The Hidden Face of Execution Sampling

Alexandre Bergel, Vanessa Peña, Juan Pablo Sandoval

Department of Computer Science (DCC)
University of Chile, Santiago, Chile

ABSTRACT

Code profilers estimate the amount of time spent in each method by regularly sampling the method call stack. However, execution sampling is fairly inaccurate. This inaccuracy may give a false sense of CPU time distribution and prevents profilers from being used to estimate the code coverage.

Multiplying the execution of the code to be profiled increases the profiler accuracy. We show that the relation between the number of iterations and the precision of the profiles follows a well determined relation. We propose a statistical model to determine the right amount of iterations to reach a particular ratio of reported methods. We use this model to estimate and increase the profiling accuracy.

1. INTRODUCTION

Precisely determining what is happening during the execution of a program is difficult. Modern programming environments provide code execution profilers to report on the execution behavior. Code execution profilers for object oriented programming languages are particularly useful at estimating how much time is spent in what methods.

Execution sampling is a technique commonly employed when profiling code execution. It has a low impact on the executed program and is reasonably efficient to identify execution bottleneck, despite its limitations [2, 7, 8]. As a consequence execution sampling is commonly employed in code execution profilers.

Execution sampling behaves relatively well on long and focused program execution, however, it is fairly inaccurate for short program execution. Software engineers address this well known problem by executing the same code multiple times [10]. This artificial increase of the execution time produces a gain in the profiling accuracy. What is however unclear, is the amount of necessary iterations to reach a satisfactory profile.

The research question addressed in this paper is: *Can the number of multiple executions be related to the accuracy of the execution sampling?*

We answer this question by carefully measuring the amount of reported methods in a profile. By executing multiple times the code to profile, we measure the gain in the reported method. We determined that this gain follows a logarithmic curve. By establishing a regression model, we are able to estimate the profiling precision based on a small number of execution samples.

We first illustrate the impact on multiple execution on a number of Smalltalk applications (Section 2). Subsequently, we measure this impact and establish a statistical model for it (Section 3). We compare the Pharo and VisualWorks platforms (Section 4). We then review the related work (Section 5) before concluding (Section 6).

2. CODE EXECUTION SAMPLING

2.1 Profiling example

Consider the Smalltalk expression `XMLDOMParser parse: xmlString`. Evaluating this expression returns an abstract syntax tree describing the provided XML content. We arbitrarily pick an XML string as the benchmark we thoroughly use in this paper. Evaluating this expression for our particular XML content takes 156 ms on our machine[1].

We profiled the XML parsing using MessageTally, the standard profiler in Pharo, as follows:

```
MessageTally spyOn: [ XMLDOMParser parse: xmlString ]
```

Profiling an application increases the execution time. The total execution reported by MessageTally is 168 ms. We see an increase of $(168 - 156)/156 = 7\%$ of the execution time. Lengthening the execution time by 7% for a time profile is a compromise acceptable in our situation (*i.e.,* non-realtime operation, with a relatively thin interaction with the operating system).

MessageTally reports for the absolute and relative time spent in each method. Consider `XMLTokenizer>>nextTag`, MessageTally reveals it takes 15% of the total execution:

```
15.5% {26ms} XMLTokenizer>>nextTag
  10.1% {17ms} SAXDriver>>
      handleStartTag:attributes:namespaces:
  3.6% {6ms} XMLTokenizer>>nextEndTag
```

The method `nextTag` is reported to call two other methods, `handleStartTag:attributes:namespaces` and `nextEndTag`.

[1]All the measurements with Pharo were realized using Pharo 1.4 with the VM 5.7b3 on a MacBook Pro, 8Gb of Ram, 2.26 GHz Intel Core 2 Duo

The source code `nextTag` is quite complex, and it sends exactly 15 different messages. However, only two of them are reported by MessageTally.

About 21 different methods defined in the XML package are part of the profile (average from 20 runs). A careful tracing of the XML parsing expression reveals that 258 methods contained in the XML package are executed in total. This means that only $21/258 = 0.08 = 8\%$ of the methods involved in parsing an XML content are reported by MessageTally.

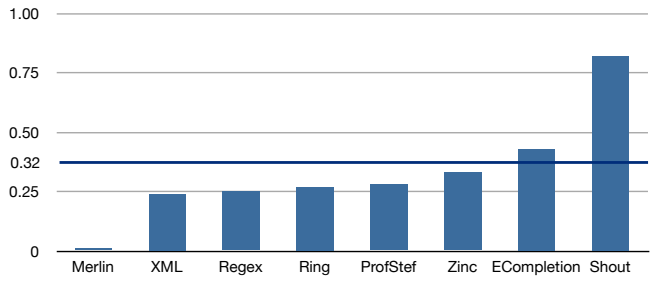

Figure 1: Ratio between reported methods and executed methods (higher is better, average = 0.32).

To verify whether this effect is particular to our particular expression or not, we run a similar experiment on 7 other applications. We took 7 popular applications contained in the standard Pharo image. We profiled the execution of the unit tests for each of them twice, the first time using an execution sampling profiler (MessageTally), and the second time by instrumenting the methods (using Compteur [2]).

Using unit tests as benchmarks is reasonable in our case since unit tests are not interactive. It has been shown that traditional profilers, including MessageTally, are inefficient to profile interaction scenarios [7].

By instrumenting the methods, we get an accurate amount of methods involved in the execution. The average of the ratio between reported methods and executed methods for our 8 applications (including the XML parsing) is 0.32.

Merlin has the lowest ratio: only 1% of the executed methods are reported by MessageTally. The reason is probably the extremely short execution of the unit tests (4 ms), meaning that less methods will be caught by the profiler (the sampling in Pharo is done every milliseconds). Shout has the highest ratio (82%), but also the longest execution time (6964 ms). This implies that more methods will be sampled by the profiler.

The fact that some methods are missed by MessageTally is a direct consequence to execution sampling, the strategy used by most of the code execution profilers.

2.2 Execution sampling

Execution sampling approximates the time spent in an application's methods by periodically stopping a program and recording the collection of methods being executed [9]. In VisualWorks and Pharo, the code to profile is executed in a new thread and the profiler runs in a thread at a higher priority [2]. When the profiling thread is activated, it inspects the runtime method call stack (accessible via the `thisContext` pseudo variable) of the observed thread. Per default, this inspection happens every millisecond[2].

Execution sampling has many advantages. Firstly, it has a low impact on the overall execution. In Pharo, code is between 5% and 12% longer to execute when being profiled. This is reasonable in the large majority of cases developed in Pharo. Secondly, execution sampling has no perceptible impact on the profiled application semantics in the large majority of cases: an application that is being profiled is expected to do what it is supposed to do, only a bit slower.

However, as we have previously shown, sampling an execution can be quite inaccurate and incomplete.

2.3 Increasing the execution time

It is common to artificially increase the execution time to gain accuracy when sampling the execution. This is easily done by running the same code multiple times. Putting the expression we are interested in a loop, also makes the profiling more precise:

```
MessageTally spyOn: [
  10 timesRepeat: [ XMLDOMParser parse: xmlString ] ]
```

The consumption of `nextTag` is now reported as:

```
15.6% {260ms} XMLTokenizer>>nextTag
  8.1% {135ms} SAXDriver>>
      handleStartTag:attributes:namespaces:
  3.1% {51ms} XMLTokenizer>>nextEndTag
  2.0% {33ms} XMLTokenizer>>
      nextAttributeInto:namespaces:
```

The call to `nextAttributeInto:namespaces:` is now revealed which readjust the overall distribution, leading to a better accuracy. The CPU share of `handle...spaces:` went from 10.1% to 8.1% and `nextEndTag` went from 3.6% to 3.1%.

By repeatedly executing the same expression, the total execution time increases, enabling (i) more methods to be detected during a sampling and (ii) the CPU share of reported methods is slimed down to gain in accuracy. 62 methods are now reported by MessageTally. This makes the ratio go from 8% to 24%.

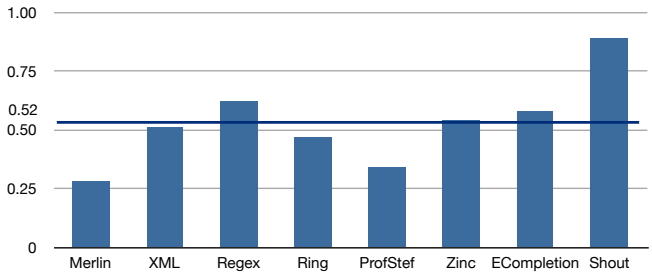

Figure 2: Ratio of reported methods with a loop of 10 iterations (higher is better, average = 0.52).

Figure 2 shows the applications for which we repeated 10 times their corresponding unit tests. The effect of the loop is significant. The average ratio of reported methods is now 0.52.

We have arbitrarily chosen a loop of 10 iterations. For some applications, the impact of using 10 iterations is stronger

[2]In Java systems, the profiling time sampling is usually 10 milliseconds [8].

than for others. For example, ProfStef has a ratio greater than XML's, however the ratio is lower with 10 iterations. Iterating has a different impact on each application essentially due from two reasons:

- Iterations stress the memory management. The number of full garbage collections is linear with the number of iterations: increasing the number of iterations results in a linear increase of the number of full garbage collections. However, this iteration is quite dispersed around the trend line. This disparity makes that multiply executing the same expression has unpredictable effects on the garbage collector.
- Methods with a very short execution time may remain hidden from the profiler, even with a high number of execution.

The following section discuss the impact of multiple executions.

3. MEASURING THE GAIN

3.1 Regression line

To get a better understanding of the impact of multiple executions, the following expression successively profiles the XML parser:

```
#(1 11 21 31 41 51 ... 191) do: [ :numberOfIterations |
  MessageTally spyOn: [
    numberOfIterations
      timesRepeat: [ XMLDOMParser parse: xmlString ] ] ]
```

The effect of using `numberOfIterations timesRepeat: [ ... ]` artificially increase the execution time of the expression we are interested in. Naturally, we assume all the executions are the same. The code we gave is a simplified version of how we actually measured the profiles. We make sure that before profiling we properly clean the memory by running the garbage collector multiple times.

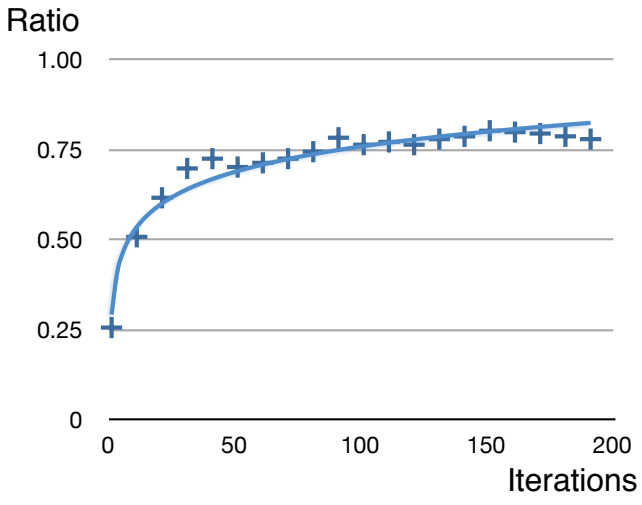

Figure 3: Evolution of the ratio against multiple execution.

Figure 3 shows our measurements. The horizontal axis is the number of iterations for the XML parsing expression. The number of times it is executed goes from 1 to 241, with an increment of 10. The vertical axis is the ratio of reported methods. It goes from 0.24 and tops at 0.81. Each cross corresponds to a measurement *(iteration, ratio)*.

The trendline is indicated with a continuous line and has a logarithmic shape. The regression equation given by common statistical tools [6] has the pattern $y = a\ ln(x) + b$. In the case of our XML parsing, $a = 0.0974$ and $b = 0.2918$. The associated "test of goodness of fit", R^2, is 0.9514. A value close to 1 means a good fit, *i.e.,* the equation matches the observed data. We will give an accurate definition of R^2 later on (Section 3.3).

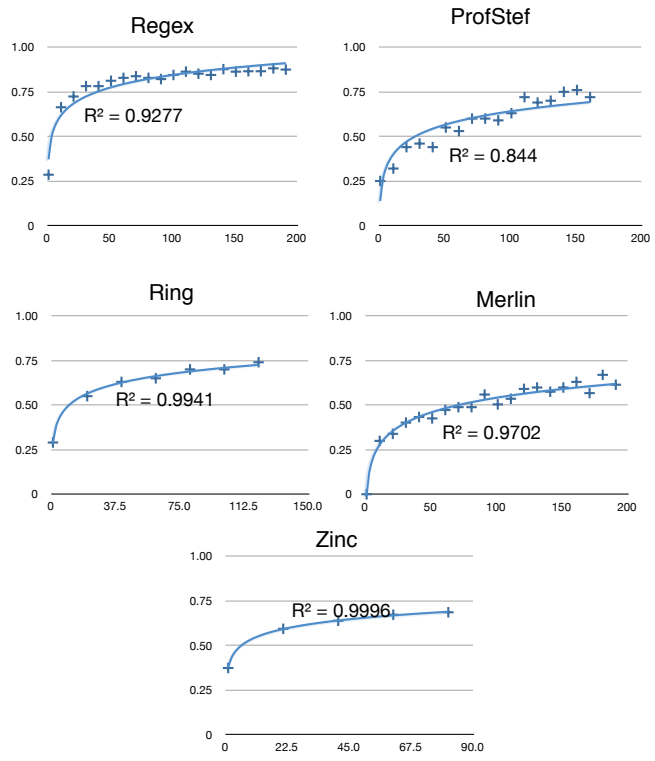

Figure 4: Evolution of the ratio.

We repeat the same analysis on our applications (Figure 4). All the regression lines are logarithmic curves, with a R^2 over 0.89.

The ratio of identified methods ranges from 0% to 100%. Obviously, it cannot be greater than 1. In practice, getting a ratio of 100% (*i.e.,* determining all the methods that are effectively used by sampling the execution) is hardly achievable. Even for a high number of iterations, the ratio stalls around 90% when we pick a high number of iteration. As a consequence, the regression model $y = a\ ln(x) + b$ is valid for values of x that produces a y lesser than 0.90.

In the following, we do not discuss any further about this asymptotic behavior as we leave it for future work.

3.2 Determining a and b

We have seen that the regression line follows the pattern $y = a\ ln(x) + b$. For a given set of iterations (x), we can determine the method ratio of the profiling (y) assuming that we know about a and b.

As our measurement show, the value of a and b are proper

to the piece of code to be profiled. Since we are interested in predicting the amount of iterations for a given ratio, we need to determine a and b.

First, we need to get rid of the logarithm by writing $X = ln(x)$ and $Y = y$. The equation becomes $Y = aX + b$, which is much simpler to reason about. For a given set of (X, Y) plots, a and b are easily determined using linear least squares fit:

$$b = \frac{SS_{XY}}{SS_{XX}} \qquad a = \overline{Y} - b\,\overline{Y}$$

where

$$SS_{XY} = \sum XY - \frac{(\sum X)(\sum Y)}{n} \qquad SS_{XX} = \sum X^2 - \frac{(\sum X)^2}{n}$$

We further have n is the number of samples; SS stands for "sum of squares"; $\overline{X}$ is the average of all the X values; $\overline{Y}$ is the average of all the Y values.

Using our example of XML parsing, we already had

iterations (x)	ratio (y)
31	0.61
61	0.72
91	0.7
121	0.78

The values of $X = ln(x)$ are therefore $\{ln(31),\ ln(61),\ ln(91),\ ln(121)\}$. By applying the formulas given above, we find $a = 0.1097$ and $b = 0.2404$.

The regression equation for the XML parsing is therefore $y = 0.1097\ ln(x) + 0.2404$. This equation is pretty close to what we have found in the previous section.

We can then deduce:

$$x = e^{\frac{y-b}{a}}$$

If we wish to obtain a ratio of 0.8 of our profile, then we need $e^{\frac{0.8-0.2404}{0.1097}} = 164$ iterations. Our measurement shows that the 0.8 ratio threshold is reached after 151 iterations.

3.3 How confident are we?

The previous section gives a model that binds the amount of code iterations with the ratio of reported methods during a profile. We use our model to "predict" the value of the ratio for a given amount of iterations.

One piece in our analysis is however missing, which is about the trust we can give in our prediction. In statistics, the coefficient of determination R^2 tells about the amount of variability in a data set. The more variable a data set is, the higher R^2 is. The definition of R^2 that is commonly used is:

$$R^2 = 1 - \frac{SS_{err}}{SS_{tot}}$$

where $SS_{err} = \sum(y_i - f(x_i))^2$ and $SS_{tot} = \sum(y_i - \overline{y})^2$. SS_{tot} is the total sum of squares. SS_{err} is the sum of squares of residuals. The modeled values are obtained with $f(x) = a\ ln(x) + b$.

When applied to the four (x, y) pair given previously, we have $R^2 = 0.84$. Being close to 1 indicates that the regression given previously is indeed an accurate model of the pair values.

4. THE CASE OF VISUALWORKS

The measurement given above have been realized in Pharo Smalltalk with a non-jitted virtual machine. To verify whether the model we have previously described is particular to Pharo or not, we take VisualWorks Smalltalk[3], a popular Smalltalk dialect, and run a similar set of experiences.

The executing environment and profiler of VisualWork (VW) differ from the one of Pharo on two essential points:

- The virtual machine of VW is significantly faster than the non-jitted one of Pharo.
- In VW, the sampling period is randomly selected in a range $[1, 32]$ milliseconds. Using a random sampling period leads to an increase of precision [8]. In Pharo the sampling period is fixed.

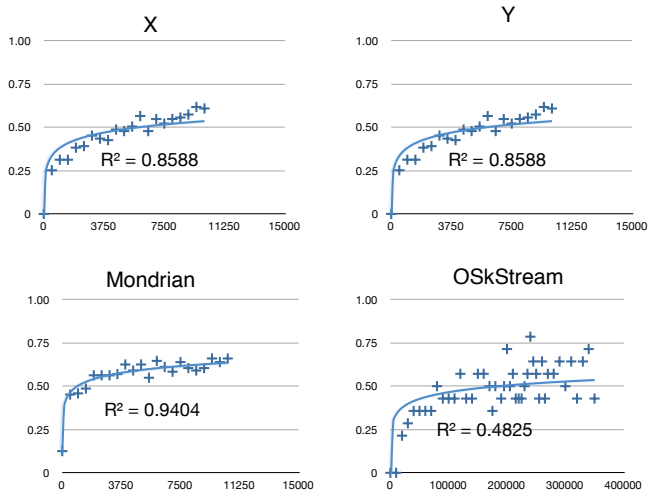

Figure 5: Evolution of the ratio on VisualWorks.

Figure 5 the evolution of the ratio for 4 applications. We tried two industrial applications, noted `X` and `Y`, and two open-source applications, Mondrian and OSkSubStream.

We profiled the two industrial applications on a Window XP machine and the two open-source applications on a Linux Ubuntu[4].

The OSkSubStream measurements are weakly correlated. We speculate that the cause is the presence of many short methods. Interestingly, we experienced in our previous work [2] a similar significant variation when profiling PetitParser, a streaming and parsing framework in Pharo.

These four experiences confirm our previous finding: the ratio between profiled methods and executed methods follows a logarithmic curve. Our results strongly suggest that the performance of the virtual machine, the just-in-time compiler and the random sampling do not impact the ratio of reported methods in a sampling-based profile.

5. RELATED WORK

Extracting accurate profiles from a software execution is challenging. Several techniques have been proposed to

[3] `http://www.cincomsmalltalk.com`

[4] VisualWorks 7.7.1 on a PC Core i5 2.30GHz, 4Gb of Ram

increase the accuracy and reduce the overhead of execution sampling.

Mytkowicz *et al.* evaluate and compare 4 profilers for Java. They discovered that the produced profiles are different: the hot methods identified by one profiler may not be the same as the ones identified by another profiler. They then propose a more accurate profiler that collects samples randomly and it does not suffer from the above problems [8].

Fischmeister and Ba [5] propose theorems to determine the sampling period in different scenarios, and heuristics to extend the sampling period to reduce the overhead.

Our approach, however, shows that using a fixed or a random sampling period produces similar result when using multiple code execution

Whaley present a sampling-based profiler for Java Virtual Machines. It is able to correctly identify calling context without walking the entire stack and distinguish between frequently-executed and long running methods. Also, the profile data is extremely accurate [9].

Binder present a sampling-based profiling framework for Java [4] based on custom profiling agents in pure Java. The sampling is realized by counting bytecode instructions. It offers a good trade-off between high accuracy of profiles and reasonable overhead. The features of Binder profiler can improve our overall impact. In this sense the ratio between profiled methods and executed methods could be more deterministic. As a future work, we plan to apply this approach on counting messages [2].

Arnold and Ryder [1] present a framework to perform instrumentation sampling. Their framework duplicate method bodies, and counter-based sampling to switch between instrumented (copy) and no-instrumented code. This reduce the overhead but a relatively expensive process has to be done first.

6. CONCLUSION & FUTURE WORK

The difficulty to properly monitor an application execution imposes a severe compromise between what information can be extracted and the cost to obtain it. In this paper we have motivated and measured a simple way to increase the accuracy of profiles based on execution sampling.

So far we have empirically determined a logarithmic relation between the amount of iterations and the ratio of identified methods. As a future work, we will investigate the cause of this relationship by monitoring the evolution of the call graphs.

The regression model that we determine follows a logarithmic curve ($y = a\ ln(x) + b$). However, we have found that the produced ratio (y) is asymptotic in practice: y does not go above a ratio (around 90%), even for a large x. As future work, we plan to refine our model with this asymptotic behavior for large number of iterations.

Most code execution profilers simply report what has been detected, without giving any feedback about the quality of the profile. As future work, we plan to integrate our model into Kai [3], a fully fledged code execution profiler. This will make Kai the very first profiler to provide feedback on the profile quality.

Acknowledgments.

We gratefully thank Chris Thorgrimsson for the multiple discussions and brainstormings we had. We thank Walter Binder for reviewing an early draft. We also thanks Johan Fabry and Romain Robbes for the discussion we had on the statistical part.

This work has been partially funded by Program U-INICIA 11/06 VID 2011, grant U-INICIA 11/06, University of Chile, and FONDECYT 1120094.

7. REFERENCES

[1] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. 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] Alexandre Bergel, Felipe Ba nados, Romain Robbes, and Walter Binder. Execution profiling blueprints. *Software: Practice and Experience*, August 2011.

[4] Walter Binder. Portable and accurate sampling profiling for java. *Softw. Pract. Exper.*, 36(6):615–650, 2006.

[5] Sebastian Fischmeister and Yanmeng Ba. Sampling-based program execution monitoring. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, LCTES '10, pages 133–142, New York, NY, USA, 2010. ACM.

[6] David Freedman, Robert Pisani, and Roger Purves. *Statistics, Third Edition*. W. W. Norton & Company, 1997.

[7] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 155–170, New York, NY, USA, 2011. ACM.

[8] 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.

[9] John Whaley. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 78–87, New York, NY, USA, 2000. ACM.

[10] Steve Wilson and Jeff Kesselman. *Java Platform Performance*. Prentice Hall PTR, 2000.