Tracking Down Performance Variation against Source Code Evolution

Juan Pablo Sandoval Alcocer, Alexandre Bergel
PLEIAD Lab, Department of Computer Science (DCC), University of Chile, Chile
{jsandova,abergel}@dcc.uchile.cl

Abstract

Little is known about how software performance evolves across software revisions. The severity of this situation is high since (i) most performance variations seem to happen accidentally and (ii) addressing a performance regression is challenging, especially when functional code is stacked on it.

This paper reports an empirical study on the performance evolution of 19 applications, totaling over 19 MLOC. It took 52 days to run our 49 benchmarks. By relating performance variation with source code revisions, we found out that: (i) 1 out of every 3 application revisions introduces a performance variation, (ii) performance variations may be classified into 9 patterns, (iii) the most prominent cause of performance regression involves loops and collections. We carefully describe the patterns we identified, and detail how we addressed the numerous challenges we faced to complete our experiment.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Testing and Debugging; D.4.8 [*Operating Systems*]: Performance

General Terms Languages, Measurement, Performance, Experimentation

Keywords Performance Variation, Performance Analysis, Performance Evolution

1. Introduction

> *"A program that is used in a real-world environment necessarily must change, or it becomes progressively less useful in that environment"* [10].

Programmers make many changes to a program to eventually find a good solution for a given task (i.e., fixing a bug, adding a feature, optimizing) [18]. Continuous software changes may inadvertently introduce a drop in performance at runtime. It is often perceived that the longer the performance loss remains undiscovered, the harder it is to remove the loss. Performance regression testing is an effective way to identify performance regressions in early stages. Such regressions refer to situations where software performance degrades in comparison to previous releases, though the new version behaves correctly [11]. Despite the numerous solutions proposed by the software engineering community to monitor performance evolution[1], little is known about the force behind source code revisions that trigger performance variations.

This paper conducts a comprehensive study of the performance evolution of 49 benchmarks along the evolution of 1,439 versions from a variety of 19 open source software projects, in order to validate, or not, the following hypothesis:

H- *Program performance is mostly affected by identified recurrent source code changes across software revisions*. More precisely, we are interested in determining how source code changes mostly affect program performance along software evolution. Such hypothesis is relevant to predict risky commits [7, 14].

Pharo Ecosystem. Verifying this hypothesis involves many considerations which are technical (*e.g.,* identifying performance variations that are reproducible and significant) and practical (*e.g.,* identifying a set of benchmarks that are executable for a large number of software revisions). Whereas our long-term goal is to understand performance evolution in general, we focus our analysis on the Pharo ecosystem. Pharo[2] is an emerging language programming environment which is dynamically typed, Smalltalk-inspired, and has a syntax close to that of Ruby and Objective-C. Pharo offers the necessary tooling, hooks, and an expressive reflective API, to measure time consumption of a benchmark set over hundreds of software versions [1]. In addition, the Pharo community is

DLS'15, October 27, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3690-1/15/10...$15.00
http://dx.doi.org/10.1145/2816707.2816718

[1] Several plugins are available for Jenkins, `https://jenkins-ci.org`.

[2] `http://pharo.org`

friendly and easily reachable, which is crucial when authors have to be contacted.

Findings. Our experiment revealed a number of facts for the Pharo applications we have considered:

- Only 2% of the application revisions have a commit message related to performance, which strongly indicates that developers are either not aware when they introduce a performance regression or improvement or performance is not within the scope of developer activities.
- Roughly 1 out of 3 application revisions introduce a performance variation ($\geq 1\%$ or $\leq -1\%$), and roughly half of these performance variations are regressions and half are improvements.
- The most prominent cause of performance regression is composing collection operations, such as filtering, mapping, and iteration.

Outline. Section 2 discusses the relevance to monitor performance variations. Section 3 describes the methodology we have adopted to answer the hypothesis formulated above. Section 4 presents the results of our experiment. Section 5 lists the performance regression patterns we have identified while Section 6 lists the performance improvement patterns. Section 7 discusses the threats to validity we are facing and how we are addressing them. Section 8 gives a brief overview of the related work. Section 9 concludes and presents an overview of our future work.

2. Imperceptible Performance Degradation

Our effort is motivated by the lack of adequate tools to monitor and compare multiple execution profiles across software revisions. A question naturally arises: *Are developers aware of the impact of their code revision on the system performance?* Answering this question may be the first step to understanding how performance regression occurs in practice. Our intuition tells us that developers are aware of abrupt performance variations, while a slow degradation remains largely imperceptible.

Commit messages are a great opportunity for developers to express themselves when committing a new application revision. Commit messages are therefore a natural source of information to understand whether or not developers are aware of performance variation when committing. We manually analyzed the commits of 1,419 software revisions. After discarding 140 (9.86%) empty commit messages, we find out that only 33 (2.58%) of the commit messages are somehow related to performance. We searched for the vocabulary commonly associated with performance[3] and references to bug fixes related to performance. As a comparison, 796 (62.23%) messages discuss functional software properties or related to improvement[4].

Our goal here is not to conduct a rigorous analysis of revision commit messages, but rather to offer a glimpse of the extent of the problem regarding hidden performance variations. Note that we are not blaming Pharo developers for this situation. Manually keeping track of performance evolution is a tremendous task that requires discipline and a dedicated infrastructure. Unfortunately, these two requirements are not a priority within requirements set by the clients.

3. Methodology

3.1 Challenges

Identifying performance variations over multiple software revisions is a difficult task. Several challenges have to be addressed to identify variations that are trustworthy, reliable and meaningful:

- *Minimizing measurement bias* – Measuring execution time is a delicate operation that is highly sensitive to external factors [12].
- *Identifying benchmarks* – Each application has to provide a set of benchmarks that are sufficiently robust against API variation that may occur during the application history. Identical benchmarks have to run through a large and significant number of software revisions.
- *Identifying performance variation* – Identifying a performance variation is not trivial because of the multiple configurations that may occur (*e.g.,* a software revision may improve the performance of particular benchmarks while degrading performance of others).
- *Using one unique runtime* – Only one runtime should be used to avoid measurement bias that may be due to the use of different execution runtimes.

3.2 Workflow

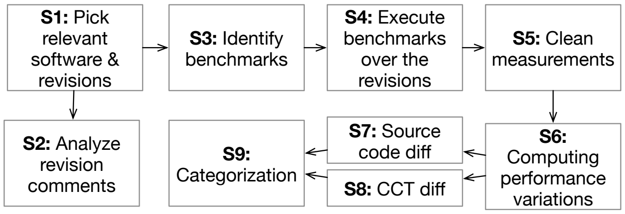

S1: Pick relevant software & revisions
S3: Identify benchmarks
S4: Execute benchmarks over the revisions
S5: Clean measurements
S2: Analyze revision comments
S9: Categorization
S7: Source code diff
S8: CCT diff
S6: Computing performance variations

Figure 1. Methodology structured around a 9 steps workflow.

To verify (or invalidate) the hypothesis given above (Section 1), we have designed a methodology and structured it along a dedicated workflow (Figure 1). The workflow contains 9 different steps:

[3] we considered the words: “speed”, “slow”, “performance”, “fast”.

[4] we considered the words “bug”, “fixes”, “refactoring”, “don’t work”, “does not work”, “failing test”.

- S1 - Our very first step is to collect a number of relevant applications. Each has to come with a number of application revisions. A critical point is to make sure all the revisions behave correctly and as expected under a same execution environment (virtual machine, runtime, essential libraries). We use a trial-and-error approach.
- S2 - We analyze the revision comments to verify whether or not authors are aware of a severe performance variation they introduce.
- S3 - For the next step, a number of benchmarks have to be identified. These benchmarks have to be produced by the software authors or members of the Pharo community. The benchmarks have to be executable for a significant portion of the software revisions, therefore they should be resistant against API changes across revisions.
- S4 - Getting relevant and trustworthy measurements by executing the benchmarks over the revisions is delicate. Understanding possible external factors to minimize measure bias [12] is key to making our result sound.
- S5 - Measurements obtained from S4 may have to be cleaned since not all the obtained measurements are exploitable. For example, a particular software revision may be broken or significantly deviate in its execution from the main execution line. Such situations have to be identified and manually inspected.
- S6 - We compute the performance variations.
- S7 - Identifying the source of performance evolution requires to differentiating two source code revisions. This step is crucial since it indicates the candidate of elementary source code changes causing the performance variation.
- S8 - We compute the difference in the execution path. Computing execution path difference complements the source code comparison. This is necessary to identify the exact source of a performance variation. For example, a revision may change several methods, and identifying which of these methods causes a performance variation requires differentiating the execution profile. We therefore compare calling context tree (CCT) [21] to estimate the differences between the execution paths.
- S9 - Source code revisions are then categorized based on their differences. Combining the difference of the source code and execution is key to precisely understanding a performance variation.

The remainder of this section elaborates on some of the steps described above.

3.3 Projects under Study (Step S1)

For this study, we have chosen a variety of 19 software projects from the Pharo ecosystem. These software applications are intensively supported by the Pharo community. Their sizes range from 5 KLOC to 8.86 MLOC (including revisions). Project size is obtained by summing all the revisions' churn. These software applications are central in many software development activities and supported by an active community that is likely to be useful whenever the author or the informal history of these software projects is consulted. Table 1 summarizes these projects. The table gives the total lines of code and the number of method modifications (M.M.) along software versions of each project. The last row shows the total for the 19 projects analyzed in this study. In total, we have analyzed 1,439 software versions that contain 19,505,646 lines of code and 40,783 method modifications. On average, each version has 28 method modifications. Table 1 also shows the average number of classes (NOC) and methods (NOM) as an estimation of the project size. Other columns of the table are described below. For the sake of making our result reproducible, the version numbers of the applications we have analyzed are available online[5].

Table 1. Projects, number of analyzed release versions (Vers), number of method modifications (MM), number of lines of code (LOC), number of classes (NOC), number of methods (NOM)

Project	Vers	Total		Average		
		LOC	MM	LOC	NOC	NOM
Morphic	214	8,860,430	1,808	41,404	270	6,897
Mondrian	150	1,822,350	2,318	12,149	205	1,776
Nautilus	214	2,370,443	1,253	11,077	145	1,710
Spec	270	2,932,847	6,320	10,863	261	2,320
Rubric	156	1,651,184	12,996	10,043	109	1,775
NeoCSV	10	80,924	99	8,093	8	114
Zinc	21	137,472	692	6,547	132	1,457
Roassal	150	952,037	8,081	6,347	173	1,199
AST	61	387,350	981	6,350	126	1,512
Regex	13	52,772	35	4,060	37	299
XMLSupport	22	72,006	2,875	3,273	76	974
Shout	16	36,414	100	2,276	17	296
PetitParser	7	14,076	288	2,011	57	452
Soup	6	9,631	62	1,606	23	262
XPath	10	13,661	1,233	1,367	33	335
GraphET	82	89,648	1,462	1,094	32	247
Announcements	12	5,521	23	461	18	127
NeoJSON	8	5,594	15	700	16	173
GTInspector	17	11,296	142	665	15	114
Total	1,439	19,505,646	40,783	130,386	1,753	22,039

3.4 Identifying Benchmarks (Step S3)

A benchmark is a repeatable and measurable application execution and is likely to describe a common and relevant usage scenario of an application. We use benchmarks across a large portion of the history of each application to measure the performance variation of the application.

The Pharo ecosystem does not contain widely accepted benchmarks, such as DaCaPo and SpecCPU [3, 9]. We therefore have to constitute a reliable benchmark.

The applications we selected are considered significant for the Pharo community. They are often publicly discussed, multi-authored (ranging from 2 to 446 authors), and have

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

been under steady improvement over a long period of time. An application is often packaged with some benchmarks. We have revised those to exclude micro-benchmarks. We are interested in large and long-running benchmarks. Micro-benchmarks have a particular purpose that largely differ from the aim of macro-benchmarks. For example, a micro-benchmark may assess the effect of the just-in-time compiler or the garbage collector for a very particular situation. Since our objective encompasses general performance evolution, we solely focus on macro-benchmarks. In case no benchmark were found for an application, we directly contacted the authors to obtain benchmarks.

The benchmarks per application are stable, which means the very same benchmarks have to be run over all the application versions. Such constraints lead to a challenge: some of the benchmarks of an application could be executed or meaningful only for a portion of the application history. It may happen that a benchmark uses a feature that has been recently introduced. This means that the benchmark cannot be executed for older versions of the application since the feature did not exist. Such benchmarks are not relevant for our experiment since having the same benchmark is a requirement. In such a case, we had to rework the benchmark to make sure the recently introduced feature is not used.

Addressing this problem was particularly time consuming since we had to go over a sequence of try-fix-repeat. In total, we have 49 benchmarks [6], with an average of 3 per software application.

3.5 Executing the Benchmarks (Step S4)

Measuring the execution time of a benchmark is a complex operation. One of the main difficulties when measuring execution time is avoiding (or reducing) measurement bias. A bias may lead to an unfounded or even wrong sense of the actual performance. Avoiding measurement bias when profiling an execution is a well known problem among the software performance community [6, 9, 12, 13].

To avoid distortion in our measurements, we adopted the following three actions:

- *Counting messages* - Execution time estimation, as produced by traditional code execution profiler, is highly sensitive to the execution environment, making it non reproductive, non-deterministic and not comparable across different execution platforms [2]. It has been shown that counting messages is a reliable proxy for execution time in Pharo [2]. Pharo is an object-oriented language making extensive use of message sends: most of the computation is carried out by sending messages, including control structure and loop handling. Such an homogeneous execution platform offers the interesting property to correlate execution time with message sends. Message counting is more stable across multiple executions than time estimation. We therefore exploit this property in our experiment.
- *Warm-up* - Correlation between message counting and execution time is increased by adequately warming up the environment before executing the benchmark. We perform such a warm-up by executing a benchmark a couple of times before starting the measurements.
- *Multiple executions* - Counting messages is robust against multiple executions. This means that executing a same piece of code twice is likely to result in a very similar number of message sends. Difference in the number of message sends across multiple executions is due to hash values (which are non-deterministically generated by the virtual machine in the case of Pharo) and the use of hash-based abstract data types, such as dictionaries and hash tables [2]. To minimize the variation between multiple executions, we execute each benchmark 10 times (after having warmed up the execution environment) and compute the average, which we refer to as $\mu[v_i, b_j]$, for version v_i and benchmark b_j.

We have monitored the performance evolution of the 49 benchmarks along the software versions of the projects under study. We developed a tool[7] to execute the benchmarks in each software versions. It took 52 days to run our benchmarks. The experiment was conducted on a MacBook Pro 2.8 GHz Intel Core i7 with OSX 10.9.1 and 6GB 1333 MHz DDR3. Counting messages has the benefit of being stable across different operating systems, which means that even if our experiment have been carried out on a MacBook, our findings will be relevant for Windows and Linux [2].

We measure the error margin of all benchmarks with a confidence level of 90 %. Since the benchmarks have different number of sent messages (ones are greater than others), we use the ratio of the error margin to the mean, to be able to compare the different error margins. We found that almost all benchmarks have a small error margin ($< 1\%$). A very few benchmarks have an error margin close to 2%: these benchmarks use has values making them non-deterministic (*e.g.,* when sets and dictionaries are employed) to store their model. Multiple executions of our benchmarks have resulted in an error margin of $\pm 0.89\%$ (on average). To not misinterpret an error as a small measurement, we excluded all the performance variations within the range $\pm 1\%$.

3.6 Computing Performance Variation (Step S6)

Consider two successive versions, v_i and v_{i-1} of a software application A and a benchmark b_j. We have i ranging from 6 to 214 depending on the considered application. The Soup application has 6 source code revisions while Morphic has 214 revisions. We also have j ranging from 1 to 6. For example, for the AST application we have only 1 benchmark while we have 6 benchmarks for Roassal.

For an application A, we define the variation between versions v_i and v_{i-1} for a given benchmark b_j as:

[6] Available at `http://tinyurl.com/exp-benchmarks`

[7] `http://smalltalkhub.com/#!/~juampi/Hydra`

$$V_A[v_i, b_j] = \frac{\mu[v_i, b_j] - \mu[v_{i-1}, b_j]}{\mu[v_{i-1}, b_j]} \tag{1}$$

In case that $V_A[v_i, b_j] > 0$, benchmark b_j takes longer to execute for version v_i when compared to version v_{i-1}, which indicates a performance regression. Having $V_A[v_i, b_j] < 0$ indicates that the benchmark executed faster, therefore resulting in a performance improvement.

3.7 Categorizing Performance Variations (Step S9)

Step S9 categorizes source code revisions triggering a performance variation. This is a crucial step to understanding the cause of performance variations. Step S9 is the last step of our workflow.

Step S6 computes (i) performance variations for benchmarks and software versions and (ii) it indicates whether a particular benchmark is slower or faster in a particular software revision. Step S7 identifies what the differences are in the source code of two software revisions. In particular, S7 outputs the list of classes and methods that have been modified between two software revisions, v_i and v_{i-1}, for which a benchmark b_j takes more or less time to execute in v_i than in v_{i-1}. The revision between v_i and v_{i-1} may have a large number of class and method modifications. Not all the modifications in a revision are responsible for the performance variation. Understanding which of the elementary changes contained in a revision leads to a performance variation requires comparing the two execution profiles, which is the objective of Step S8.

Step 9 identifies the exact source code modification that leads to the performance variation by manually contrasting source code changes and execution differences between two software revisions. For example, code reformatting, code comment addition, and dead code removal are likely to be excluded at that stage since they usually do not have an impact on performance variation.

4. Overall Results

In total we have gathered 1,439 different software revisions from two Pharo source forges[8] and 49 benchmarks. Running these benchmarks on all the software revisions (Step S4) reveals that 1,248 (87%) of the versions execute at least 1 benchmark and 191 (13%) versions were unable to run a benchmark.

Failing Benchmark Execution. The cause for a software revision to fail to run a particular benchmark is multiple: a revision may capture only a portion a larger software change, implying that the revision may be broken; a benchmark may be meaningless for a particular software revision, for example if a revision implements a configuration management and the benchmark does not set any. We have excluded these 191 revisions from our analysis. Non-working versions are simply left out during the variation computation and that the variation between the last running version before and the first version running after the defective version is computed.

Performance Evolution and Revision. Step S6 indicates revisions v_i that increase or reduce the execution time of benchmark b_j when compared with v_{i-1}, the immediate previous revision. We have categorized each software revision v_i into one of the four categories:

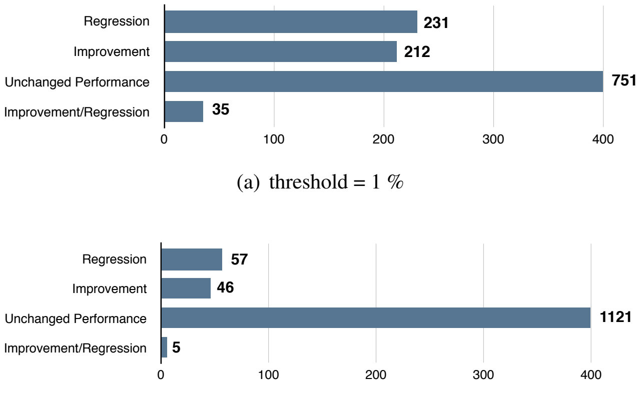
- *Performance regression* – the revision increases the execution time of some benchmarks executed on the revision without decreasing any. This means $V_A[v_i, b_j] \geq 0$ for all the executable benchmarks b_j of the application. At least one benchmark has a variation greater than 1% ($V_A[v_i, b_k] \geq 0.01$). The software revision is therefore said to be a performance regression since at least one benchmark executes more slowly.
- *Performance improvement* – the revision decreases the execution time of some benchmarks b_j executed on the revision ($V_A[v_i, b_j] \leq 0$). At least one benchmark has a variation less than -1% ($V_A[v_i, b_k] \leq -0.01$). By speeding up at least one benchmark without slowing down any, the software revision is a performance improvement.
- *Unchanged performance* – the revision keeps the benchmark execution time unmodified (*i.e.,* within the range $\pm 1\%$). A typical example of such a situation is (i) when the revision adds comments without modifying an executable portion of the application or (ii) when fixing typos in classes or method names.
- *Improvement and regression* – the revision improves the execution of some benchmarks and decreases the execution of some other benchmarks. A typical situation is when two benchmarks involve evolving distinct features.

Figure 2 reports our finding. We report the number of versions according to two threshold, 1% and 5%. We have 1,248 revisions for which benchmarks may be executed on. The four values reported in the figure total 1,229. The first revision of the 19 applications are therefore excluded since the first application commit cannot be associated with a performance variation. We have adopted a threshold of 1%. A performance variation within the range (-1%; +1%) is considered as constant.

We found that 231 (18.80%) revisions are performance regressions. We also found that, 212 (17.25%) revisions introduce a global improvement. These results shows that there are slightly more commits introducing a slowdown than a speedup.

A total of 751 (61.11%) revisions do not modify the benchmark execution time, which means that 478 (38.90%) revisions have an impact on the application. Such a result is significant: about 1 revision every 3 revisions introduces a performance variation.

[8] `http://smalltalkhub.com`, `http://squeaksource.com`



(a) threshold = 1 %

(b) threshold = 5 %

Figure 2. Number of versions that i) cause a performance regression in at least one benchmark ii) cause a performance improvement in at least one benchmark iii) do not change the performance of any benchmark and iv) cause a performance regression in one benchmark and a performance improvement in another benchmark.

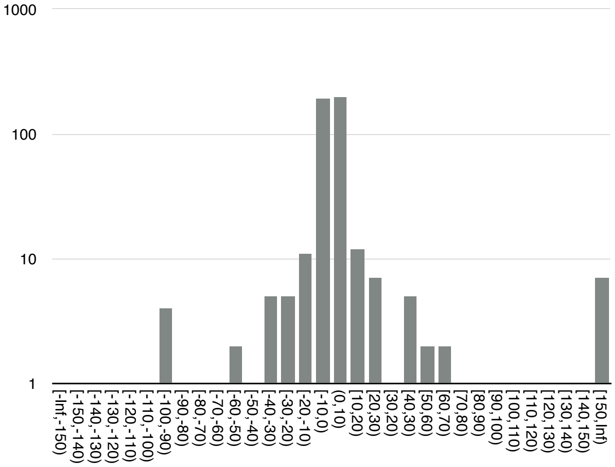


Figure 3. Number of Performance Variations $V[v_i, b_{max}]$ of all versions (%), where b_{max} is the benchmark with most variation in that version (Y log scale).

A total of 35 application revisions are both a regression and an improvement: at least 1 benchmark has its execution time increased while another has it decreased. Interestingly, these 35 revisions comprise 3 UI frameworks: Spec, Morphic and Roassal.

Performance Variation Distribution. Figure 3 gives the performance variation distribution. The bar chart indicates the amount of revisions for each range of performance variations. To cope with large disparities between values, we use a logarithmic scale for the Y-axis. We have reported the greatest benchmark regression or improvement for each version. Adding up the bar totals 1,229, the number of versions as reported in Figure 2. Variations within our error margin are not reported. The distribution is not normal, even with a logarithm transformation and excluding the outlier values.

The chart indicates that most of revisions have a relatively small variation, within a range of -10% and +10%. The [-100,-90) performance variation range contains 4 major speedups, involving the Spec, Nautilus, Roassal, Zinc applications. A close look at the source code variation related to 3 of these speedups are the consequence of having adopted a better algorithm and the 4th speedup is due to an improvement of the dependent API (Pattern P9, described in Section 6).

The [150,inf) range indicates 7 severe slowdowns, involving the Spec, Roassal, PetitParser applications. Within these 7 slowdowns, 3 are caused by composing collection operations (Pattern P1, Section 5), another 3 are due to domain related optimization, and 1 due to an object proxy introduction (Pattern P3, Section 5).

5. Categorizing Performance Regressions

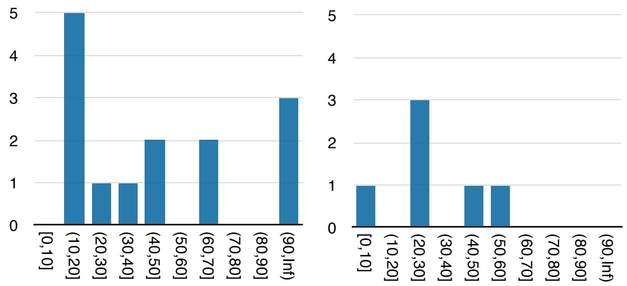
By contrasting difference of source code (Step S7) and variation in the execution path and calling-context trees (Step S8), we have identified 55 performance variations of more than +5% and less than -5%. This section discusses the 39 performance variations greater than +5% found in the 1,248 application revisions. Some of these 39 performance regressions describe recurrent situations, which is the topic of this section of the paper. Due to the precision of the calling-context tree analysis, we could not go get a precision better than 5%.

Measuring small performance variations is difficult because a profiler introduces a bias when gathered run-time information. This effect is know as the observer effect. We only consider variations are greater than 5 % and lower than -5%.

Table 2. Patterns of performance variations

Performance Regressions	**39(100%)**
P1: Composing Collection Operations	14 (36%)
P2: Excessive UI Update	6 (15%)
P3: Introducing Object Proxy	4 (10%)
P4: Regression in Dependent Project	4 (10%)
P5: Heavy Object Construction	3 (7.5%)
Others	8 (20%)
Performance Improvements	**16(100%)**
P6: Deleting Redundant Method Call	4 (25%)
P7: Cache Introduction	3 (19%)
P8: Conditional Statements Addition	2 (13%)
P9: Improvement in Dependent Project	2 (13%)
Others	5 (31%)

Table 2 list of patterns with their occurrence in the set of application variations we have considered.



(a) Composing Collection Operations

(b) Excessive UI Update

Figure 4. Performance Variation Distribution of Patterns P1 and P2

Each performance regression pattern has a title, a description and one or more examples taken from the examined application revisions. Each example describes the source code evolution of one or more methods. A source code line with the leading – indicates the line was removed in the revision while the leading + indicates the line was added. Unmarked lines (without leading - or +) are found in both versions.

P1: Composing Collection Operations. It is known that abstractions for data collection play a significant role in application performance [4, 17]. This situation is exacerbated in Pharo since loops and iterations are operations performed on collections.

The pattern *composing collection operations* involves the combination of collections operations typically filtering, mapping, and iterating. The intuition behind the pattern is that composing a collection operation on top of another operation may contribute to degrade performance.

As an example, consider the modification of the on: method in the Roassal application:

```
ROAdjustSizeOfNesting class>>on: element
   ..
–  element do: [ :el |  ...].
+  element elementsNotEdge do: [ :el |  ...].

+ROElement>>elementsNotEdge
+  ^ elements reject: #isEdge
```

The method elementsNotEdge and the call to it was added in the revision of the method on:. In the revision, the expression element do: [:el |...], which iterates over the collection contained in the variable element, has been replaced by element elementsNotEdge do: [:el |...], which first filter out some elements from the collection and then perform the iteration. Even though the do: operation is carried out on a subset of the original collection, the cost of filtering before iterating is greater than simply iterating. The method on: is invoked many times in one benchmark, causing a regression of 39%.

The pattern P1 occurs 14 times (35.89%) in our set of 39 performance regressions. Figure 4, left side, gives the distribution of the performance variations due to P1. Although there is no reason to expect a particular range of performance regression to be associated with P1, one can notice that regressions due to this pattern are likely to be severe.

P2: Excessive UI Update. Graphical user interfaces often have to be updated whenever the object model behind the interface is updated. For example, if an offline server goes online, the status icon in the monitoring UI interface may have to go green. However, updating the whole UI instead of the icon status may be the cause of serious performance regressions.

The pattern *excessive UI update* refers to a source code revision that introduces a redraw, relayout or rebuild of the user interface, resulting in a negative performance variation. Such operations may be expensive, directly depending on the number and complexity of the inner visual components.

For example, consider the modification made in the extent: method in the Morphic application. Developers have replaced the call to updateSliderBounds by removeAllMorphs and initializeSlider, because the method updateSliderBounds did not work as expected in a number of scenarios. This issue was fixed by rebuilding all the Slider inner-components, causing a performance regression of 21%.

```
Slider>>extent: newExtent
   ...
–  self updateSliderBounds
+  self removeAllMorphs; initializeSlider
```

Resetting the whole UI is apparently perceived as an easier solution than updating what is strictly necessary. This pattern occurs 6 times involving 4 of the 7 applications that provide a graphical user interface. Figure 4, right side, gives the distribution of the performance variations due to the pattern. One will notice that a performance regression due to P2 is likely to be less severe than when due to P1.

P3: Introducing Object Proxy. The Adapter and Proxy are two design patterns [5] commonly employed whenever messages received by an object have to be intercepted. The purpose of such interception may be multiple: logging or checking particular calls made on the wrapped object are frequent situations.

The pattern *introducing object proxy* refers to the situation when wrapping an object that receives a large number of messages may introduce a negative performance regression. Naturally, the regression depends on how costly the new behavior added by the adapted / proxy is.

For instance, consider the following stream adapter added in the XMLSupport application:

```
+XMLPeekableStreamAdapter>>next
+  | nextChar |
+  peekChar
+     ifNotNil: [nextChar := peekChar. peekChar := nil]
+     ifNil: [ nextChar := stream atEnd
+           ifTrue: [nil]
+           ifFalse: [stream next]].
+  ^ nextChar.
```

The method next defined on the class XMLPeekableStreamAdapter adds extra checks before performing stream next. Since a stream objects in XMLSupport receives a large number of messages within the benchmarks, this change causes a performance regression of 16%.

In total, this pattern occurred 4 times (10%) within the software revisions we considered in our experiment.

P4: Regression in Dependent Project. Applications rarely live on their own. Instead, an application often depends on other applications, typically libraries and frameworks. Consider a dependency $A1 \rightarrow A2$ between two applications $A1$ and $A2$. Beside outsourcing functionalities, the dependencies may result in a performance variation in $A1$ in case of source code modification in $A2$.

The pattern *regression in dependent project* involves a revision of a dependent application causing a performance regression in the depending application.

For example, Nautilus, a code navigation browser, depends on ClassOrganizer, a reflective API to query the structure of classes and packages. Nautilus uses ClassOrganizer to let users navigate through Pharo classes and packages. ClassOrganizer went through a heavy refactoring. Although functionalities were fully preserved, more resources are necessary, thus negatively impacting Nautilus by 100%.

We have seen 4 occurrences of this pattern (10% of the total of performance regressions).

P5: Heavy Object Construction. The purpose of a class constructor is to ensure an object is properly defined when created. For example, when creating an instance of a class Point, the constructor of that class will initialize the x and y variables to 0. In the case of the class Point, the initialization of an object is pretty light since assigning 0 is a cheap operation. However, the initialization may create other objects, which themselves have to be initialized, thus creating of chain of object initializations.

The pattern *heavy object construction* involves an expensive chain of object initializations. Such a chain may be expensive due to its length (*e.g.,* initializing an object o_n requires initializing o_{n+1} first) or its width (*e.g.,* initializing an object o requires creating and initializing objects $o_1 ... o_n$). Such a chain of object initializations and creations chain may have a negative performance impact, especially in the presence of a high number of objects.

For example, consider the modification of the constructor initialize of the class GETDataObject from the GraphET application:

```
GETDataObject>>initialize
+ self roElement: ROElement new.
```

The revision of the constructor initialize introduces the creation of an object ROElement, a complex object with 12 instance variables, each being initialized again, with large objects. Since the class GETDataObject is central in GraphET and massively instantiated by our benchmark, the constructor revision introduced a performance regression of 11%.

This pattern occurred 3 times (7.5%) within our set of identified performance regressions.

Outliers. We found 8 (20%) singular cases causing performance regressions. No commonalities were identified between these regressions and seem to be due to adding new features or improving existing features.

For instance, the caching support in Mondrian was causing a 40% regression with the following method:

```
MOGraphElement>>resetMetricCaches
− cache := nil.
+ self removeAttributesMatching: 'cache*'
```

Since resetMetricCaches is frequently invoked, the incurred regression is significant.

6. Categorizing Performance Improvements

From the 1,248 analyzed versions we found that 16 introduce a negative performance variation, representing a performance improvement. There are therefore more revisions that introduce a performance regression than an improvement. This section revises the source code revisions that lead to performance improvements.

P6: Deleting Redundant Method Call. Removing an unnecessary computation is probably the most effective way to improve performance.

The pattern *deleting redundant method call* identifies source code revisions that remove redundant method calls. We qualify a method call as redundant if it does an unnecessary computation. Deleting redundant method calls are likely to improve the performance of a benchmark, especially when such calls are heavily involved in the overall computation.

For example, consider the modification made in the method nodes:forEach: in the Roassal application:

```
ROMondrianViewBuilder>>nodes: objects forEach: aBlock
  ...
  nodes do: [ :n |
     ...
−    self applyLayout.
  ].
```

The revision removes the applyLayout call, likely to be a redundant message. In the new method version, the layout is applied only once, after the graph has been properly defined.

We have counted 4 occurrences of this pattern in the 15 performance improvements we have identified.

P7: Cache Introduction. Preventing the same computation to be realized more than once is typically done by caching the first-time-computed result. Applying a memorization technique [19] under certain conditions eliminates this redundancy.

The pattern *cache introduction* identifies source code revisions that cache computed values, thus avoiding unnecessary repetitive computation.

For example, an optimization was made in the NeoCSV application by introducing a cache initialization in the peekChar method. The cache is reset in nextChar:

```
+ NeoCSVReader>>peekChar
+  ^ charBuffer
+     ifNil: [ charBuffer := readStream next ].
+ NeoCSVReader>>nextChar
+  ^ charBuffer
+     ifNil: [  readStream next ]
+     ifNotNil: [ | char |
+        char := charBuffer.
+        charBuffer := nil. ^ char ]
```

This pattern occurred 3 times (19%) in the analyzed software source code revisions.

P8: Conditional Statements Addition. Identifying a particular situation prior to carrying out a computation may be effective in avoiding such computation.

The pattern *conditional statements addition* identifies a revision that introduce a conditional statement to avoid unnecessary computation. Such conditional statements usually check for *singular values* such as an empty collection, a negative numerical value or a null value.

For example, an early method exit, implemented using a conditional statement was introduced in the method displayIn: in GraphET:

```
GETAbstractDiagram>>displayIn: aView
  self generateIn: aView.
+  self hasValues ifFalse: [ ^ self ].
  self createAxis: aView.
  self addInteractions
```

The condition simply checks on whether some values have been added in an instance of the class GETAbstractDiagram. If no value has been added then the method exits early, using the instruction ^ self.

Interestingly, Jin *et al.* [8] present a case study and describe how a conditional break introduces a slowdown, because most of the time the condition was evaluated false in their monitored situations. Our experiment indicates an opposite result. Two occurrences of Pattern P8 have been identified in our revisions.

P9: Improvement in Dependent Project. Similarly to the Pattern P4 mentioned above, an application may depend on another application, *e.g.,* a library or a framework. A source code revision in a dependent application may impact other applications that depend on it.

The pattern *improvement in dependent project* refers to the situation where an application $A1$ depends on application $A2$ and $A2$ has been improved. Such situation may favor the performance of $A1$.

For example, the Spec application depends on the Pharo Kernel (*i.e.,* essential classes including the class Object, the collection classes and the elementary I/O APIs). One of the benchmarks associated with Spec exhibited a shorter execution time because Kernel has been improved.

This pattern occurred 2 times (13%) in the whole set of revisions we analyzed.

Others. We found 5 singular program optimizations. These optimizations are either related to the domain handled by the application or are simply puntual optimizations. For example, a revision in Roassal introduced an unnecessary logging facility toward an external file, probably some leftover from a debugging session. Removing this logging facility contributed significantly to speeding up Roassal.

7. Threats to Validity and Discussion

Projects under Study. There is a potential project selection bias in our study. Since we considered relatively large, multi-authored, and relevant applications, we do not consider small applications made by a restricted group of developers. Small applications indeed constitute the majority of the project hosted on the Pharo forges. For future work we will expand our set of analyzed applications. Note that this bias is inherent to most software analysis experiments.

Non-Exhaustive List of Patterns. The patterns we have identified are related to the domain handled by the considered applications. Other patterns may be related to a domain not covered by our experiment. For example, our benchmarks do not involve a database and it has been shown that frequently executed DB queries may introduce a performance regression [14].

Identifying Variations. Our model to identify performance variation is relatively simple, as indicated in Section 3.6 (Step S6). We currently compare a version v_i with v_{i-1}, its immediate successor. However, it may be that the comparison with $v_{i-\alpha}$, with $\alpha > 1$ would lead to a different set of performance variations. Our future work will explore this to increase the accuracy of our variation identification.

Outside the Pharo Ecosystem. This paper is voluntarily focused around the Pharo ecosystem. The relatively simple execution model of Pharo makes it possible to take care of all the obvious external factors which could bias our measurement. To have an indication on whether our findings are relevant for a different language and execution environment, we crawled over some bug repositories and searched for bug related to performance. Mozilla reported performance issues related to use of loops[9] and excessive UI update[10]. JUnit also exhibits performance regression similar to the one we identified. It noted that an excessive number of filtering over a collection

[9] https://bugzilla.mozilla.org/show_bug.cgi?id=540236

[10] https://bugzilla.mozilla.org/show_bug.cgi?id=319739#c7

causes a performance regression[11]. MySql is also facing the issue of "subquerying" over collections[12] and excessive UI redraw[13].

8. Related Work

There have been a number of empirical studies over performance bugs, and therefore related to our study.

Jin *et al.* [8] studied the root cause of 109 performance bugs from five code bases (Apache, Chrome, GCC, Mozilla and MySQL). They look for root-cause patterns among performance bugs (*i.e.,* skippable functions, uncoordinated functions or synchronization issues). They propose a rule-based performance-bug detection using rules implied by patches to find unknown performance problems.

Nistor *et al.* [16] study performance and non-performance bugs from three popular code bases: Eclipse JDT, Eclipse SWT, and Mozilla. They describe how fixing a performance bug could introduce a functional bug and how fixing performance bugs are more difficult than functional bugs. Both studies focus on how performance bugs are discovered, reported, and fixed.

Zaman *et al.* [20] study the bug reports for performance and non-performance bugs in Firefox and Chrome. They studied how users perceive the bugs, how the bugs were reported, what developers discussed about the bug causes and the bug patches. Their study is similar to that of Nistor *et al.* [15] but they goes further by analyzing additional information for the bug reports.

Nguyen *et al.* [14] interviewed the performance engineers responsible for an industrial software system, to understand these regression-causes. The engineers noted that the same regression-cause in different parts of the code will result in similar values for the performance counters (*i.e.,* signature) as long as the regression causing code is inserted anywhere along the same execution path. They 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.

Huang *et al.* [7] conduct an empirical study on 100 randomly selected real-world performance regression issues from three widely used, open source software Mysql, Postgresql and Chrome. These issues have been collected from the tracking system or mailing list of these projects. The authors selected the issues for which the fixes and the responsible change set can be found. They observe where the change takes place (*i.e.,* primitive function) and what is the impact of the changes on the source code (*i.e.,*). Their rationale is that program performance depends on how expensive an operation is and how many times the operation gets executed. Based on the insights from the study, they propose a performance risk analysis design and implementation based on static analysis.

Most of the empirical studies were done over performance bugs. These studies differ for our study in different aspects. First, we focus our research on performance variations. We analyze the performance evolution of a number of benchmarks along software evolution, in this sense, we consider performance drops and improvements that were not reported as a bug or bug-fixed. Second, we contrast the performance variations with the source code changes at fine levels of granularity; Third, we describe how the source code changes affect software performance. Four, our study consider a large variety of applications and benchmarks.

9. Conclusions

Understanding how performance varies across multiple software revisions is challenging due to several technical aspects. We propose a methodology to face these challenges and reduce measurement bias.

We found out that, for our set of considered applications, *performance variation is largely affected by identified and repetitive situations across application revisions*, which validates our hypothesis, given in Section 1.

Our future work will extend our set of analyzed applications and generalize our approach to identify performance variations. In addition, we will create strategies to statically lookup source code changes to detect software versions that could introduce a performance regression.

Acknowledgments

Juan Pablo Sandoval Alcocer is supported by a Ph.D. scholarship from CONICYT and AGCI, Chile. CONICYT-PCHA/Doctorado Nacional para extranjeros/2013-63130199. We thank Yasett Acurana for her feedback on an early version of the paper. We also thank the European Smalltalk User Group (www.esug.org) for the sponsoring.

References

[1] Juan Pablo Sandoval Alcocer and Alexandre Bergel. Tracking performance failures with rizel. In *Proceedings of the 2013 International Workshop on Principles of Software Evolution*, IWPSE 2013, pages 38–42, New York, NY, USA, 2013. ACM.

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

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming*

[11] `https://github.com/junit-team/junit/issues/38`

[12] `https://bugs.mysql.com/bug.php?id=47914`

[13] `https://bugs.mysql.com/bug.php?id=33749`

systems, languages, and applications, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[4] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy. Patterns of memory inefficiency. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 383–407, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Reading, Mass., 1995.

[6] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[7] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 60–71, New York, NY, USA, 2014. ACM.

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

[9] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 63–74, New York, NY, USA, 2013. ACM.

[10] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

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

[12] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 265–276, New York, NY, USA, 2009. ACM.

[13] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of java profilers. In *Proceedings of the 31st conference on Programming language design and implementation*, PLDI '10, pages 187–197, New York, NY, USA, 2010. ACM.

[14] Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. An industrial case study of automatically identifying performance regression-causes. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 232–241, New York, NY, USA, 2014. ACM.

[15] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 237–246, Piscataway, NJ, USA, 2013. IEEE Press.

[16] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press.

[17] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, New York, NY, USA, 2009. ACM.

[18] Bastian Steinert, Damien Cassou, and Robert Hirschfeld. Coexist: overcoming aversion to change. In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 107–118, New York, NY, USA, 2012. ACM.

[19] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '07)*, pages 75–82, New York, NY, USA, 2007. ACM.

[20] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 199–208, Piscataway, NJ, USA, 2012. IEEE Press.

[21] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 263–271, New York, NY, USA, 2006. ACM.