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 shows that the relation between the number of iterations and the precision of the profiler follows a logarithm curve. We propose a statistical model to determine the right amount of iterations to reach a particular ration 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 are particularly useful at estimating where the CPU (or the virtual machine) is spending time on.

Precisely determining the CPU consumption time share for each method involved in a computation is challenging. Inspecting the execution of a program has an impact on the execution itself. Execution sampling is a technique commonly employed when profiling code execution. It is relatively accurate and it keeps the impact on the executed program relatively low. As a consequence execution sampling is used in the large majority of code execution profilers.

Execution sampling behaves relatively well on long and focused program execution, however, it is fairly inaccurate for short program execution. This is a well known problem that software engineers address by executing the same code multiple times [9]. 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 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. Evaluating this expression with an arbitrary 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. The source code nextTag is quite complex, and it sends

[1] MacBook Pro, 8Gb of Ram, 2.26 GHz Intel Core 2 Duo

exactly 15 different messages. However, only two of them are reported by MessageTally.

About 64 different methods are part of the profile (average from 20 runs). A careful tracing of the xml parsing expression reveals that 258 methods are executed in total. This means that only 0.24 = 24% of the methods involved in parsing an xml content are reported by MessageTally.

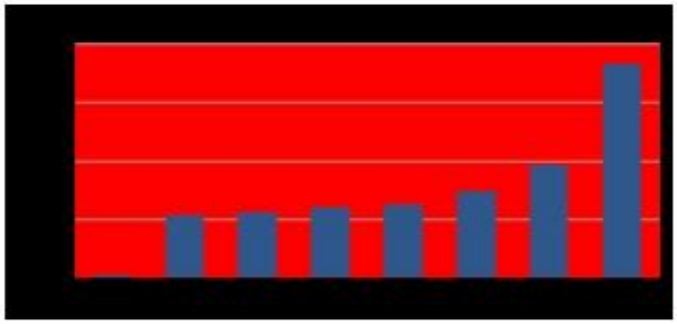

Figure 1: Ratio between reported methods and executed methods (higher is better).

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 ~~applications~~ 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 method (using Compteur [2]). By instrumenting the method, 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 0.35.

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 (we recall by the sampling is done every milliseconds). Shout has the highest ratio (0.82). Shout has 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 [7]. 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, ~~a~~ code is between 5% and 12% longer to execute when being profiled.

[2] In Java systems, the profiling time sampling is usually 10 milliseconds [6].

This is reasonable in the large majority of ~~the~~ cases developed in Pharo (i.e., non-realtime application with little dependencies on the operating system). Secondly, execution sampling has no impact on the profiled application semantics in the large majority of cases.

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 ~~in~~ accuracy when sampling the execution. This is easily done by running the same code multiple times ~~the code to profile~~. 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. By repeatedly executing the same expression, the total execution time increases, enabling more methods to be detected during a sampling. 132 methods are now reported by MessageTally. This makes the ratio goes from 0.24 to 0.51.

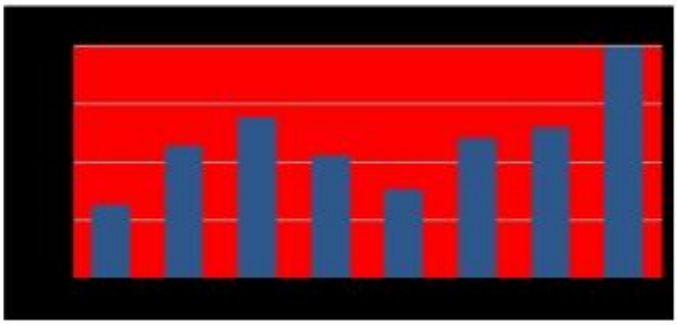

Figure 2: Ratio of reported methods with a loop of 10 iterations (higher is better).

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 0.58.

We have arbitrarily chosen a loop of 10 iterations. For some applications, the impact of using 10 iterations is stronger than for others. For example, sampling 10 executions of the unit tests of Regex reports 66% of the used methods, whereas it was only 25% with a single 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 ... 241) 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.

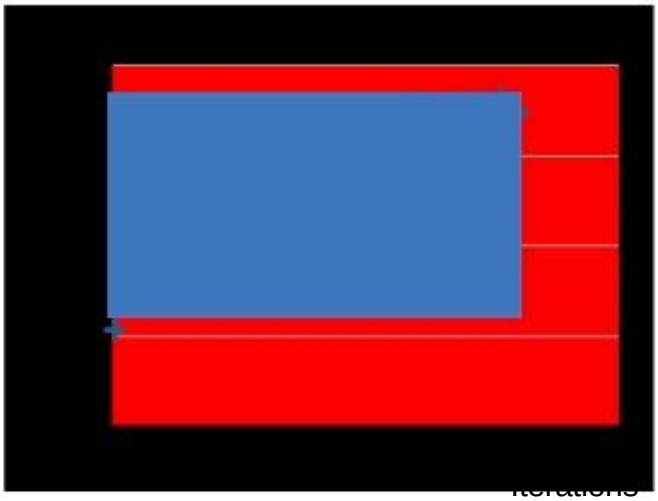

Figure 3: Evolution of the ratio against multiple execution.

Figure 3 shows our measurements. Horizontally 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. Vertically 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 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).

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

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

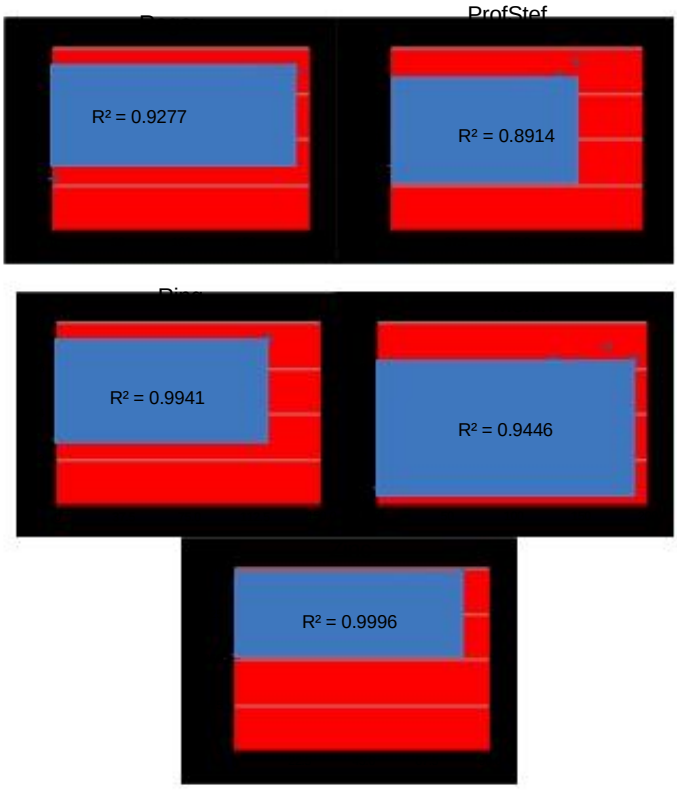

Figure 4: Evolution of the ratio.

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 the standard statical books:

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

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" ; X is the average of all the X values; $\bar{Y}$ 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 ~~have~~ what we have found in the previous section.

We can then deduce:

$$x = e^{yab}$$

If we wish to obtain a ratio of 0.8 of our profile, then we need $e^{\frac{0.1097}{0.8 - 0.2404}}$ = 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.

4. VISUALWORKS AND PHARO

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 [6]. In Pharo the sampling period is fixed.

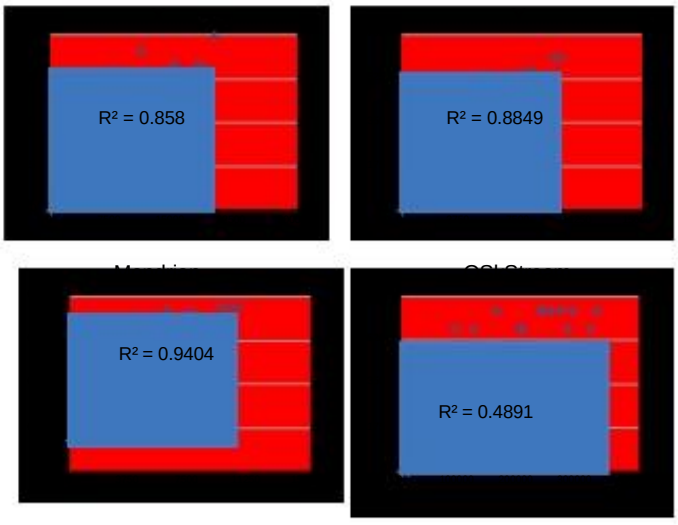

Figure 5: Evolution of the ratio on VisualWorks.

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. 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.

The paragraph below is repeated above.

~~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.~~

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

5. RELATED WORK

Extracting accurate profiles from a software execution is challenging. Several techniques have been proposed to increase the accuracy and reduce the overhead of execution sampling.

Mytkowicz et al. evaluate 4 profilers for Java. They found that they do not match in their evaluation to identify the hot methods, candidates to optimize. They then propose a more accurate profiler that collects samples randomly and it does not suffer from the above problems [6].

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

Our approach, however, we can see that even if we use random sampling or a determinate sampling period, the ratio between reported methods and executed methods is inaccurate.

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 [7].

Binder present his sampling-based profiling framework for Java [?] to implement custom profiling agents in pure Java. It is a sampling based on byte-code instruction counting and 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.

Arnold and Ryder [1] present a framework to perform instrumentation sampling. Their framework perform a code duplication, and counter-based sampling to switch between instrumented (copy) and no-instrumented code. This reduce the overhead but need to do an expensive instrumentation first.

In order to improve profiler accuracy, we propose to execute the scenario a larger number of times, allowing the profiler to catch more methods. However this has a big impact on the profiler overall execution overhead. For this, our approach does not apply to real time systems, for example.

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.

As future work, we plan to integrate our model into Kai [3], a fully fledged code execution profiler.

Acknowledgments.

We thanks Johan Fabry and Romain Robbes for the discussion we had on the statistical part.

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., 36:615-650, May 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] 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.

[7] 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.

[8] 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.

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