Prioritizing Versions for Performance Regression Testing: The Pharo Case

Juan Pablo Sandoval Alcocer[a,*], Alexandre Bergel[b], Marco Tulio Valente[c]

[a] *Departamento de Ciencias Exactas e Ingeniería*
Universidad Católica Boliviana "San Pablo", Cochabamba, Bolivia
[b] *ISCLab, Department of Computer Science (DCC), University of Chile, Santiago, Chile*
[c] *Federal University of Minas Gerais, Brazil*

Abstract

Context: Software performance may suffer regressions caused by source code changes. Measuring performance at each new software version is useful for early detection of performance regressions. However, systematically running benchmarks is often impractical (*e.g.,* long running execution, prioritizing functional correctness over non-functional).
Objective: In this article, we propose *Horizontal Profiling*, a sampling technique to predict when a new revision may cause a regression by analyzing the source code and using run-time information of a previous version. The goal of *Horizontal Profiling* is to reduce the performance testing overhead by benchmarking just software versions that contain costly source code changes.
Method: We present an evaluation in which we apply *Horizontal Profiling* to identify performance regressions of 17 software projects written in the Pharo programming language, totaling 1,288 software versions.
Results: *Horizontal Profiling* detects more than 80% of the regressions by benchmarking less than 20% of the versions. In addition, our experiments show that *Horizontal Profiling* has better precision and executes the benchmarks in less versions that the state of the art tools, under our benchmarks.
Conclusions: We conclude that by adequately characterizing the run-time information of a previous version, it is possible to determine if a new version is likely to introduce a performance regression or not. As a consequence, a signif-

*Corresponding Author
Email addresses: sandoval@ucbcba.edu.bo (Juan Pablo Sandoval Alcocer), bergel@dcc.uchile.cl (Alexandre Bergel), mtov@dcc.ufmg.br (Marco Tulio Valente)

icant fraction of the performance regressions are identified by benchmarking only a small fraction of the software versions.

Keywords: Performance regression, software performance, software evolution, performance regression prediction, regression benchmarking

---

1. Introduction

Performance regressions are caused by source code changes made along the software development process. Identify which software version introduces a performance regression is important for addressing performance regressions. Measuring and comparing the performance of each software version under the same workload is a common technique to spot which version causes the performance regression. However, sometimes measuring the performance of each software version is impractical for different factors, including: i) the high-overhead of the benchmark execution [9], ii) difficulty in identifying multiple executions to achieve comparable performance measurements [8], and iii) the amount of software versions released by day [9]. Furthermore, besides the performance testing activities, developers first need to test if the new version remains functional. Functional regression testing could also be a considerable time-consuming activity [12].

Understanding *how* and *what* source code changes affect software performance may help reduce the overhead associated to performance regression testing. For instance, given a particular source code change one may anticipate whether or not it could affect software performance. Diverse empirical studies analyze performance bug reports to better understand the roots of performance bugs [9, 10, 14]. However, these studies voluntarily ignore performance regressions that are not reported as a bug or bug-fix.

Previous work. This paper is an extension of a paper presented in ICPE'16 [1]. In this previous work, we conducted an empirical study of real-world performance variations detected after analyzing the performance evolution of 17 open source projects along 1,288 software versions [1]. In particular, this study addressed two research questions:

- *RQ1: Are performance variations mostly caused by modifications of the same methods?* This question is particular critical to understand where performance variations stem 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 portion of "risky" methods may be 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.

Our study reveals 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.

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

Horizontal Profiling. Based on these findings, we also proposed *Horizontal Profiling*, a sampling technique to statically identify versions that may introduce a performance regression [1]. *Horizontal Profiling* 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 evaluated our technique over 1,125 software versions. By profiling the execution of only 17% of the versions, Horizontal Profiling is able to identify 83% of the performance regressions greater than 5% and 100% of the regressions greater than 50%.

PerfScope. Related to our approach, Huang *et al.* [9] propose a technique to measure the risk of introducing performance regressions of a source code change, together with a tool PerfScope. PerfScope uses a static approach to measure the risk of a software version based on the worst case analysis. PerfScope categorizes the source code change (*i.e.,* extreme, high, and low) and assigns a risk score to each category. Both approaches rely on the fact that most performance regressions depend on 1) how expensive the involved source changes are and 2) how frequently these changes are executed.

Empirical Comparison. In this paper extension, we present an empirical comparison between *Horizontal Profiling* and a Pharo implementation of *PerfScope*. We argue that PerfScope may not accurately assess the risk of performance regression issues in dynamic languages, like the Pharo programming language. In order to support our hypothesis we address a fourth research question.

- *RQ4: How well does Horizontal Profiling perform compared to the state-of-the-art risk analysis tools using a dynamically typed language?* We are interested in comparing how well these approaches predict if a new software version introduces a performance regression or not, in terms of precision and recall.

We found that *Horizontal Profiling* can more accurately estimate the expensiveness and frequency of source code changes. As a consequence, it has a better precision and executes the benchmarks in less versions than PerfScope, under our benchmarks. By using a dedicated profiling technique, Horizontal Profiling does not require painful manual tuning, and it performs well, independently of the performance regression threshold.

Outline. Section 2 motivates our work thought an empirical study about the roots of performance variations. Section 3 presents and evaluates the cost model based on the run-time history. Section 4 compares our proposed technique with the state-of-the-art risk analysis tools. 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. Understanding Performance Variations

To address our first two research questions. 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 for iteratively executing 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.

Pharo has a different syntax compared with C-like languages. Readers unfamiliar with the syntax of Pharo might need an introduction to the Pharo Syntax to better understand the examples presented in this paper. For this purpose, we provide equivalent expressions for common cases in Table A.11 in the Appendix section.

2.1. The Pharo programming language

Pharo is an emerging object-oriented programming language that is close to Python and Ruby. Pharo's syntax follows the one of Smalltalk, and Pharo has a minimal core and few but strong principles. In Pharo, sending a message is the primitive syntactic construction from which all computations are expressed. Loops, object creations, and conditional branches are all realized via sending message. In addition, the virtual machine of Pharo is relatively simple, which greatly reduces the number of source of bias. The simplicity of both the language and executing environment greatly mitigate possible biases resulting from the technical challenges we have to address to run benchmark over multiple software revisions.

Although we conducted our experiment using Pharo, we have no indication that our results and claims are not applicable to other programming languages and environments.

2.2. Projects under Study

We pick 1,288 release versions of 17 software projects from the Pharo ecosystem stored on the Pharo forges (*SqueakSource*[2], *SqueakSource3*[3] and

[1] `http://pharo.org`
[2] `http://www.squeaksource.com/`
[3] `http://ss3.gemstone.com/`

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**

SmalltakHub[4]). The set of considered projects has a broad range of application: user interface frameworks (Morphic and Spec), a source code highlighter (Shout), visualization engines (Roassal and Mondrian), an 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 number of lines of code (LOC) per project.

We chose these applications 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 for us to ask questions about their projects.

2.3. 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.

[4] `http://smalltalkhub.com/`

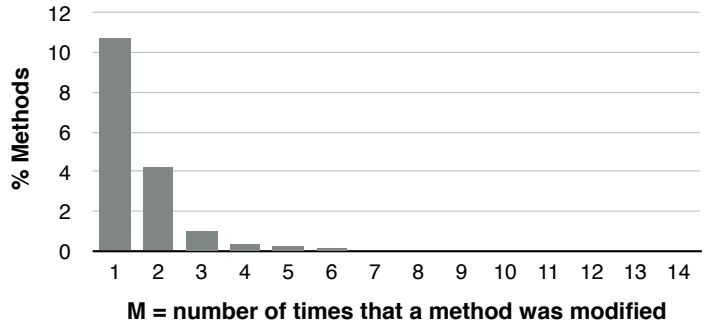

Figure 1: Source Code Changes histogram at method level.

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 found that in all but one project, the number of methods modified more than once are relatively small compared to the number of methods that are modified only 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.

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.

2.4. Benchmarks

In order to get reliable and repeatable execution footprints, we select a number of benchmarks for each considered application. Each benchmark is a representative execution scenario that we will carefully measure. Several of

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%)**

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 with the benchmarks are available online[5].

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

2.5. Execution Time Measurements

Having a good execution time estimation is challenging, because there are many factors that may affect the measurements, for instance, the just-in-time compiler and no deterministic garbage collection. To reduce the bias in our measurements, we follow three actions:

- *Warming-up.* We execute the benchmarks a number of times before start our measurements.

- *Compteur.* We use the *Compteur* profiler, it uses the number of sent message for estimate the average execution time of a given benchmark [2]. In particular, since most of computation in Pharo is done by sending messages (even for the control structures and loops) the execution time has a very high correlation with the number of sent messages. Therefore, counting messages provides more deterministic and replicable results, which is crucial in an empirical study.

- *Multiple Executions.* Besides the great advantages of message counting, it is necessary consider that the number of sent message have a small variation mainly due to the hash values (*i.e.,* used on dictionaries), which are generated by the Pharo Virtual Machine in a non-deterministic fashion. Even do, it have been show that this variation is small, below 1% [2]. To reduce such variation, we execute the benchmark 5 times (after the warming up session) and use the media for our experiments.

2.6. 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.

We carefully conducted a quantitative study about source code changes that directly affect 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 [2].

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.

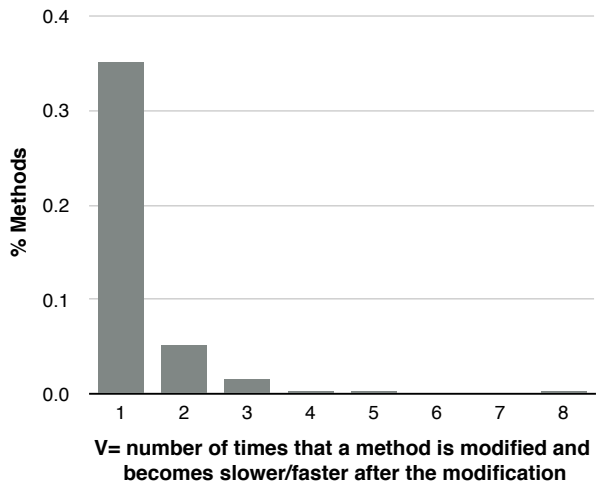

Figure 2: Performance Variations of Modified Methods (threshold = 5%), 111 methods are here reported.

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 . 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.

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

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%)**

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.

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.

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

Summary. 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 with performance regression 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.Therefore, we answer our first research question as follows:

RQ1: Are performance variations mostly caused by modifications of the same methods? 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.

2.7. Negative Performance Variations: Author Feedback

Accurately identifying the root of a negative performance variation is difficult. We investigate this by surveying authors of method modifications causing a negative variation. From the 84 method modifications mentioned, we obtained author feedback for 21 of them. Each of 21 method modifications

is the cause of a negative variation 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 the method modification causing the regression.

For 6 (29%) out of 21 modifications, the authors were aware of the variation 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 variation.

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

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

This small and informal survey of practitioners indicates that a significant number of performance negative variation are apparently inevitable. On the other hand, such incertitude expressed by the authors regarding the presence of a negative variation and providing change alternative highlights the relevance of our study and research effort.

2.8. 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 we 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 in the whole set of method modifications. Such a change resulted in a performance regression.

```
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.

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%)

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. In total, we found 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. Therefore, we answer our second research question as follows:

RQ2: What are the recurrent source code changes that affect performance along software evolution? Most source code changes that cause a performance variation are directly related to method call addition, deletion or swap.

2.9. 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. 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 caused 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:

[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 in detecting this pattern automatically.

- *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 for determining 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 varies depending on how many times the method call is executed and the cost of each method call execution.

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.

3. 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.

3.1. LITO: A Horizontal Profiler

We built LITO to statically identify software versions that introduce a performance regression. 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 component 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 with 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 estimates for each method m (i) the accumulated total execution cost and (ii) the average execution cost for calling m once during the benchmark executions. Note that LITO uses the notion of *cost* as a proxy of the execution time. We denote u as the unit of time we use in our cost model. 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 [2].

LITO Cost Model. LITO abstracts all source code changes as a set of method call 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

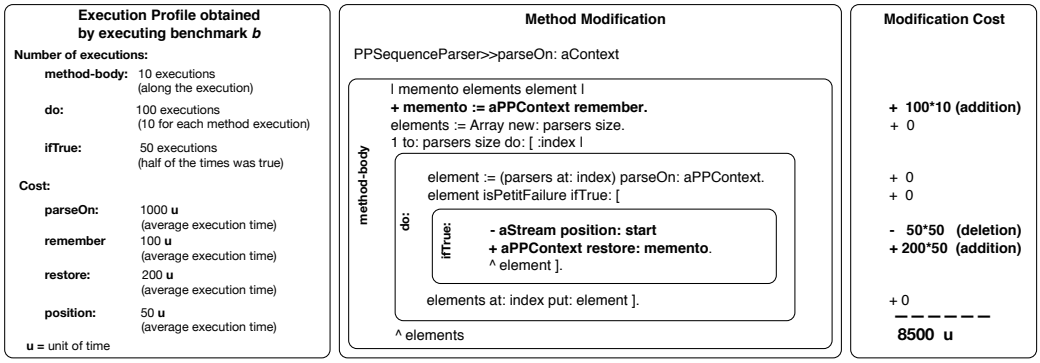

Figure 3: LITO cost model example

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 times per single execution of parseOn: on average).

The execution profile also has the cost of each method call. On the example, the method parseOn: costs 1000u, and remember 100u, implying that remember

is 10 times faster to execute than parseOn:. Where u refers to the number of times the send message bytecode is executed by the virtual machine [2].

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 the previous version for a given benchmark b. For instance, if the benchmark b execution cost 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 $cost_b$ be 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 modifying 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 the case when a method is modified with a new loop addition or a conditional block, no cost has been associated with it. LITO hypothesizes that the conditional block will be executed and the loop will be executed the same number of times as the most executed 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.

Detecting Method Call Additions and Deletions. Horizontal profiling also depends on accurately detecting method call additions and deletions. LITO uses a very simple strategy to detect these method call differences. LITO takes as input two software versions, it contrasts the versions method by method through their abstract syntax tree (AST). Consider two versions of the same method m and m', where m' is the newer version of the method. The strategy is to find all the nodes of both trees that are equivalent. We consider that two nodes of the AST are equivalent if they have the same values (*i.e.,* receiver, method name, and arguments) and the same path to the root. As consequence, the method call nodes of m that have not a equivalent in m' are considered deleted, and the nodes that exist in m' and does not in m are considered added. For instance, when a developer move a line in to a loop block, this line will have a different path to the root in the tree. Therefore, the algorithm will consider two changes a deletion and an addition. Our initially attempt, prior having the algorithm we just described, was to solely rely on methods names and method signature. However, this is very fragile when a method is renamed (history is then lost). In this naive initial approach, the method is considered new and it will be a worst case scenario as we describe in the previous paragraph. This is a common problem when contrasting source code in different versions. However, in next sections we

show that this fact have few impact in our benchmark. Note also that other approach have the same limitation.

Mapping Run-time Information. Once the method call additions and deletions are located. LITO maps each method call with its associated cost and the number of executions from the profiling history. We use the method signature to search the method call cost in the profiling history. For the number of executions, we search the closest block closure node in the Abstract Syntax Tree (*i.e.,* the closest loop) from the method call to the root, then we search an equivalent node in the profiling history to determine how many times this block was executed in the past. Using the same definition of equivalence presented in previous paragraph. Note that in Smalltalk the loops and conditions are represented as a block closures node in the Abstract Syntax Tree. If the block closure node does not have an equivalent in the profiling history then as we mention before, we take the worst case scenario described in previous paragraphs.

Project Dependencies. An application may depend on externally provided libraries or frameworks. As previously discussed (Section 2), a performance regression perceived by using an application may be in fact located in a dependent and external application. LITO takes such analyses 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.

3.2. Evaluation

To answer our third research questionwe evaluate LITO using project versions where at least one benchmark can be executed. We use the following 3-step methodology to evaluate LITO:

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

Table 7: Detecting performance regressions with LITO using a threshold=5% and a sample rate of 20 (ver= versions, Perf. Reg. = Performance Regressions).

Project	Ver.	Selected Versions	Perf. Reg.	Detected Perf. Reg.	Undetected Perf. Reg.	Perf. 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%)**	

S2. We pick a sample of the benchmark executions, every k version, 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. We 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 cost to execute benchmark b multiple times on version v_i. The execution time cost is measured in terms of sent messages (u unit, as presented earlier). Since this metric has a great stability [2], 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 [2] and have reproducible over multiple executions.

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

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

Consequently, the time variation is defined as:

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

For a given *threshold*, we say v_i introduces a performance regression if 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.

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

We have j, the closest inferior version number that has been sampled at an interval k. If $\Delta C[v_i, b] \geq$ *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 3.1) 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 the total 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.

We further analyze the performance regression that LITO was not able to detect and we found two main reasons. First, a number of performance regressions were cause by changes in the project dependencies that we were not able to collect runtime information. Second, in some cases the run-time information was outdated. For instance, a method that was cheap to execute when LITO took the sample become expensive the the run-time information was not updated until the next sample. Therefore, a number of performance regressions were not detected due the sampling strategy for collecting run-time information.

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% (totaling 10). The figure 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.

Therefore, we answer our third research question as follows:

RQ3: How well can Horizontal Profiling prioritize the software versions and reduce the performance testing overhead? 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 multiple sample rates (10, 20, 30, 40, 50, 60, 70, 80, 90 and 100). Figure 5 shows the percentage of performance regressions detected by LITO with the different sample rates. As expected, the accuracy of LITO increased when we take a sample of the execution every 10 versions (sample rate = 10). Consequently the accuracy gets worse when we take a greater sample rate (*i.e.,* >= 50). Figure 5 shows that sampling a software source code history each 100 versions allows LITO to detect a great portion of the

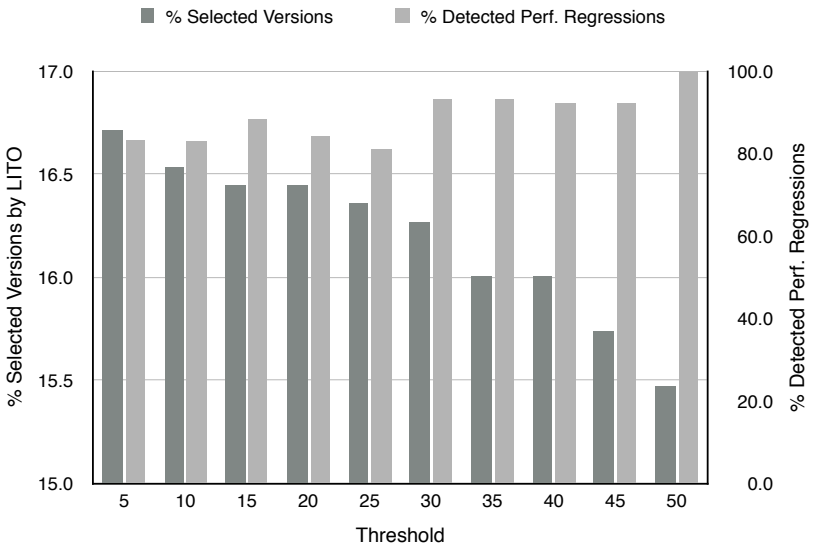

Figure 4: The effect of the threshold on the percentage of detected performance regressions and the percentage of selected versions by LITO (> threshold).

performance regression, for any threshold lower than 50%. Independent of the sample rate, LITO detects a considerable fraction of performance regressions. For instance, by taking only one sample LITO is able to detect between the 72 and 80% of the performance regressions depending of the performance regression threshold. Independent of the sample rate and the threshold, our results also show that LITO selects less than the 20% of the software versions.

Note that with a sample rate of 60 the recall is 100%. This is because the samples were taken close to the versions that introduce the performance regressions.

Overhead. The time to statically analyze a software version depends on the number of methods, classes and lines of code. However, it is considerably cheaper than executing the benchmarks in a software version. For instance, LITO takes 12 seconds (on average) to analyze each software version. On the other hand, each time that LITO collects the run-time information is seven times (on average) more expensive than executing the benchmarks. Because LITO instruments all method projects, and executes the benchmarks twice, 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,

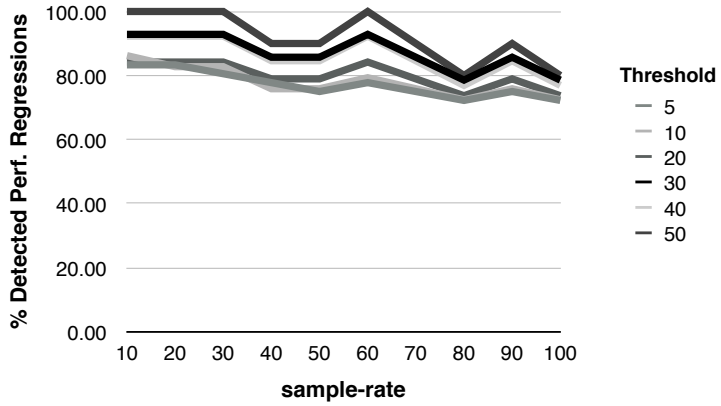

Figure 5: Evaluating LITO with sample rates of 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100. (threshold = 5, 10, 20, 30 and 50)

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 of conducting exhaustive performance testing in all software versions takes 218 hours; on the other hand, the process of prioritizing the versions and executing the benchmarks only in the prioritized versions takes 54 hours (25%).

4. Empirical Comparison

Related to our approach, Huang *et al.* [9] proposed PerfScope, a technique to measure the risk of introducing performance regressions for a code commit. Both approaches, PerfScope and LITO's Horizontal Profiling, rely on the fact that performance variations largely depend on 1) how expensive the involved source code changes are and 2) how frequently these changes are executed. As we discussed in previous sections, these two factors are good indicators to detect a large number of performance regressions.

To answer our fourth research question we carefully compare LITO against PerfScope, using applications written in Pharo, a dynamically typed object-oriented programming language. We therefore have implemented PerfScope in Pharo and run the experiment as described below.

4.1. PerfScope

PerfScope categorizes each source code added in the newer version using a scale of risk levels: *extreme*, *high*, *moderate* and *low*. This categorization is based on the expensiveness of the operation and the execution frequency, using Table 8. As presented in the original paper describing PerfScope [9], once the risk level distribution is measured, the total risk score of a new version is measured with the following formula:

$$risk\_score = N_{extreme} * 100 + N_{high} * 10 + N_{moderate} * \frac{1}{100} + N_{low} * \frac{1}{1000} \tag{4}$$

Where N_x is the number of operations in the category x. If the $risk\_score$ is greater than a $risk\_score\_threshold$ then PerfScope considers the new version as risky, for instance, using a $risk\_score\_threshold = 100$ means that all new versions with a $risk\_score >= 100$ are going to be considered risky. Therefore, the precision and recall depend on the $risk\_score\_threshold$ that needs to be manually set. Establishing a good $risk\_score\_threshold$ requires some initial tuning, depending on the performance regression threshold.

Table 8: Risk matrix of a change's expensiveness and frequency [9]

Expensiveness	**Frequency**		
	Frequent	**Normal**	**Rare**
Expensive	Extreme	High	Moderate
Normal	High	Moderate	Low
Minor	Moderate	Low	Low

4.1.1. Expensiveness

To estimate the expensiveness of a source code change PerfScope uses a static cost model. By using this cost model PerfScope determines if a change is expensive or not.

Code Addition. The cost model is focused only in code added in the new version. For instance, the expensiveness of added statements. The cost model emphasizes on relative cost rather than an absolute cost. PerfScope expresses the cost of an expression using an abstract unit, denoted as δ, instead of actual CPU cycles. The cost is measured depending on the expression.

This cost model is applied only to source code added in the new version.

- *Method calls.* A *call* instruction cost is equal to calling convention overhead plus the callee's cost.

- *Blocks.* A block is a section of code which is grouped together. A block consists of one or more declarations and statements. A block's cost is a sum of the cost of those statements that live inside the basic block.

- *Loops.* We consider a loop as a code block that is continually repeated until a certain condition is reached. The cost of a loop is the cost of the code block multiplied by the number of times that this block will be executed. For *loops*, PerfScope tries to statically infer the *trip count* (maximum number of iterations). If its *trip count* can be statically inferred, then PerfScope multiplies the code block cost by the *trip count*. Otherwise, the loop is considered to be frequently executed and therefore categorized as expensive.

- *Control flow.* For control flows, PerfScope adopts the worst case, taking the maximum cost among the possible paths. For instance, consider the method $parse\ MessagePattern$ of the class $SHParserST80$ in the *Shout* project:

```
SHParserST80>>parseMessagePattern
  self isName
    ifTrue: [self parseUnaryMessagePattern]
    ifFalse: [self isBinary
        ifTrue: [self parseBinaryMessagePattern]
        ifFalse: [self failUnless: self isKeyword.
          self parseKeywordMessagePattern]]
```

  It has three possible paths, where one path should be executed depending on the results of the boolean expressions `self isName` and `self isBinary`. PerfScope measures the cost of all possible paths, and takes the most costly.

- *Operations.* An operation has an arbitrary and manually set cost which could vary depending on the operands. For instance, + instruction has a cost of 1δ, $*$ has cost of 4δ.

Code Deletion. The model described above deals only with code addition. However, code changes can also be replaced or deleted statements (method calls in the case of Pharo). For PerfScope the cost of a deleted change is

zero and the cost of replacing is only the cost of the new program elements. Because the code deletion cost can offset the total cost, and miss performance regressions. For instance, an inaccurate estimation of the method call deletion can lead to miss costly versions.

Expensiveness Thresholds. Once the cost of the expression is measured, then PerfScope categorizes this expression as extreme, high, moderate or low. This categorization is done using thresholds (in terms of δ). Such thresholds may be computed by running the cost model on the whole program to obtain a cost distribution for functions and basic blocks.

4.1.2. Frequency

PerfScope analyzes the intra-procedural scope that a change lies in. If the change is enclosed in any loop and the *trip count* (maximum number of iterations) of this loop including all its parent loops is statically determined, the execution frequency of this change instruction is estimated by the product of these trip counts. If any enclosing loop has a non-determined trip count, it is considered a frequently executed loop. If a code change lies in recursive functions, it is also considered to be a frequently executed loop.

Frequency Thresholds. Similarly to the instruction cost, PerfScope categorizes the context that may be executed: frequently, normal or rarely executed. This categorization is also done using a thresholds. Such thresholds may be automatically computed by running the frequencies of the whole program [9] .

4.2. PerfScope for Pharo

To compare PerfScope with LITO in the context of dynamic languages (which is the case of Pharo) and under the same benchmark, we implement PerfScope in the Pharo programming language. This subsection describes a number of considerations we take to implement PerfScope in Pharo.

Method Call Expensiveness. Since Pharo is a dynamically typed language, it is difficult to measure the cost of a method call. More precisely, it is difficult to statically detect which method is invoked by just knowing the method name and the arguments. For this reason, if a method call has more than one implementor, we first try to statically infer the possible types of the receiver by checking if the receiver is created in the same method by directly instantiating a class, or in the constructor of the class. If it is not possible and there are multiple possible types for the receiver, then we take the method

with the maximum cost. For instance, consider the method *nodes:using:* of the class *MOViewRenderer*.

```
MOViewRenderer>>nodes: aCollection using: aShape
  | nodes newNode |
  ...
  nodes := aCollection collect:
    [ :anEntity |
    newNode := MONode new.
    newNode model: anEntity.
    ...].
  ^ nodes
```

There are three method calls (*collect:*, *new* and *model:*). The cost of calling the method *collect:* is one (the overhead of the call) plus the cost to execute the method *collect:*. However, there are 22 possible method candidates of collect: and it is not possible to statically detect which of these 22 methods will be called. The method invoked during the execution depends on the class of the object aCollection, typically a subclass of the class Collection. In this case, we take the worst case and assume that the more expensive method will be called. On the other hand, we have the method call *model:*, the receiver of this method call is the variable *newNode*. The type of the variable *newNode* can be statically inferred, because it has been initialized in the same method. In the same fashion, the receiver of the method call *new* can be statically inferred, because the receiver is the same class and *new* is a class method. Note that this is the same criteria that *LITO* uses to look up the information of a method call in the profile history. If there are multiple implementors for a method call, *LITO* takes the worst case.

Operation Expensiveness. Since operations (*i.e.,* $+, -, *, /$) are methods implemented in Pharo with primitives calls, we consider that operations have a constant cost of 1δ, because there is only one message involved to call the primitive. Note that LITO uses a similar criteria.

Loop Frequency. PerfScope determines the trip count only if the loop has a constant bound limit. In the case of Pharo, most if not all loops are associated with object collections, and it is necessary to statically infer the size of the collection to determine the loop boundaries or the trip count.

Statically determining the size of a collection is a difficult task because it can grow and decrease over time. There are a number of expressions where the collection size and therefore the loop boundary can be statically determined.

It usually occurs when the collection is created and used in the same method. For instance:

```
($a to: $z) do:[ :char | ... ].
(0 to: 256) do:[ :number |...].
#($a $e $i $o $u) do:[ :vocal | ...].
```

Similarly to PerfScope we consider particular cases for the static analysis. For our implementation, we consider collections that are created in the same method than the loop, and their size can be statically inferred. We also considered the particular case of the method *timesRepeat*:

```
10 timesRepeat:[  ... ].
```

If it is not possible to statically infer the size, we consider that the loop could be executed frequently. Note that if PerfScope can not estimate the expensiveness or frequency of a code change, it assumes the worst case. Therefore, the change is considered expensive and frequently executed. This is because we cannot afford to miss potential risky versions. However, these situation could trigger a great number of false positives.

Detecting Loops. Another particular problem is that the loops are normally done by sending messages to the collections, where the loop body is passed as parameter. Therefore, it is difficult to determine if a message could be considered a loop. For instance, consider the following two expressions:

```
elements select: [ :each | ... ].
elements at: key ifAbsentPut:[ ... ].
```

Both expressions sent a code block as parameter, however just the first one executes the block a number of times, depending on the size of the collection. For this reason, we need to evaluate if a code block could be executed multiple times. For our implementation, if the method name is registered in the standard protocol "enumerating" of the class Collection (*i.e., do:*, *reject:*), we consider that this message is a loop, and their arguments could be executed multiple times.

4.3. Methodology

For the experiments, we detect performance regression greater than 5 %. We run PerfScope and the Naive approach under the same software versions that we use to evaluate *LITO*. We compare the different approaches in terms of precision, recall, and reduction:

- *Precision*, the percentage of selected versions that introduce a performance regression. For instance, a precision of 50% means that one out of two selected versions introduces a performance regression.

- *Recall*, the percentage of detected performance regressions. To measure this metric, we initially run the benchmarks in all versions and detect which versions introduce a performance regression. This metric measures how many of these versions are selected.

- *Reduction*, the percentage of versions that have not been selected, and therefore the benchmarks were not executed on them. This metric was introduced by Huang *et al.* [9]. The goal of the performance prediction tools is to run the benchmarks in the least amount of versions possible. The *reduction* represents the percentage of versions where no benchmarks is executed. For instance, we could have a reduction of 100% if we do not run the benchmark in any version, but we will not be able to detect any performance regression.

4.4. Baseline for Comparison

We compare our approach with 2 performance prediction approaches: a naive approach, and PerfScope4Pharo.

- *Naive Approach.* It considers that a version is risky if at least one method call was added in the newer version. This naive approach ignores the source code changes done in test methods. In other words, this approach considers any new piece of code as expensive and frequently executed. This approach gives us a baseline to understand the importance of estimating the expensiveness and frequency of source code changes.

- *PerfScope4Pharo.* A PerfScope implementation in Pharo, as described in the previous sections.

PerfScope Thresholds. As we describe in Section Section 4.1, PerfScope uses three thresholds to perform its analysis. For the experiment, we use the following thresholds.

- *Expensiveness thresholds.* Following the line of the PerfScope evaluation [9], we automatically computed these thresholds by running the cost model on the whole program to obtain a cost distribution for functions and basic blocks.

- *Frequency thresholds.* Similar to the previous point, we automatically computed these thresholds by running the frequencies of the whole program [9].

- *Risk_score_threshold.* PerfScope measures the *risk_score* of each software version and uses a *risk_score_threshold* to mark the version as risky or not. Therefore, the precision, recall and reduction of PerfScope can vary depending on the chosen *risk_score_threshold*. To compare PerfScope with the other approaches, we evaluate PerfScope using different *risk_score_threshold* and pick the *risk_score_threshold* that has the better trade-off between reduction and recall.

Risk Score Threshold Computing. We evaluate PerfScope using multiple thresholds. Figure 6 gives the precision, recall and reduction of PerfScope for the different *risk_score_threshold* we used. Figure 6 shows that a greater *risk_score_threshold* leads to a higher reduction, but a lower recall.

In the past PerfScope was evaluated using a $risk\_score\ \_threshold = 200$ which has multiple implications. For instance, a new version will be categorized as risky if it contains two or more source code changes categorized as extreme or 20 changes flagged as high, among other possible combinations. In our case, if we use such threshold PerfScope detect less than the 60% of the performance regressions (Figure 6). Since the goal is to execute the benchmarks in the least number of software versions and detect the greater amount of performance regressions, we pick a $risk\_score\_threshold = 600$ because this *risk_score_threshold* has the better trade-off between reduction and recall.

Note that to get a good *risk_score_threshold* for each project, we had to know in advance which versions introduce a regression or not. In practice, it means that we may need to execute the benchmarks in all versions, which is the problem we are trying to address. In addition, the goal of our comparison is to show that LITO does not need a previous threshold configuration and provide comparable results.

LITO Sample Rate. Contrary to PerfScope, LITO does not depend on thresholds, but on the number of samples collected during the evolution. Figure 7 shows the effect of the sample rate on the precision, recall and reduction. As we showed in previous sections, a sample rate of $k = 20$ (collecting run-time information each 20 versions) has a good trade off between reduction and recall (Figure 7).

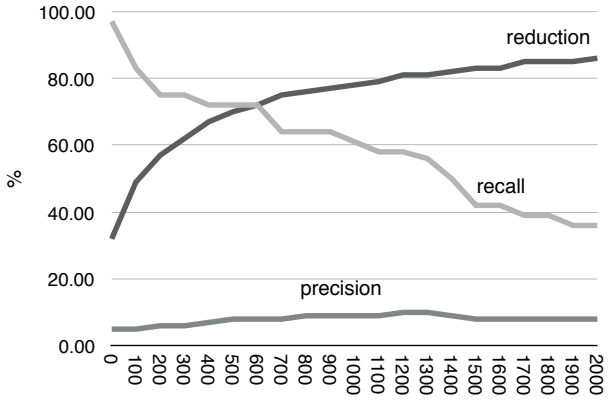

Figure 6: PerfScope - Risk score tuning

However, we argue that even with a "large" sample rate *LITO* can lead to good results. For this reason, we also compare *LITO* with a sample rate of $k = 50$. Using a sample rate of $k = 50$ means that we will take between $1 - 5$ samples depending on the project, having 1.3 samples per project on average. Note that *LITO* needs at least one sample to do the analysis.

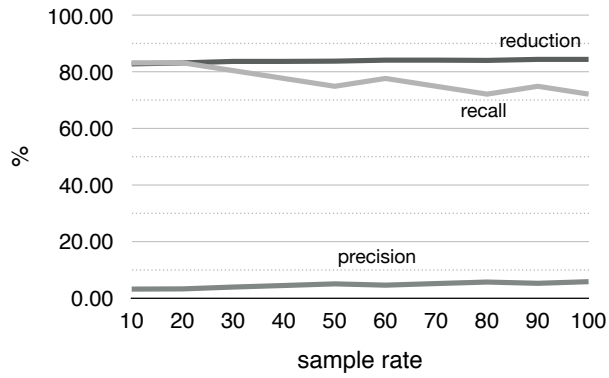

Figure 7: LITO - sample rate effect on the precision, recall and reduction

4.5. Comparison Results

We evaluate PerfScope and the Naive approach under the same 1,125 software versions than LITO. Figure 8 summarizes the results of *LITO*,

PerfScope, and the *Naive* approach. It shows that PerfScope has better precision and reduction than the *Naive* approach, and *LITO* has better precision and reduction than PerfScope, even with a sample-rate of $k = 50$. Although the difference of precision and recall is small between LITO and PerfScope, we have to consider that we compare *LITO* with the best configuration setup of PerfScope ($risk\_score\_threshold = 600$). Note that in practice, finding the optimal $risk\_score\_threshold$ for PerfScope without executing the benchmarks in all versions is a difficult task and it needs to be done manually.

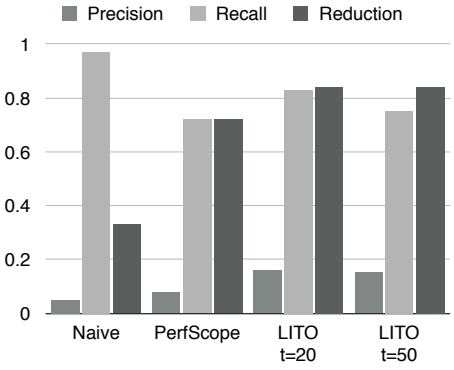

Figure 8: Comparison between different approaches, t = sample-rate

We found that the *Naive* approach discards 33% of software versions and finds 97% of software versions that introduce a performance regression. This means that just focusing the analysis in versions that contain method call additions we are able to reduce the analysis by benchmarking only the 67% of the software versions, and found almost all performance regressions. Note that not all performance regressions are caused by method call additions. As we discussed in the previous sections, a performance regression could be caused by multiple source code changes in the same version. Therefore, any approach that just considers method call additions is not able to detect the 100% of performance regressions. Besides this fact, the *Naive* approach has a better recall than *LITO* and *PerfScope*. However, it has the worst precision and reduction.

Results by Project. We also compare the results of *LITO* and PerfScope at project level. Table 9 gives the results by project. It shows that *LITO* has better precision and reduction in almost all the projects. The exceptions are

Table 9: Precision, Recall and Reduction Comparison of PerfScope (PS) and LITO (LT) using k=20

	Precision(%)			Recall(%)			Reduction(%)		
Project	**PS**	**LT**		**PS**	**LT**		**PS**	**LT**	
Spec	11	19	↑	73	73	=	72	84	↑
Nautilus	6	8	↑	100	100	=	58	68	↑
Mondrian	5	22	↑	100	100	=	71	94	↑
Roassal	7	12	↑	100	100	=	67	82	↑
Morphic	3	13	↑	50	50	=	77	94	↑
GraphET	13	20	↑	20	80	↑	88	71	↓
Rubric	0	0	=	-	-		98	97	↓
XMLSupport	36	50	↑	100	00	=	39	50	↑
Zinc	0	0	=	-	-		78	89	↑
GTInspector	-	100		0	100	↑	100	100	=
Shout	0	-		0	0	=	87	100	↑
Regex	50	100	↑	100	100	=	83	92	↑
NeoCSV	0	0	=	-	-		56	67	↑
NeoJSON	0	-		-	-		86	100	↑
PetitParser	33	100	↑	100	100	=	50	83	↑
Soup	-	-		-	-		100	100	=
XPath	0	-		-	-		50	100	↑

the projects *GraphET* and *Rubric*. In the case of *GraphET*, PerfScope has better reduction, but it has lower recall than *LITO*. In the case of *Rubric*, the difference between reductions is minimal (only 1%).

Figure 9 summarizes the precision, reduction and recall of the different approaches per project. It confirms that *LITO* has better precision and reduction even with a sample rate of $k = 50$. Note that there are a number of metrics that can not be computed depending on the project. For instance, the project *GTInspector* has one performance regression, but PerfScope does not select any versions(because the *risk_score* of all versions was smaller than the *risk_score_threshold*) and the precision of PerfScope can not be computed. On the other hand, *LITO* only selects the version that causes the performance regression and its precision is 100%. Figure 9 compares only the metrics that can be computed by all approaches.

Therefore, we answer our fourth research question as follows:

RQ4: How well does Horizontal Profiling perform compared to the state-of-the-art risk analysis tools using a dynamically typed language? *LITO* estimates with a greater accuracy the expensiveness and frequency of source code changes. As consequence, it has a higher precision and reduction than PerfScope under our benchmarks.

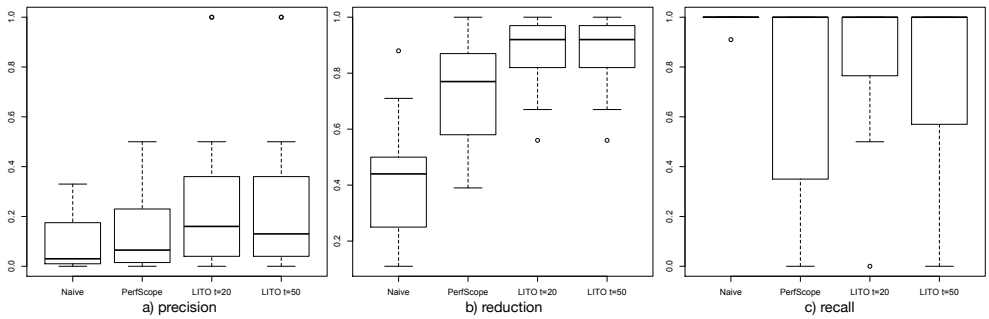

Figure 9: Precision, reduction and recall between different approaches

Table 10: Evaluation of PerfScope4Pharo and LITO over two additional projects (prec=precision, red=reduction, Run.=runnable)

Project	Versions		Reg.	LITO (%)			PerfScope4Pharo (%)		
	Total	Run.		Prec.	Recall	Red.	Prec.	Recall	Red.
Roassal2	100	95	2	22	100	90,53	8	100	72,63
Artefact	100	52	1	9	100	78,84	6	100	65,38

4.6. Evaluation over new projects and versions

In the previous subsection we compare PerfScope4Pharo and LITO over the same versions that we used in the empirical study. Since we designed LITO based on the result of the empirical study, then LITO may be biased in favor of the performance regressions that we manually analyzed. To address this threat to validity, we analyze 200 software versions obtained from two software systems. In this case, we are agnostic about roots of the performance variations. We use the same configuration setups of both tools ($risk\_score\_threshold = 600$ and $sample - rate = 20$). The results (Table 10) shows similar results to our previous experiments. Both tools have a recall of 100%, but LITO has better precision and reduction.

4.7. Discussion

Although, in previous section, we show that LITO has similar results than PerfScope, but without any preliminar configuration. We believe that our experiment also helped us to identify the advantages and disadvantages of each tool in the context of dynamic languages, which we discuss in this section.

LITO Advantages:

- *LITO* uses run-time information of benchmarks past executions. This allows *LITO* to estimate the expensiveness and frequency of source code changes more accurately.

- *LITO* does not need an initial manual tuning. Regardless of the sample rate, *LITO* performs better than PerfScope.

- In comparison to PerfScope, *LITO* considers method call deletions in the cost model. As a result, *LITO* has better reduction than PerfScope. *LITO* can consider method call deletions in the cost model because it uses the number of sent messages to estimate the average execution time. This metric has a small error margin. As a consequence, *LITO* can estimate the trade-off between deletions and additions accurately.

LITO disadvantages:

- *LITO* does not accurately estimate the frequency and expensiveness of new methods and loops, because new source code does not have an execution history. In these cases, *LITO* considers these changes expensive.

- Since *Horizontal Profiling* is a sampling technique, the accuracy of the analysis could change depending on the way that we take samples along software evolution.

PerfScope Disadvantages:

- *Loop Boundaries.* We perform a small experiment with PerfScope, when we use run-time information to categorize the method calls in three categories: expensive, normal, and minor. The results are similar, therefore we conclude that one of the main challenges of PerfScope is to statically detect loop boundaries, which is crucial in this kind of analysis. In fact, PerfScope original implementation is for C/C++, when it is possible to infer loop boundaries in many cases.

- *Method Call Expensiveness.* PerfScope relies on a good categorization of the expensiveness of a new method call. However, in a dynamic language, statically categorize the expensiveness is a difficult activity because the lack of a static type system. As a consequence, it is difficult to categorize a method call as “expensive”,“normal”, or “minor”. As

such, if it is not possible to determine the implementor of a method call, the algorithm takes a worst case marking these method calls normally as "expensive".

- *Risk Score.* Reflecting on the PerfScope cost model, the threshold may be abstracted as how many high, low, moderate and extreme changes should a new version have in order to consider it risky. For instance, a threshold of 200 may be reached with two extreme changes, or with one extreme and two high changes (see section Section 4.1). Therefore, setting an adequate threshold is not an easy task. For instance, only one method call addition may cause a major regression, and many method call additions may not cause a performance difference. In other words, the risk categorization (*i.e.,* method call expensiveness and loop boundaries) plays an important role, but as we discuss in previous points it is not accurate. Another potential issue with the risk score are method call deletions. Essentially, a method call addition may cause a regression and a deletion may cause a speed up. Therefore, since the risk score do not consider method call deletions, it may trigger a number of false positives.

Baseline for Comparison. As we describe previously both approaches have a number disadvantages. We implement the PerfScope risk model in Pharo in order to understand how these disadvantages affect the analysis in practice. Our results show that both approaches are applicable to Pharo and have comparables results. However, we show that the risk model highly depends of the risk threshold. In our experiment, we choose the optimal threshold for all projects. Although it is possible also pick an optimal threshold for each individual project, it would mean that the risk threshold is not generalizable across projects. Moreover, choosing the optimal threshold may involve running all versions, which is not practical.

Other Source Code Changes. In this experiment, we only consider method call additions and deletions. In previous sections, we show that there are different changes that could affect software performance. For instance, changing an object field, an argument or a condition expression. Huang *et al.* [9] propose an extension of PerfScope that implements slicing techniques to detect these kind of changes. This analysis is done after using the analysis with their model, and in independent fashion. We believe that using a slicing technique is still a good option even with *LITO*. We exclude from the comparison the

slicing techniques, because if we use the same slicing infrastructure in both approaches, the comparison results should be similar.

Dynamic and Reflective Features of Programming Languages. Dynamic and statically typed programming languages offer a number of dynamic and reflective features, such as reflection. In the particular case of Smalltalk, everything is done by sending messages, therefore LITO model will consider these cases as a normal method call. However, this may not be the case of other programming languages, such Java o C++. Therefore, there is not evidence how the cost model of LITO or PerfScope will behave with this languages. Our hypothesis is that these features should not affect too much the analysis, because Callau *et al.* shows that dynamic features are not use often in software projects [5]. In addition, they show that these features are mostly used in core libraries, such us programming tools and testing.

Sample rate. We show that with different sample rates LITO has a good precision and recall. However, it is not clear what is the best strategy to select which version of the codebase to run the benchmarks. There are different alternatives to pick samples along the evolution. For instance, random sampling or a rule-based sample strategies. We plan to explore these different sampling strategies as future work.

Continuous Performance Monitoring. This paper does a post mortem analysis of a number of project and software versions to show the effectiveness of our approach. It is relevant to highlight that horizontal profiling may be used together with a continuous integration setup. In this case, we should first collect run-time information on the first commit and then using this information for detect performance regression in the incoming versions. Then this run-time information may also update with different strategies. For instance, after 20 versions as we show in our experiment.

5. Threats to Validity

To structure the threats to validity, we follow the Wohlin *et al.* [23] 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 smaller variations is difficult, because many factors may distort variance to the observable performance, such as inaccuracy of the profiler [2].

External Validity. This paper voluntarily focuses 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 generalize to other software projects beyond the specific scope this study was conducted. As future work, we will replicate our experiments for the JavaScript and Java ecosystem. In addition, we will analyze LITO's performance 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 each method modification twice: the first time to understand the root-cause of the performance variation and the second time to confirm the analysis.

Conclusion Validity. Having a fair comparison between Horizontal Profiling and Perfscope is challenging. We show that Horizontal Profiling has comparable results with PerfScope under our benchmarks, but this difference is not significant. Our experiment reveals a high dependency of PerfScope to the *risk_score_threshold*. Although it is possible to find an optimal threshold configuration to achieve high precision and recall, it may require to execute all benchmarks in all versions which is what we trying to avoid at the beginning. However, our empirical comparison help us to illustrate the advantages of Horizontal Profiling regarding PerfScope in the context of dynamic languages, which is the case of the Pharo programming language.

6. Related Work

Performance Bug Empirical Studies. Empirical studies over performance bug reports [10, 18] 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.* [24] 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.* [17] but they go further by analyzing additional information for the bug reports. Nguyen *et al.* [16] interviewed the performance engineers responsible for an industrial software system, to understand these regression-causes.

Sandoval *et al.* [20] 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 [7, 15, 21]. Jin *et al.* [10] propose a rule-based performance-bug detection using rules implied by patches to found unknown performance problems. Nguyen *et al.* [16] 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.* [3] propose an approach to guide performance optimization processes and to help developers find performance bottlenecks via execution profile comparison. Heger *et al.* [8] 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 [6, 19] and test suite reduction

[4, 11, 13, 25]. However, few projects have been able to reduce the performance regression testing overhead.

Huang *et al.* [9] 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 [22]. 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. We showed that LITO performs better than the state-of-the-art tools.

As future work, we will extend our model to prioritize benchmarks and generalize *Horizontal Profiling* to identify memory and energy performance regressions.

Acknowledgments

We are very grateful to Lam Research for partially sponsoring this work. We also thank the European Smalltalk User Group (www.esug.org) for the sponsoring. This work has been partially sponsored by the STICAmSud project 14STIC-02 and the FONDECYT project 1200067.

Appendix A. Pharo Syntax

Table A.11: Pharo and Java syntax comparison

Pharo	***Java***
element height.	element.height();
element width: defaultWidth.	element.width(defaultWidth);
element width: defaultWidth height: defaultHeight	element widthheight(defaultWidth,defaultHeight);
c := Color new	Color c = new Color();
self	this
super	super
^object	return object
array := #(1 2 3)	Object[] array = {1,2,3};
(1 to:100) do:[:each | ...]	for (int each:= 1; each<=100; each++){ ... }
elements do:[:each | ...]	for (Object each: elements){ ... }

The Pharo model is very close to the one of Python and Ruby. The most relevant syntactic elements of Pharo are: a space between an object and a message name indicates a message send: *object messageName*; the dot separates statements: *statement1. statement2*; square brackets denotes code blocks or anonymous functions: *[statements]*; and single quotes delimit strings: *'a string'*.

References

[1] Alcocer, J.P.S., Bergel, A., Valente, M.T., 2016. Learning from source code history to identify performance failures, in: Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ACM.

[2] Bergel, A., 2011. Counting messages as a proxy for average execution time in pharo, in: Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11), Springer-Verlag. pp. 533–557.

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

[4] Black, J., Melachrinoudis, E., Kaeli, D., 2004. Bi-criteria models for all-uses test suite reduction, in: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society.

Table A.12: Project under Study Description

Project	**Description**
Morphic	It is the Pharo's graphical user interface. It is fully portable between operating systems. It works with graphical objects called Morphs.
Spec	It is a framework in Pharo for defining user interfaces, allowing users to construct a variety of user interfaces. The goal of Spec is to reuse the logic and the visual composition of the widgets.
Rubric	It is a deep refactoring of the Pharo text editor tool. Rubric has improve the performance to edit big texts and facilitates extensions.
Mondrian	It is a library to describe and render graphs.
Nautilus	It is the standard advanced development environment for developing smalltalk programs. Similar to Eclipse or IntelliJ but for Smalltalk.
XMLSupport	It is the standard XML Parser in Pharo.
Roasssal	It is a script based system for creating advanced interactive visualizations. Similar tu d3 in javascript.
Zinc	It is a Smalltalk Framework to deal with HTTP networking protocols.
XPath	It is a XPath Library in Smalltalk.
PetitParser	It is a framework to build parsers in Smalltalk. It allow to dynamically reuse, compose, transform and extend grammars.
GraphET	It is a library for Pharo which provides a builder to create interactive charts.
Shout	It is the default source code highlighting mechanism of Pharo.
Regex	It is a regex library.
Soup	It is a Pharo library for scrapping HTML.
NeoJSON	It is the Pharo standard library for parsing and creating JSON.
GTInspector	It is a moldable domain-aware object inspector.
NeoCSV	It is the Pharo standard library for parsing and creating CSV files
Artefact	It is the Pharo standard library for creating PDF documents.

[5] Callaú, O., Robbes, R., Tanter, E., Röthlisberger, D., 2011. How developers use the dynamic features of programming languages: The case of smalltalk, in: Proceedings of the 8th Working Conference on Mining Software Repositories, ACM, New York, NY, USA. pp. 23–32. URL: http://doi.acm.org/10.1145/1985441.1985448, doi:10.1145/1985441.1985448.

[6] Elbaum, S.G., Malishevsky, A.G., Rothermel, G., 2000. Prioritizing test cases for regression testing, in: International Symposium on Software Testing and Analysis, ACM Press. pp. 102–112.

[7] Han, S., Dang, Y., Ge, S., Zhang, D., Xie, T., 2012. Performance debugging in the large via mining millions of stack traces, in: Proceedings of the 34th International Conference on Software Engineering, IEEE Press.

[8] Heger, C., Happe, J., Farahbod, R., 2013. Automated root cause isolation of performance regressions during software development, in: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, ACM.

[9] Huang, P., Ma, X., Shen, D., Zhou, Y., 2014. Performance regression testing target prioritization via performance risk analysis, in: Proceedings of the 36th International Conference on Software Engineering, ACM.

[10] Jin, G., Song, L., Shi, X., Scherpelz, J., Lu, S., 2012. Understanding and detecting real-world performance bugs. SIGPLAN Notices 47, 77–88.

[11] Kim, J.M., Porter, A., 2002. A history-based test prioritization technique for regression testing in resource constrained environments, in: Proceedings of the 24th International Conference on Software Engineering, ACM.

[12] Kim, J.M., Porter, A., Rothermel, G., 2000. An empirical study of regression test application frequency, in: Proceedings of the 22Nd International Conference on Software Engineering, ACM.

[13] Li, Z., Harman, M., Hierons, R., 2007. Search algorithms for regression test case prioritization. IEEE Transactions on Software Engineering .

[14] Liu, Y., Xu, C., Cheung, S.C., 2014. Characterizing and detecting performance bugs for smartphone applications, in: Proceedings of the 36th International Conference on Software Engineering, ACM.

[15] Maplesden, D., Tempero, E., Hosking, J., Grundy, J., 2015. Performance analysis for object-oriented software: A systematic mapping. Software Engineering, IEEE Transactions on 41, 691–710.

[16] Nguyen, T.H.D., Nagappan, M., Hassan, A.E., Nasser, M., Flora, P., 2014. An industrial case study of automatically identifying performance regression-causes, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM.

[17] Nistor, A., Jiang, T., Tan, L., 2013a. Discovering, reporting, and fixing performance bugs, in: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press.

[18] Nistor, A., Song, L., Marinov, D., Lu, S., 2013b. Toddler: Detecting performance problems via similar memory-access patterns, in: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press.

[19] Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., 1999. Test case prioritization: An empirical study, in: Proceedings ICSM 1999.

[20] Sandoval Alcocer, J.P., Bergel, A., 2015. Tracking down performance variation against source code evolution, in: Proceedings of the 11th Symposium on Dynamic Languages, ACM.

[21] Shen, D., Luo, Q., Poshyvanyk, D., Grechanik, M., 2015. Automating performance bottleneck detection using search-based application profiling, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ACM.

[22] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P., 2008. The worst-case execution-time problem; overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. .

[23] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering. Kluwer Academic Publishers.

[24] Zaman, S., Adams, B., Hassan, A.E., 2012. A qualitative study on performance bugs, in: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, IEEE Press.

[25] Zhong, H., Zhang, L., Mei, H., 2006. An experimental comparison of four test suite reduction techniques, in: Proceedings of the 28th International Conference on Software Engineering, ACM.