Learning from Source Code History to Identify Performance Failures

Juan Pablo Sandoval Alcocer
PLEIAD Lab
DCC, University of Chile
jsandova@dcc.uchile.cl

Alexandre Bergel
PLEIAD Lab
DCC, University of Chile
abergel@dcc.uchile.cl

Marco Tulio Valente
Federal University of Minas Gerais, Brazil
mtov@dcc.ufmg.br

ABSTRACT

Source code changes may inadvertently introduce performance regressions. Benchmarking each software version is traditionally employed to identify performance regressions. Although effective, this exhaustive approach is hard to carry out in practice. This paper contrasts source code changes against performance variations. By analyzing 1,288 software versions from 17 open source projects, we identified 10 source code changes leading to a performance variation (improvement or regression). We have produced a cost model to infer whether a software commit introduces a performance variation by analyzing the source code and sampling the execution of a few versions. By profiling the execution of only 17% of the versions, our model is able to identify 83% of the performance regressions greater than 5% and 100% of the regressions greater than 50%.

Keywords

Performance variation; performance analysis; performance evolution

1. INTRODUCTION

Software evolution refers to the dynamic change of characteristics and behavior of the software over time [17]. These progressive changes may negatively decrease the quality of the software and increase its complexity [3, 15]. Such deterioration may also affect the application performance over time [20]. Testing software continuously helps detect possible issues caused by source code changes [2, 8].

Diverse approaches have been proposed to detect performance regressions along software evolution [5, 18, 22, 26]. The most commonly employed technique is exhaustively executing all benchmarks over all versions: comparing the performance metrics of the recently released version with the previous ones are then used to spot performance variations [5, 11]. However, such approaches are highly time consuming because benchmarks can take days to execute [12]. Furthermore, there are a number of factors (*e.g.,* garbage collection, JIT compiler) that can affect the measurements and benchmarks need to be executed multiple times to reduce the measurement bias [22]. For this reason, testing software performance periodically (*e.g.,* daily or per release basis) is an expensive task. It has been shown that by identifying the relations between source code changes and performance variations, it is possible to estimate whether a new software version introduces a performance regression or not; without executing the benchmarks [12].

Existing research [12, 13, 24, 30] predominantly categorizes recurrent performance bugs and fixes by analyzing a random sample of performance bug reports. These studies voluntary ignore performance related issues that are not reported as a bug or bug fix. Therefore, in this paper, we aim to bridge this gap by conducting a comprehensive study of real-world performance variations detected by analyzing the performance evolution of 17 open source projects along 1,288 software versions. The two research questions addressed in this study are:

- RQ1 – *Are performance variations mostly caused by modifications of the same methods?* This question is particularly critical to understanding what performance variation stems from. Consider a method m that causes a performance regression when it is modified. It is likely that modifying m once more will impact the performance. Measuring the proportion of such "risky" methods is relevant for statically predicting the impact a code revision may have.

- RQ2 – *What are the recurrent source code changes that affect performance along software evolution?* More precisely, we are interested in determining which source code changes mostly affect program performance along software evolution and in which context. If performance variations actually do match identified source code changes, then it is posible to judge the impact of a given source code change on performance.

Findings. Our experiments reveal a number of facts for the source code changes that affect the performance of the 17 open source systems we analyzed:

- Most performance variations are caused by source code changes made in different methods. Therefore, keeping track of methods that participated in previous performance variations is not a good option to detect performance variations.

ICPE'16, March 12-18, 2016, Delft, Netherlands
 ISBN 978-1-4503-4080-9/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851553.2851571

- Most source code changes that cause a performance variation are directly related to method call addition, deletion or swap.

Based on the result of our study, we propose *horizontal profiling*, a sampling technique to statically identify versions that may introduce a performance regression. It collects run-time metrics periodically (*e.g.,* every k versions) and uses these metrics to analyze the impact of each software version on performance. *Horizontal profiling* assigns a cost to each source code change based on the run-time history. The goal of *horizontal profiling* is to reduce the performance testing overhead, by benchmarking just software versions that contain costly source code changes. Assessing the accuracy of horizontal profiling leads to the third research question:

- RQ3 – *How well can horizontal profiling prioritize the software versions and reduce the performance testing overhead?* This question is relevant since the goal of *horizontal profiling* is to reduce the performance regression testing overhead by only benchmarking designated versions. We are interested in measuring the balance between the overhead of exercising *horizontal profiling* and the accuracy of the prioritization.

We evaluate our technique over 1,125 software versions. By profiling the execution of only 17% of the versions, our model is able to identify 83% of the performance regressions greater than 5% and 100% of the regressions greater than 50%. These figures are therefore comparable with the related work: Huang *et al.* [12] have proposed a static approach and identify 87% (without using program slicing) of regression with 14% of software versions. However, by using a dedicated profiling technique, our cost model does not require painful manual tuning, and it performs well, independently of the performance regression threshold. Moreover, our hybrid technique (static and dynamic) is applicable to a dynamically typed and object-oriented programming languages.

Outline. Section 2 describes the projects under study and the benchmarks used to detect performance variations. Section 3 contrasts source code changes with the performance variations. Section 4 presents and evaluates the cost model based on the run-time history. Section 5 discusses threats to validity we face and how we are addressing them. Section 6 overviews related work. Section 7 concludes and presents an overview of our future work.

2. EXPERIMENTAL SETUP

2.1 Project under Study

We conduct our study around the Pharo programming language[1]. Our decision is motivated by a number of factors: First, Pharo offers an extended and flexible reflective API, which is essential to iteratively execute benchmarks over multiple application versions and executions. Second, application instrumentation and monitoring its execution are also cheap and with a low overhead. Third, the computational model of Pharo is uniform and very simple, which means that applications for which we have no knowledge are easy to download, compile and execute.

[1]http://pharo.org

Table 1: Projects under Study

Project	Versions	LOC	Classes	Methods
Morphic	214	41,404	285	7,385
Spec	270	10,863	404	3,981
Nautilus	214	11,077	173	2012
Mondrian	145	12,149	245	2,103
Roassal	150	6,347	227	1,690
Rubric	83	10,043	173	2,896
Zinc	21	6,547	149	1,606
GraphET	82	1,094	51	464
NeoCSV	10	8,093	9	125
XMLSupport	22	3,273	118	1,699
Regex	13	4,060	39	309
Shout	16	2,276	18	320
PetitParser	7	2,011	63	578
XPath	10	1,367	93	813
GTInspector	17	665	17	128
Soup	6	1,606	26	280
NeoJSON	8	700	16	139
Total	**1,288**	**130,386**	**2,106**	**26,528**

We pick 1,288 release versions of 17 software projects from the Pharo ecosystem stored on the Pharo forges (*SqueakSource*[2], *SqueakSource3*[3] and *SmalltakHub*[4]). The set of considered project have a broad range of application: user interface frameworks (Morphic and Spec), a source code highlighter (Shout), visualization engines (Roassal and Mondrian), a HTTP networking tool (Zinc), parsers (PetitParser, NeoCSV, XMLSupport, XPath, NeoJSON and Soup), a chart builder (GraphET), a regular expression checker (Regex), an object inspector (GTInspector) and code browsers and editors (Nautilus and Rubric).

Table 1 summarizes each one of these projects and gives the number of defined classes and methods along software evolution. It also shows the average lines of code (LOC) per project.

These applications have been selected for our study for a number of reasons: (i) they are actively supported and represent relevant assets for the Pharo community. (ii) The community is friendly and interested in collaborating with researchers. As a result, developers are accessible in answering our questions about their projects.

2.2 Source Code Changes

Before reviewing variation of performance, we analyze how source code changes are distributed along all the methods of each software project. Such analysis is important to contrast performance evolution later on.

Let M be the number of times that a method is modified along software versions of each software project. Figure 1 gives the distribution of variable M of all projects under study. The y-axis is the percentage of methods, and x-axis is the number of modifications. One method has been modified 14 times. In total, 83% of the methods are simply defined without being modified in subsequent versions of the application ($M = 0$).

There are 2,846 methods (11%) modified only once ($M = 1$) in the analyzed versions. Only 6% of the methods are modified more than once ($M > 1$). Table 2 gives the number of methods that: i) are not modified ($M = 0$), ii) are modified only once ($M = 1$), and iii) are modified more than once ($M > 1$) for each software project. We have found that in all

[2]http://www.squeaksource.com/
[3]http://ss3.gemstone.com/
[4]http://smalltalkhub.com/

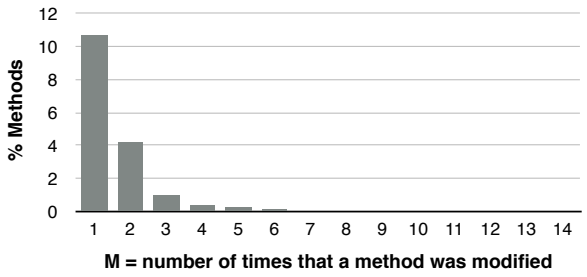

Figure 1: Source Code Changes histogram at method level

but one project, the number of methods that are modified more than once are relatively small compared to the number of methods that are modified once. The Mondrian project is clearly an outlier since 28% of its methods are modified twice or more. A discussion with the authors of Mondrian reveals the application went through long and laborious maintenance phases on a reduced set of particular classes.

Table 2: M = number of times that a method is modified

Project	Methods	M = 0	M = 1	M >1
Morphic	7,385	6,810 (92%)	474 (6%)	101 (1%)
Spec	3,981	2,888 (73%)	730 (18%)	363 (9%)
Rubric	2,896	2,413 (83%)	362 (13%)	121 (4%)
Mondrian	2,103	1,361 (65%)	146 (7%)	596 (28%)
Nautilus	2,012	1,646 (82%)	248 (12%)	118 (6%)
XMLSupport	1,699	1,293 (76%)	276 (16%)	130 (8%)
Roassal	1,690	1,379 (82%)	232 (14%)	79 (5%)
Zinc	1,606	1,431 (89%)	139 (9%)	36 (2%)
XPath	813	780 (96%)	33 (4%)	0 (0%)
PetitParser	578	505 (87%)	66 (11%)	7 (1%)
GraphET	464	354 (76%)	70 (15%)	40 (9%)
Shout	320	304 (95%)	12 (4%)	4 (1%)
Regex	309	303 (98%)	5 (2%)	1 (0%)
Soup	280	269 (96%)	11 (4%)	0 (0%)
NeoJSON	139	131 (94%)	7 (5%)	1 (1%)
GTInspector	128	119 (93%)	0 (0%)	9 (7%)
NeoCSV	125	84 (67%)	35 (28%)	6 (5%)
Total	**26,528**	**22,070 (83%)**	**2,846 (11%)**	**1,612 (6%)**

Similarly, we analyzed the occurrence of class modification: 59% of the classes remain unmodified after their creation, 14% of the classes are modified once (*i.e.,* at least one method has been modified), and 27% of the classes are modified more than once.

2.3 Benchmarks

In order to get reliable and repeatable execution footprints, we select a number of benchmarks for each considered application. Each benchmark represents a representative execution scenario that we will carefully measure. Several of the applications already come with a set of benchmarks. If no benchmarks were available, we directly contacted the authors and they kindly provided benchmarks for us. Since these benchmarks have been written by the authors, they are likely to cover part of the application for which its performance is crucial.

At that stage, some benchmarks have to be worked or adapted to make them runnable on a great portion of each application history. The benchmarks we considered are therefore generic and do not directly involve features that have been recently introduced. Identifying the set of benchmarks runnable over numerous software versions is particularly time consuming since we had to test each benchmark over a sequence of try-fix-repeat. We have 39 executable benchmarks runnable over a large portion of the versions.

All the application versions and the metrics associated to the benchmarks are available online[5].

3. UNDERSTANDING PERFORMANCE VARIATIONS OF MODIFIED METHODS

A software commit may introduce a scattered source code change, spread over a number of methods and classes. We found 4,458 method modifications among 1,288 analyzed software versions. Each software version introduces 3.46 method modifications on average. As a consequence, a performance variation may be caused by multiple method source code changes within the same commit.

3.1 Performance Variations of Modified Methods

We carefully conducted a quantitative study about source code changes that directly affect the method performance. Let V be the number of times that a method is modified and becomes slower or faster after the modification. We consider that the execution time of a method varies if the absolute value of the variation of the accumulated execution time between two consecutive versions of the method is greater than a threshold. In our situation, we consider *threshold = 5%* over the total execution time of the benchmark. Below 5%, it appears that the variations may be due to technical consideration, such as inaccuracy of the profiler [4].

Figure 2 gives the distribution of V for all methods of the projects under study. In total, we found 150 method modifications where the modified method becomes slower or faster. These modifications are made over 111 methods; 91 methods are modified only once ($V = 1$) and 20 more than once ($V > 1$). Table 3 gives the number of methods for each software project.

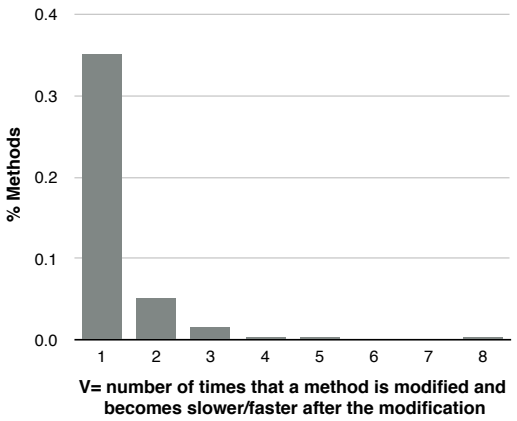

Figure 2: Performance Variations of Modified Methods (threshold = 5%), 111 methods are here reported.

[5]http://users.dcc.uchile.cl/~jsandova/hydra/

Table 3: V= number of times that a method is modified and becomes slower/faster after the modification. (threshold = 5%)

Project	Methods	V = 0	V = 1	V >1
Morphic	7,385	7,382 (100%)	2 (0%)	1 (0%)
Spec	3,981	3,944 (99%)	24 (1%)	13 (0%)
Rubric	2,896	2,896 (100%)	0 (0%)	0 (0%)
Mondrian	2,103	2,091 (99%)	11 (1%)	1 (0%)
Nautilus	2,012	2,008 (100%)	4 (0%)	0 (0%)
XMLSupport	1,699	1,689 (99%)	10 (1%)	0 (0%)
Roassal	1,690	1,675 (99%)	14 (1%)	1 (0%)
Zinc	1,606	1,597 (99%)	7 (0%)	2 (0%)
XPath	813	813 (100%)	0 (0%)	0 (0%)
PetitParser	578	566 (98%)	12 (2%)	0 (0%)
GraphET	464	459 (99%)	3 (1%)	2 (0%)
Shout	320	320 (100%)	0 (0%)	0 (0%)
Regex	309	309 (100%)	0 (0%)	0 (0%)
Soup	280	280 (100%)	0 (0%)	0 (0%)
NeoJSON	139	138 (99%)	1 (1%)	0 (0%)
GTInspector	128	128 (100%)	0 (0%)	0 (0%)
NeoCSV	125	119 (95%)	5 (4%)	1 (1%)
Total	**26,528**	**26,417(99.6%)**	**91(0.33%)**	**20(0.07%)**

False Positive. However, not all these 150 modifications are related to the method performance variations because there are a number of false-positives. Consider the change made in the open method on the class ROMondrianViewBuilder:

```
ROMondrianViewBuilder>>open
   | whiteBox realView |
   self applyLayout.
   self populateMenuOn: viewStack.
−  ^ stack open
+  ^ viewStack open
```

This modification is only a variable renaming: the variable stack has been renamed into viewStack. Our measurement indicates that this method is now slower, which is odd since a variable renaming should not be the culprit of a performance variation. A deeper look at the method called by open reveals that the method applyLayout is also slower. Therefore, we conclude that open is slower because of a slower dependent method, and not because of its modification. Such a method is a false positive and its code modification should not be considered as the cause of the performance variation.

Example code with a leading "-" is from the previous version, while code with a leading "+" is in the current version. Unmarked code (without a leading "-" or "+") is in both versions.

Manually Cleaning the Data. We manually revised the 150 method variations by comparing the call-graph (obtained during the execution) and the source code modification. We then manually revised the source code (as we just did with the method open). In total, we found 66 method modifications (44%) that are not related with the method performance variation. The remaining 84 method modifications (56%) cause a performance variation in the modified method. These modifications are distributed along 11 projects; table 4 gives the distribution by project.

Summary. *Are performance variations mostly caused by modifications of the same methods?* We found that 84 method modifications that cause a performance variation (regression or improvement) were done over 67 methods, which means 1.25 modifications per method. Table 4 shows the ratio between method modifications and methods is less than two in all projects. In addition, we found that the these methods were modified a number of times along source code evolution without causing a performance variation.

Table 4: Method modifications that affect method performance (R= regression, I= improvement, R/I = regression in some benchmarks and Improvement in others)

Project	Method Modifications				Involved	Mod. by
	R	I	R/I	Total	Methods	Method
Spec	19	9	0	28	16	1.75
Roassal	7	5	0	12	11	1.09
Zinc	2	1	4	7	7	1.00
Mondrian	5	3	0	8	7	1.14
XMLSupport	6	0	0	6	6	1.00
GraphET	4	3	0	7	5	1.4
NeoCSV	0	5	0	5	5	1.00
PetitParser	5	0	0	5	5	1.00
Morphic	2	1	0	3	2	1.50
Nautilus	2	0	0	2	2	1.00
NeoJSON	0	1	0	1	1	1.00
Total	52	28	4	84	67	1.25

> Most performance variations were caused by source code changes made in different methods. Therefore, keeping track of methods that participated in previous performance variations is not a good option to detect performance variations.

3.2 Understanding the Root of Performance Regressions

Accurately identifying the root of a performance regression is difficult. We investigate this by surveying authors of method modifications causing a regression. From the 84 method modifications mentioned in Section 3.1, we obtained author feedback for 21 of them. Each of 21 method modifications is the cause of a regression greater than 5%. We also provided the benchmarks to the authors since it may be that the authors causing a regression are not aware of the application benchmarks. These methods are spread over four projects (Roassal, Mondrian, GraphET, and PetitParser). Each author was contacted by email and we discussed about the method modification causing a regression.

For 6 (29%) of these 21 modifications, the authors were aware of the regression at the time of the modification. The authors therefore consciously and intentionally made the method slower by adding or improving functionalities. We also asked them whether the regression could be avoided while preserving the functionalities. They answered that they could not immediately see an alternative to avoid or reduce the performance regression.

For 5 (24%) of the modifications, authors did not know that their new method revision caused a performance regression. However, authors acknowledged the regressions and were able to propose an alternative method revision that partially or completely removes the regression.

For the 10 remaining modifications, author did not know that they caused a performance regression and no alternative could be proposed to improve the situation.

This is a preliminary result and we can not draw any strong conclusion from only 21 method modifications. However, this small and informal survey of practitioners indicates that a significant number of performance regressions are apparently inevitable. On the other hand, such incertitude expressed

by the authors regarding the presence of a regression and providing change alternative highlights the relevance of our study and research effort.

3.3 Categorizing Source Code Changes That Affect Method Performance

This section analyzes the cause of all source code changes that affect method performance. We manually inspected the method source code changes and the corresponding performance variation. We then classify the source code changes into different categories based on the abstract syntax tree modifications and the context in which the change is used. In our study, we consider only code changes that are the culprits for performance variation (regression or improvement), ignoring the other non-related source code changes.

Subsequently, recurrent or significant source code changes are described. Each source code change has a title, a brief description, followed by one source code example taken from the examined projects.

Method Call Addition. This source code change adds expensive method calls that directly affect the method performance. This situation occurs 24 times (29%) in our set of 84 method modifications, all these modifications cause performance regressions. Consider the following example:

```
GETDiagramBuilder>>openIn: aROView
  self diagram displayIn: aROView.
  + self relocateView
```

The performance of openIn: dropped after having inserted the call to relocateView.

Method Call Swap. This source code change replaces a method call with another one. Such a new call may be either more or less expensive than the original call. This source change occurs 24 times (29%) in our set of 84 method modifications; where 15 of them cause a performance regression and 9 a performance improvement.

```
MOBoundedShape>>heightFor: anElement
  ^ anElement
  - cachedNamed: #cacheheightFor:
  - ifAbsentInitializeWith: [ self computeHeightFor:
    anElement ]
  + cacheNamed: #cacheheightFor:
  + of: self
  + ifAbsentInitializeWith: [ self computeHeightFor:
    anElement ]
```

The performance of heightFor: dropped after having swapped the call to cacheNamed:ifAbsentInitializeWith by cacheNamed: of:ifAbsentInitializeWith.

Method Call Deletion. This source code change deletes expensive method calls in the method definition. This pattern occurs 14 times (17%) in our set of 84 method modifications - all these modifications cause performance improvements.

```
MOGraphElement>>resetMetricCaches
  - self removeAttributesMatching: "cache*"
  + cache := nil.
```

This code change follows the intuition that removing a method call makes the application faster.

Complete Method Change. This category groups the source code changes that cannot be categorized in one of these situations, because there are many changes in the method that contribute to the performance variation (*i.e.,* a combination of method call additions and swaps). We have seen 9 complete method rewrites (11%) among the 84 considered method modifications.

Loop Addition. This source code change adds a loop (*i.e.,* while, for) and a number of method calls that are frequently executed inside the loop. We have seen 5 occurrences of this pattern (6%) - all of them cause a performance regression.

```
ROMondrianViewBuilder>>buildEdgeFrom:to:for:
  | edge |
  edge := (ROEdge on: anObject from: fromNode to:
    toNode) + shape.
+ selfDefinedInteraction do: [:int | int value: edge ].
  ^ edge
```

Change Object Field Value. This source code change sets a new value in an object field causing performance variations in the methods that depend on that field. This pattern occurs 2 times in the whole set of method modifications have analyzed.

```
GETVerticalBarDiagram>>getElementsFromModels
  ^ rawElements with: self models do: [ :ele :model |
      + ele height: (barHeight abs).
      count := count + 1].
```

On this example, the method height: is a variable accessor for the variable height defined on the object ele.

Conditional Block Addition. This source code change adds a condition and a set of instructions. These instructions are executed upon the condition. This pattern occurs 2 times in the whole set of method modifications we analyzed. Both of them cause a performance improvement.

```
ZnHeaders>>normalizeHeaderKey:
+ (CommonHeaders includes: string) ifTrue: [ ^ string ].
  ^ (ZnUtils isCapitalizedString: string)
    ifTrue: [ string ]
    ifFalse: [ ZnUtils capitalizeString: string ]
```

Changing Condition Expression. This source code change modifies the condition of a conditional statement. This change could introduce a variation by changing the method control flow and/or the evaluation of the new condition expression is faster/slower. This pattern occurs 2 times in the whole set of method modifications we have analyzed.

```
NeoCSVWriter>>writeQuotedField:
  | string |
  string := object asString.
  writeStream nextPut: $".
  string do: [ :each |
-   each = $"
+   each == $"
      ifTrue: [ writeStream nextPut: $"; nextPut: $" ]
      ifFalse: [ writeStream nextPut: each ] ].
  writeStream nextPut: $"
```

The example above simply replaces the equal operation = by the identity comparison operator ==. The latter is significantly faster.

Change Method Call Scope. This source code change moves a method call from one scope to another executed more or less frequently. We found 1 occurrence of this situation

Table 5: Source code changes that affect method performance (R= Regression, I= Improvement, R/I = Regression in some benchmarks and Improvement in others)

	Source Code Changes	R	I	R/I	Total
1	Method call additions	23	0	1	24 (29%)
2	Method call swaps	15	9	0	24 (29%)
3	Method call deletion	0	14	0	14 (17%)
4	Complete method change	6	0	3	9 (11%)
5	Loop Addition	5	0	0	5 (6%)
6	Change object field value	2	0	0	2 (2%)
7	Conditional block addition	0	2	0	2 (2%)
8	Changing condition expression	0	2	0	2 (2%)
9	Change method call scope	1	0	0	1 (1%)
10	Changing method parameter	0	1	0	1 (1%)
	Total	52	28	4	84 (100%)

in the whole set of method modifications. Such a change resulted in a performance improvement.

```
GETCompositeDiagram>>transElements
  self elements do: [ :each | | trans actualX |
    + pixels := self getPixelsFromValue: each getValue.
    (each isBig)
      ifTrue: [ | pixels |
        − pixels := self getPixelsFromValue: each
    getValue.
        ...
      ifFalse: [ ^ self ].
    ...
    ]
```

Changing Method Parameter. The following situation changes the parameter of a method call. We found only 1 occurrence of this situation in the whole set of method modifications.

```
ROMondrianViewBuilder>>buildEdgeFrom:to:for:
  | edge |
  edge := (ROEdge on: anObject from: fromNode to:
    toNode) + shape.
−  selfDefinedInteraction do: [:int | int value: edge ].
+  selfDefinedInteraction do: [:int | int value: (Array with:
    edge) ].
  ^ edge'
```

Table 5 gives the frequency of each previously presented source code change.

Categorizing Method Calls. Since most changes that cause a performance variation (patterns 1,2,3) involve a method call. We categorize the method call additions, deletions and swaps (totaling 62) in three different subcategories:

- *Calls to external methods:* 10% of the method calls correspond to method of external projects (*i.e.,* dependent projects).
- *Calls to recently defined methods:* 39% of the method calls correspond to method that are defined in the same commit. For instance, a commit that defines a new method and adds method calls to this method.
- *Calls to existing project methods:* 51% of the method calls correspond to project methods that were defined in previous versions.

Summary. *RQ2: What are the most common types of source code changes that affect performance along software evolution?* We found, in total, that 73% of the source code changes that cause a performance variation are directly related to method call addition, deletion or swap (patterns 1,2,3). This percentage varies between 60% and 100% in all projects, with the only exception of the Zinc project that has a 29%; most Zinc performance variations were caused by complete method changes.

> Most source code changes that cause a performance variation are directly related to method call addition, deletion or swap.

3.4 Triggering a Performance Variation

To investigate whether a kind of change could impact the method performance we compare changes that caused a performance variation with those that do not cause a performance variation. For this analysis, we consider the source code changes: loop addition, method call addition, method call deletion and method call swap [6].

To fairly compare between changes that affect performance and changes that do not affect performance, we consider changes in methods that are executed by our benchmark set. Table 6 shows the number of times that a source code change was done along software versions of all projects (Total), and the number of times that a source code change cause a performance variation (Perf. Variation) greater than 5% over the total execution time of the benchmark.

Table 6: Comparison of source code changes that cause a variation with the changes that do not cause a variation (R= regression, I= improvement, R/I = regression in some benchmarks and Improvement in others)

Source Code Changes	Total	Perf. Variations			
		R	I	R/I	Total
Method call additions	231	23	0	1	24(10.39%)
Method call deletions	119	0	14	0	14(11.76%)
Method call swap	321	15	9	0	24 (7.48%)
Loop additions	8	5	0	0	5(62.5%)

Table 6 shows that these four source code changes are frequently done along source code evolution; however just a small number of instances of these changes cause a performance variation. After manually analyzing all changes that cause a variation, we conclude that there are mainly two factors that contribute to the performance variation:

- *Method call executions.* The number of times that a method call is executed plays an important role to determine if this change can cause a performance regression. We found that 92% of source code changes were made over a frequently executed source code section.
- *Method call cost.* The cost of a method call is important to determine the grade of performance variation. We found that 7 (8%) method calls additions/deletions were only executed once and cause a performance regression greater than 5%. In the other 92% the performance vary depending on how many times the method call is executed and the cost of each method call execution.

[6]These changes correspond the top-4 most common changes, with the exception of "Complete method change" which we did not consider in the analysis since it is not straightforward to detect this pattern automatically.

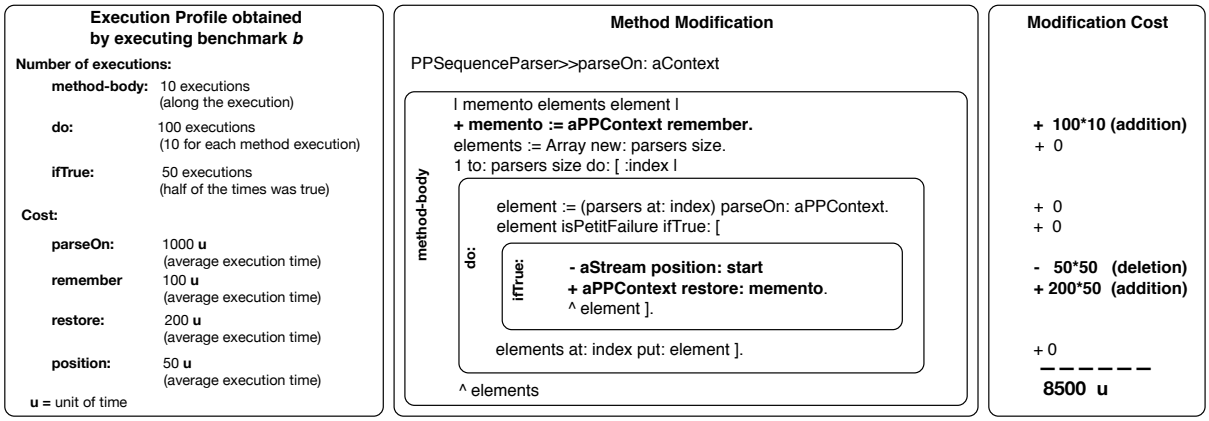

Figure 3: LITO cost model example

We believe these factors are good indicators to decide when a source code change could introduce performance variation. We support this assumption by using this criteria to detect performance regressions, as we describe in the following sections.

4. HORIZONTAL PROFILING

We define *horizontal profiling* as a technique to statically detect performance regressions based on benchmark execution history. The rationale behind *horizontal profiling* is that if a software execution becomes slow for a repeatedly identified situation (*e.g.,* particular method modification), then the situation can be exploited to reduce the performance regression testing overhead.

4.1 LITO: A Horizontal Profiler

We built LITO to (mostly) statically identify software versions that introduce a performance regressions. LITO takes as input (i) the source code of a software version V_n and (ii) the profile (obtained from a traditional code execution profiler) of the benchmarks execution on a previous software version V_m. LITO identifies source code changes in the analyzed software version V_n, and determines if that version is likely to introduce a performance regression or not.

The provided execution profile is obtained from a dedicated code execution profiler and is used to infer components dependencies and loop invariants. As discussed later on, LITO is particularly accurate even if V_m is a version distant from V_n.

Using our approach, practitioners prioritize the performance analysis in the selected versions by *LITO*, without the need to carry out costly benchmark executions for all versions. The gain here is significant since LITO helps identify software commits that may or may not introduce a performance variation.

Execution Profile. LITO runs the benchmarks each k versions to collect run-time information (*e.g.,* each ten versions, $k = 10$). Based on the study presented in previous sections, LITO considers three aspects to collect run-time information in each sample:

- *Control flow* – LITO records sections of the source code and method calls that are executed. This allows LITO to ignore changes made in source code sections that are not executed by the benchmarks (*e.g.,* a code block associated to an if condition or a method that is never executed).

- *Number of executions* – As we presented in the previous sections, the method call cost itself is not enough to detect possible performance regressions. Therefore LITO records the number of times that methods and loops are executed.

- *Method call cost* – LITO associates the average execution time of each method as the cost of executing each method call. Note that LITO does not estimate the execution time variation itself, it uses this average as a metric to detect possible performance regressions.

- *Method execution time* – LITO estimates for each method m (i) the accumulated total execution time and (ii) the average execution time for calling m once during the benchmarks executions.

LITO Cost Model. LITO abstracts all source code changes as a set of method calls additions and/or deletions. To LITO, a *method call swap* is abstracted as a method call addition and deletion. *Block additions*, such as loops and conditional blocks, are abstracted as a set of method call additions.

The LITO cost model is illustrated in Figure 3. Consider the modification made in the method parseOn: in the class PPSequenceParser. In this method revision, one line has been removed and two have been added: two method call additions (remember and restore:) and one deletion (position:). In order to determine whether the new version of parseOn: is slower or faster than the original version, we need to estimate how the two call additions compare with the call deletion in terms of execution time. This estimation is based on an execution profile.

The LITO cost model assesses whether a software version introduces a performance regression for a particular benchmark. The cost of each call addition and deletion depends therefore on the benchmark b when the execution profile is produced.

We consider an execution profile obtained from the execution of a benchmark on the version of the application that contains the original definition of parseOn:. LITO determines

whether the revised version of parseOn: does or does not introduce a performance regression based on the execution profile of the original version of parseOn:.

The execution profile indicates the number of times that each block contained in the method parseOn: is executed. It further indicates the number of executions of the code block contained in the iteration (*i.e.,* do: [:index | ...]). The profile also gives the number of times the code block contained in the ifTrue: statement is executed. In Figure 3, the method parseOn: is executed 10 times, the iteration block is executed 100 times (*i.e.,* 10 times per single execution of parseOn: on average) and the conditional block is executed 50 times (*e.g.,* 0.5 time per single execution of parseOn: on average).

LITO uses the notion of *cost* [12] as a proxy of the execution time. We denote u as the unit of time we use in our cost model. In our setting, u refers to the number of times the send message bytecode is executed by the virtual machine. We could have used a direct time unit as milliseconds, however it has been shown that counting the number of sent messages is significantly more accurate and this metric is more stable than estimating the execution time [4]. On the example, the method parseOn: costs 1000u, and remember 100u, implying that remember is 10 times faster to execute than parseOn:.

The modification cost estimates the cost difference between the new version and original version of a method. On the example, the modification cost of method parseOn: is 8500u, meaning that the method parseOn: spends 8500u more than previous version for a given benchmark b. For instance, if the benchmark b execution time is 10,000u, then the new version of the method parseOn: results in a performance regression of 85%.

The average cost of calling each method is obtained by dividing the total accumulated cost of a method m by the number of times m has been executed during a benchmark execution. In our example, calling remember has an average cost of 100u. The theoretical cost of a method call addition m is assessed by multiplying the cost of calling m and the number of times that it would be executed based on the execution profile (Figure 3 right hand).

Let A_i be a method call addition of a given method modification and D_j a method call deletion. Let be $cost_b$ a function that returns the average cost of a method call when executing benchmark b, and $exec_b$ a function that returns the number of times a method call is executed. Both functions lookup the respective information in the last execution sample gathered by LITO.

Let $MC_b(m)$ be the cost of modify the method m for a benchmark b, na the number of method call additions and nd the number of method call deletions. The method modification cost is the sum of the cost of all method call additions less the cost of all method call deletions.

$$MC_b(m) = \sum_{i=1}^{na} cost_b(A_i) * exec_b(A_i) - \sum_{j=1}^{nd} cost_b(D_j) * exec_b(D_j).$$

Let C be the cost of all method modifications of a software version, and m the number of modified methods, we therefore have:

$$C[v, b] = \sum^{m \in v} MC_b(m)$$

In case we have $C[v, b] > 0$ for a particular version v and a benchmark b, we then consider that version v introduces a performance regression.

New Method, Loop Addition, and Conditions. Not all the methods may have a computed cost. For example, a new method, for which no historical data is available, may incur a regression. In such a case, we statically determine the cost for code modification with no historical profiling data.

We qualify as fast a method that is returning a constant value, an accessor / mutator, or doing arithmetic or logic operations. A fast method receives the lowest method cost obtained from the previous execution profile. All other methods receive a high cost, the maximal cost of all the methods in the execution profile.

In case a method is modified with a new loop addition or a conditional block, no cost has been associated to it. LITO hypothesizes that the conditional block will be executed and the loop will be executed the same number of times as the most recently executed enclosing loop in the execution profile.

The high cost we give to new methods, loop additions, and conditions is voluntarily conservative. It assumes that these additions may trigger a regression. As we show in Table 5, loop and conditional block additions represent 6% and 2%, respectively, of the source code changes that affect software performance.

Project Dependencies. An application may depend on externally provided libraries or frameworks. As previously discussed (Section 3), a performance regression perceived by using an application may be in fact located in a dependent and external application. LITO takes such analysis into account when profiling benchmark executions. The generated profile execution contains runtime information not only of the profiled application but also of all the dependent code.

During our experiment, we had to ignore some dependencies when analyzing the Nautilus project. Nautilus depends on two external libraries: ClassOrganizer and RPackage. LITO uses these two libraries. We exclude these two dependencies in order to simplify our analysis and avoid unwanted hard-to-trace recursions. In the case of our experiment, any method call toward ClassOrganizer or RPackage is considered costly.

4.2 Evaluation

For the evaluation, we use the project versions where at least one benchmark can be executed. In total, we evaluate LITO over 1,125 software versions. We use the following 3-steps methodology to evaluate LITO:

S1. We run our benchmarks for all 1,125 software versions and measure performance regressions.

S2. We pick a sample of the benchmark executions, every k versions, and apply our cost model on all the 1,125 software versions. Our cost model identifies software versions that introduce a performance regression.

S3. Contrasting the regressions found in S1 and S2 will measure the accuracy of our cost model.

Step S1 - Exhaustive Benchmark Execution. Consider two successive versions, v_i and v_{i-1} of a software project P and a benchmark b. Let $\mu[v_i, b]$ be the mean execution time to execute benchmark b multiple times on version v_i.

Table 7: Detecting performance regressions with LITO using a threshold=5% and a sample rate of 20.

Project	Versions	Selected Versions	Performance Regressions	Detected Perf. Reg.	Undetected Perf. Reg.	Performance Evolution by benchmark
Spec	267	43(16%)	11	8 (73%)	3	
Nautilus	199	64 (32%)	5	5 (100%)	0	
Mondrian	144	9 (6%)	2	2 (100%)	0	
Roassal	141	26 (18%)	3	3 (100%)	0	
Morphic	135	8 (6%)	2	1 (50%)	1	
GraphET	68	20 (29%)	5	4 (80%)	1	
Rubric	64	2 (3%)	0	0 (100%)	0	
XMLSupport	18	8 (44%)	4	4 (100%)	0	
Zinc	18	2 (11%)	0	0 (100%)	0	
GTInspector	16	1 (6%)	1	1 (100%)	0	
Shout	15	0 (0%)	1	0 (0%)	1	
Regex	12	1 (8%)	1	1 (100%)	0	
NeoCSV	9	3 (33%)	0	0 (100%)	0	
NeoJSON	7	0 (0%)	0	0 (100%)	0	
PetitParser	6	1 (17%)	1	1 (100%)	0	
Soup	4	0 (0%)	0	0 (100%)	0	
XPath	2	0 (0%)	0	0 (100%)	0	
Total	1125	188 **(16.7%)**	36	30 **(83.3%)**	6 **(16.7%)**	

The execution time is measured in terms of sent messages (u unit, as presented earlier). Since this metric has a great stability [4], we executed each benchmark only 5 times and took the average number of sent messages. It is known that the number of sent messages is linear to the execution time in Pharo [4].

We define the time difference between versions v_i and v_{i-1} for a given benchmark b as:

$$D[v_i, b] = \mu[v_i, b] - \mu[v_{i-1}, b] \quad (1)$$

Consequently, the time variation is defined as:

$$\Delta D[v_i, b] = \frac{D[v_i, b]}{\mu[v_{i-1}, b]} \quad (2)$$

For a given *threshold*, we say v_i introduces a performance regression if it exists a benchmark b_j such that $\Delta D[v_i, b_j] \geq$ *threshold*.

Step S2 - Applying the Cost Model. Let $C[v_i, b]$ be the cost of all modifications made in version v_i from v_{i-1}; using the run-time history of benchmark b.

$$\Delta C[v_i, b] = \frac{C[v_i, b]}{\mu[v_j, b]} \quad (3)$$

We have j, the closest inferior version number that has been sampled at an interval k. If $C[v_i, b] \geq$ *threshold* in at least one benchmark, then LITO considers that version v_i may introduce a performance regression.

Step S3 - Contrasting $\Delta C[v_i, b]$ ***with*** $\Delta D[v_i, b]$**.** The cost model previously described (Section 4.1) is designed to favor the identification of performance regression. Such design is reflected in the high cost given to new methods, loop additions, and conditions. We therefore do not consider performance optimizations in our evaluation.

Results. We initially analyze the software versions with LITO and collect the run-time information each $k = 20$ versions, and a *threshold* of 5%. LITO is therefore looking for all the versions that introduce a performance regression of at least 5% in one of the benchmarks. These benchmarks are executed every 20 software versions to produce execution profiles that are used for all the software versions. LITO uses the cost model described previously to assess whether a software version introduces a regression or not.

Table 7 gives the results of each software project. During this process LITO selected 189 costly versions that represent 16.7% of total of analyzed versions. These selected versions contain 83.3% of the versions that effectively introduce a performance regression greater than 5%. In other words, based on the applications we have analyzed, practitioners could detect 83.3% of the performance regressions by running the benchmarks on just 16.8% of all versions, picked at a regular interval from the total software source code history.

Table 8 shows that LITO has a high recall (83.3%) despite having a low precision (15.95%). This high recall indicates that LITO helps practitioners to identify a great portion of the performance regressions by running the benchmarks over a few software versions.

Threshold. To understand the impact of the threshold in our cost model, we carry out the experiment described above but using different thresholds (5, 10, 15, 20, 25, 30, 35, 40, 45, and 50). Figure 4 shows the percentage of selected versions and detected performance regressions by LITO. Figure 4 shows that LITO detects all regressions greater than 50%

Table 8: Precision and recall of LITO to detect performance regressions greater than 5%(threshold) using a sample-rate of 20. (TP = true-positive, TN = true-negative, FP = false-positive, FN = false-negative, Prec. = Precision)

Project	TP	FP	FN	TN	Prec.	Recall
Spec	8	35	3	221	0.19	0.73
Nautilus	5	59	0	135	0.08	1
Mondrian	2	7	0	135	0.22	1
Roassal	3	23	0	115	0.12	1
Morphic	1	7	1	126	0.13	0.5
GraphET	4	16	1	47	0.2	0.8
Rubric	0	2	0	62	0	-
XMLSupport	4	4	0	10	0.5	1
Zinc	0	2	0	16	0	-
GTInspector	1	0	0	15	1	1
Shout	0	0	1	14	-	0
Regex	1	0	0	11	1	1
NeoCSV	0	3	0	6	0	-
NeoJSON	0	0	0	7	-	-
PetitParser	1	0	0	5	1	1
Soup	0	0	0	4	-	-
XPath	0	0	0	2	-	-
Total	30	158	6	931	**15.95%**	**83.33%**

(totaling ten). Figure 4 also shows that the number of selected versions decreases as the threshold increases, meaning that LITO safely discards more versions because their cost is not high enough to cause a regression with a greater threshold.

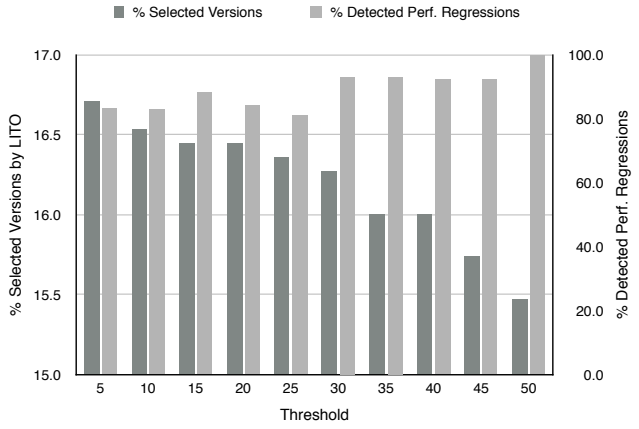

Figure 4: The effect of the threshold on the percentage of detected performance regressions and the percentage of selected versions by LITO (> threshold)

> By profiling the execution of only 17% of the versions, our model is able to identify 83% of the performance regressions greater than 5% and 100% of the regressions greater than 50%. Such versions are picked at a regular interval from the software source code history.

Sample Rate. To understand the effect of the sample rate, we repeated the experiment using different tree sample rates 1, 20 and 50. Figure 5 shows the percentage of performance regressions by LITO with the different sample rates. As it was expected, the accuracy of LITO increment when we take a sample of the execution every version (sample rate = 1). Consequently the accuracy get worse when we take a sample each 50 versions. Figure 5 shows that sampling a software source code history each 50 versions make LITO able to detect a great portion of the performance regression, for any threshold lower than 50%.

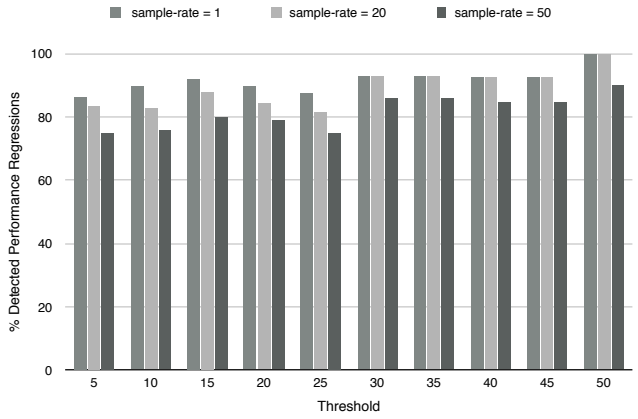

Figure 5: Evaluating LITO with sample rates of 1, 20, and 50

Overhead. Statically analyzing a software version with LITO takes 12 seconds (on average). It is considerably cheaper than executing the benchmarks in a software version. However, each time that LITO collects the run-time information is seven times (on average) more expensive than executing the benchmarks. LITO instruments all method projects, and executed twice the benchmarks: the first one to collect the average time of each method and the second one to collect the number of executions of each source code section. Even with this, the complete process of prioritizing the versions and executing a performance testing over the prioritized versions is far less expensive than executing the benchmarks over all application versions.

For instance, in our experiment, the process to do an exhaustive performance testing in all software versions takes 218 hours; on the other hand, the process of prioritize the versions and executed the benchmarks only in the prioritized versions takes 54 hours (25%).

5. THREATS TO VALIDITY

To structure the threats to validity, we follow the Wohlin *et al.* [29] validity system.

Construct Validity. The method modifications we have manually identified may not be exhaustive. We analyzed method modifications that cause performance variations greater than 5%, over the total execution time of the benchmark. Analyzing small performance variations, such as the one close to 5%, is important since it may sum up over multiple software revisions. Detecting and analyzing variations smaller variation is difficult, because many factors may distort variance to the observable performance, such as inaccuracy of the profiler [4].

External Validity. This paper is voluntarily focused on the Pharo ecosystem. We believe this study provides relevant findings about the performance variation in the studied projects. We cannot be sure of how much the results gen-

eralize to other software projects beyond the specific scope this study was conducted. As future work, we plan to replicate our experiments for the Javascript and Java ecosystem. In addition, we plan to analyze how LITO performs with multi-thread applications.

Internal Validity. We cover diverse categories of software projects and representative software systems. To minimize the potential selection bias, we collect all possible release versions of each software project, without favoring or ignoring any particular version. We manually analyze twice each method modification: the first time to understand the root-cause of the performance variation and the second time to confirm the analysis.

6. RELATED WORK

Performance Bug Empirical Studies. Empirical studies over performance bug reports [13, 24] provide a better understanding of the common root causes and patterns of performance bugs. These studies help practitioners save manual effort in performance diagnosis and bug fixing. These performance bug reports are mainly collected from the tracking system or mailing list of the analyzed projects.

Zaman *et al.* [30] study the bug reports for performance and non-performance bugs in Firefox and Chrome. They studied how users perceive the bugs, how bugs are reported, what developers discuss about the bug causes and the bug patches. Their study is similar to that of Nistor *et al.* [23] but they go further by analyzing additional information for the bug reports. Nguyen *et al.* [21] interviewed the performance engineers responsible for an industrial software system, to understand these regression-causes.

Sandoval *et al.* [1] have studied performance evolution against software modifications and have identified a number of patterns from a semantic point of view. They describe a number of scenarios that affect performance over time from the intention of a software modification (vs the actual change as studied in this paper).

We focus our research on performance variations. In this sense we consider performance drops and improvements that are not reported as a bug or a bug-fix. We contrast the performance variations with the source code changes at method granularity. In addition, we analyze what kind of source code changes cause performance variations in a large variety of applications.

Performance Bug Detection and Root-Cause Analysis. Great advances have been made to automate the performance bug detection and root-cause analysis [10, 19, 27]. Jin *et al.* [13] propose a rule-based performance-bug detection using rules implied by patches to found unknown performance problems. Nguyen *et al.* [21] propose the mining of a regression-causes repository (where the results of performance tests and causes of past regressions are stored) to assist the performance team in identifying the regression-cause of a newly-identified regression. Bezemer *et al.* [6] propose an approach to guide performance optimization processes and to help developers find performance bottlenecks via execution profile comparison. Heger *et al.* [11] propose an approach based on bisection and call context tree analysis to isolate the root cause of a performance regression caused by multiple software versions.

We improve the performance regression overhead by prioritizing the software versions. We believe that our work complements these techniques in order to help developers address performance related issues. We do not attempt to detect performance regression bugs or provide root-cause diagnosis.

Performance Regression Testing Prioritization. Different strategies have been proposed in order to reduce the functional regression testing overhead, such as test case prioritization [9, 25] and test suite reduction [7, 14, 16, 31]. However, few projects have been able to reduce the performance regression testing overhead.

Huang *et al.* [12] propose a technique to measure the risk given to a code commit in introducing performance regressions. Their technique uses a full static approach to measure the risk of a software version based on worst case analysis. They automatically categorize the source code change (*i.e.,* extreme, high, and low) and assign a risk score to each category; these scores may require an initial tuning. However, a fully static analysis may not accurately assess the risk of performance regression issues in dynamic languages. For instance, statically determining the loop boundaries may not be possible without special annotations [28]. Dynamic features of programming languages such as dynamic dispatching, recursion and reflexion make this task more difficult.

In this paper we propose a hybrid (dynamic and static) technique to automatically prioritize the performance testing; it uses the run-time history to track the control flow and the loop boundaries. Our technique reduces a number of limitations of a fully static approach and does not need an initial tuning. We believe that these techniques can complement each other to provide a good support for developers and reduce the overhead of performance regression testing.

7. CONCLUSION

This paper studies the source code changes that affect software performance of 17 software projects along 1,288 software versions. We have identified 10 source code changes leading to a performance variation (improvement or regression). Based on our study, we propose a new approach, *horizontal profiling*, to reduce the performance testing overhead based on the run-time history.

As future work, we plan to extend our model to prioritize benchmarks and generalize *horizontal profiling* to identify memory and energy performance regressions.

8. ACKNOWLEDGMENTS

Juan Pablo Sandoval Alcocer is supported by a Ph.D. scholarship from CONICYT, Chile. CONICYT-PCHA/Doctorado Nacional para extranjeros/2013-63130199. We also thank the European Smalltalk User Group (www.esug.org) for the sponsoring. This work has been partially sponsored by the FONDECYT 1160575 project and STICAmSud project 14STIC-02.

9. REFERENCES

[1] Juan Pablo Sandoval Alcocer and Alexandre Bergel. Tracking down performance variation against source code evolution. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015. ACM.

[2] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, jun 2015.

[3] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Syst. J.*, 15(3):225–252, September 1976.

[4] Alexandre Bergel. Counting messages as a proxy for average execution time in pharo. In *Proceedings of ECOOP'11*.

[5] C. Bezemer, E. Milon, A. Zaidman, and J. Pouwelse. Detecting and analyzing i/o performance regressions. *Journal of Software: Evolution and Process*, 26(12):1193–1212, 2014.

[6] C. Bezemer, E. Milon, A. Zaidman, and J. Pouwelse. Detecting and analyzing i/o performance regressions. *Journal of Software: Evolution and Process*, 26(12):1193–1212, 2014.

[7] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of ICSE '04*. IEEE.

[8] Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, first edition, 2007.

[9] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. *SIGSOFT Softw. Eng. Notes*, 25(5):102–112, August 2000.

[10] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of ICSE 2012*.

[11] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *Proceedings of ICPE '13*.

[12] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of ICSE '14*.

[13] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *SIGPLAN Not.*, 47(6):77–88, June 2012.

[14] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of ICSE '02*.

[15] M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Symposium on Software Metrics*, METRICS '97, pages 20–32. IEEE Computer Society.

[16] Zheng Li, M. Harman, and R.M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, April 2007.

[17] Nazim H. Madhavji, Juan Fernandez-Ramil, and Dewayne E. Perry. *Software Evolution and Feedback: Theory and Practice*. Wiley, Chichester, UK, 2006.

[18] H. Malik, Zhen Ming Jiang, B. Adams, A.E. Hassan, P. Flora, and G. Hamann. Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *14th European Conference on Software Maintenance and Reengineering (CSMR), 2010*.

[19] D. Maplesden, E. Tempero, J. Hosking, and J.C. Grundy. Performance analysis for object-oriented software: A systematic mapping. *IEEE Transactions on Software Engineering*, 41(7):691–710, July 2015.

[20] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 1st edition, 2009.

[21] Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. An industrial case study of automatically identifying performance regression-causes. In *Proceedings of MSR '14*.

[22] Thanh H.D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of ICPE '12*.

[23] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of MSR '13*.

[24] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of ICSE '13*.

[25] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Test case prioritization: an empirical study. In *Proceedings of ICSM '99*.

[26] Wei Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using regression models on clustered performance counters. In *Proceedings of ICPE '15*.

[27] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of ISSTA '15*.

[28] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem; overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[29] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000.

[30] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *Proceedings of MSR '12*.

[31] Hao Zhong, Lu Zhang, and Hong Mei. An experimental comparison of four test suite reduction techniques. In *Proceedings of ICSE '06*.