degradation and optimization. Likewise, the input

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

and the last parameter of this method is the *relative error*, which decides whether to given performance change is sufficiently large or not.

```
Method Evaluation($\mathcal{M}$, $abs\_sum$, $\varepsilon_{rel}$, $\Delta$)
    if $\Delta$ != $\emptyset$ then
        if $abs\_sum > 0$ then
            if $\varepsilon_{rel} > \xi$ then
                $\delta = \downarrow_{deg}$
            else
                $\delta = \swarrow_{maydeg}$
            end
        else
            if $\varepsilon_{rel} < -\xi$ then
                $\delta = \uparrow_{opt}$
            else
                $\delta = \nearrow_{mayopt}$
            end
        end
    yield $\mathcal{M}_{m_b} \rightarrow \mathcal{M}_{m_t}$, $\Delta$, $\delta$, Min($\mathcal{M}_n R^2$), $\varepsilon_{rel}$
```

**Algorithm 4:** The evaluation algorithm for the final steps in the detection, based on previously acquired data. We denote *set of models* by $\mathcal{M}$, *absolute error* by *abs_sum* and *relative error* by $\varepsilon_{rel}$. $\Delta$ represents the kind of the performance change and $\delta$ represents whether it is *degradation*, *optimization* or there have been no changes occurred. Each such type of change is divided into two levels, that depends on the value of *relative error*. In the case that its value is less as the threshold (respectively greater) we denote change as *maybe degradation* ($\swarrow_{maydeg}$), respectively maybe optimization ($\nearrow_{mayopt}$).

# Chapter 5

# Experimental Evaluation

We tested both main methods on the set of artificial examples consisting of profiles with selected types of models (constant, linear, logarithmic, quadratic, exponential and power) and profiles with injected changes, in particular, profiles injected with constant, linear and quadratic changes. The individual injected changes represent artificial loops that have been embedded in testing examples. Moreover, we tested each type of the models to both favorable (optimization) and unfavorable (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 5.2 shows the results of our experimental evaluation. We denote with colors 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 than in reality. Table 5.1 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 5.1: 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 the logarithmic model. Hence, the shift of this type of model is not as obvious and detectable as opposed to the rest of models.

The second method based on polynomials was, in this case, more successful and has proven all degradation marks with the 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 the independent variable, had similar results, with the only problem recorded for power models. The quadratic change was correctly detected by all changes, however, it showed the highest misclassification rate. We believe this is caused by a small value of injected change, which causes this degradation. Thus, the *quadratic change* was not large enough to be classified as *quadratic*, but it was always classified by lower degree of change, concretely such as *linear* change.

| #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 5.2: 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.

# Chapter 6

# Conclusion

In this paper, we briefly introduced a framework for long-term and continuous performance change monitoring during the project development. However, the main task of this paper was to introduce the methods for detection of performance changes. Specifically, were introduced three methods, which mainly differ in the actual classification of changes according to the computed data. All methods were discussed in previous subsections and have been referred to its use, advantages, and disadvantages.

We use the *version difference analysis* (integrated into performance profiling platform Perun [3]) 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 these methods, improving the performance data collector and precision of pinpointing the actual problem source to prepare the 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.

# Bibliography

[1] Stupinský Šimon. *Automatic Detection of Performance Degradation.* Project practice, BUT, FIT, 2018.

[2] Pavela Jiří and Stupinský Šimon. *Towards the detection of performance degradation.* In *Excel@FIT'18*.

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

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

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