Counting Messages as a Proxy for Execution Time in Pharo

Alexandre Bergel

PLEIAD Lab, University of Chile, Santiago, Chile
http://bergel.eu
http://pleiad.dcc.uchile.cl

Accepted at ECOOP'11 - Do not distribute, this is not the final version

Abstract. Code profilers are used to identify execution bottlenecks and understand the cause of a slowdown. Execution sampling is a monitoring technique commonly employed by code profilers because of its low impact on execution. Nevertheless, regularly sampling the state of a program under execution is highly sensitive to the executing environment, making it non-deterministic and not comparable for cross-platform executions. These factors are moreover exacerbated when profiling programs running on a virtual machine.
We propose to count method invocations as a proxy for estimating execution time. Using principally the Pharo platform for experimentation, we show that such a proxy is more accurate, more reliable over multiple executions and profiles are comparable, even when obtained in different execution contexts. We have produced *Compteur*, a new code profiler that does not suffer from execution sampling limitations.

1 Introduction

Software execution profiling is an important activity to identify execution bottlenecks. Most programming environments come with one or more powerful code execution profilers.

Profiling the execution of a program is delicate and difficult. The main reason is that introspecting the execution has a cost, itself hardly predictable. This situation is commonly referred as the Heisenberg effect[1]. Profiling an application is essentially a matter of compromise between accuracy of the obtained result and the perturbation generated by the introspection.

Execution profiling is commonly achieved via different mechanisms, often complementary: simulation [20], application instrumentation and periodically sampling the execution, typically the method call stack. Sampling the execution is favored by traditional code profilers since it has a low overhead and it is accurate for a long application execution.

Profilers based on execution sampling assume that the number of samples for a method is proportional to the time spent in the method. If samples were

[1] "Observation that the very act of becoming a player changes the game being played.", http://www.businessdictionary.com/definition/Heisenberg-effect.html.

obtained independently from the executing context (e.g., garbage collector, thread scheduling) then execution sampling would be reliably and accurate.

Nevertheless, execution profiling is highly sensitive to garbage collection, thread scheduling and characteristics of the virtual machine, making it non-deterministic (e.g., the same execution profiled twice does not generally give two identical profiles) and tied to the executing platforms (e.g., two profiles of the same execution realized on two different virtual machines or operating systems cannot be meaningfully related to other other).

Over the last decade there has been a whole range of new emerging object-oriented programming languages that are syntactically simple, with a minimal core, with few but strong principles. One of them is Pharo[2]. In Pharo, invoking a method (also termed "sending a message") is a syntactic construction in which a computation can solely be expressed in terms of. Class and method creation, loops, conditional branches are all realized via sending messages. Pharo is an open source Smalltalk dialect. The results presented in this paper were obtained with Pharo, we believe they equally apply to other "pure" object-oriented programming languages however (e.g., Jython[3], JRuby[4], Newspeak[5], Groovy[6]).

This paper argues that counting method invocations has strong benefits over execution sampling in Pharo. Since Pharo realizes a computation by almost exclusively invoking methods, it is natural to evaluate whether counting method invocations can be used as a proxy for estimating the application execution time.

The three research questions addressed in this paper are:

- A - *Does the number of methods invoked during the execution of an expression is related to the time taken for the expression to execute?*
- B - *Is the number of invoked methods more stable than the execution time over multiple executions?*
- C - *Is the number of invoked methods as useful as the execution time to identify an execution bottleneck?*

This paper confirms these three questions after careful and extended measurements in different execution settings. We show that counting the amount of executed send bytecode instructions is an accurate proxy for estimating the execution time of an application and an individual method.

Naturally, the execution time of a piece of code is not solely related to the amount of invoked methods. Garbage collection, use of primitives offered by the virtual machine, and native calls are likely to contribute to the execution time. However, for all the applications we have considered in our experiments, these factors represent a minor perturbation. The amount of method invocations is highly correlated with execution time (correlation of 0.99 when considering the application execution and 0.97 when considering individual methods).

The main innovations and contributions of this paper are summarized as follows:

[2] `http://www.pharo-project.org`
[3] `http://www.jython.org`
[4] `http://jruby.codehaus.org`
[5] `http://newspeaklanguage.org`
[6] `http://groovy.codehaus.org`

– the limitations of execution sampling are identified
– for a number of representative applications, we show that the number of method invocations is correlated with the execution time, for each individual method and each application
– we describe a general model for evaluating the stability and precision of profiles over multiple executions
– we show empirically that the number of method invocations is a more stable criterion for profiling applications than execution sampling

The paper is outlined as follows. The limitations of profiler based on execution sampling are first presented (Section 2). We introduce counting the number of method invocations as a reliable profiling criterion (Section 3). We then show the ability of counting method invocations to identify execution bottlenecks (Section 4). Limitations of execution sampling are then reviewed against message counting (Section 5). Subsequently, key implementation points are presented (Section 6). Perspective and lessons learnt are given next (Section 7). We then review the related work (Section 8) before concluding (Section 9).

2 Profiling based on Execution Sampling

Profiling is the recording and analysis of a program execution. Profiling is often considered essential when one wants to obtain the representation of a program execution. A profiler has to be carefully designed to provide a satisfactory balance between the different tradeoffs associated with accuracy and overhead.

Execution sampling approximates the time spent in an application's methods by periodically stopping a program and recording the collection of methods being executed. Such a profiling technique is relatively accurate since it has little impact on the overall execution. Almost all mainstream profilers (JProfiler[7], YourKit[8], xprof [10], hprof[9]) use execution sampling. Although profilers have been part of the standard software engineering toolset for decades, execution sampling comes with a number of serious issues. As we will see, some of these issues were already pointed out by other researchers but we have deliberately chosen to list them in this section for sake of completeness and because we will address them in the forthcoming sections.

This section is presented from the point of view of the Pharo programming language.

Dependency on the executing environment. Sampling profiling is highly sensitive to the executing environment. As one may expect, profiling while other threads or OS processes are running is likely to consume resources including CPU and memory which could invalidate the measurements. Most of operating systems use the *multilevel feedback queue* algorithm to schedule threads [13]. The algorithm determines the nature of a process and gives preferences to short job and input/output processes. The thread scheduling disciplines offered by

[7] `http://www.ej-technologies.com`
[8] `http://www.yourkit.com`
[9] `http://java.sun.com/developer/technicalArticles/Programming/HPROF.html`

operating systems and/or virtual machines makes thread scheduling a source of measurement perturbation that cannot reliably be predicted.

Execution sampling traditionally requires virtual machine support[10] or an advanced reflective mechanism. In Pharo, execution sampling is realized via a thread running at a high priority that regularly introspects the method call stack of the thread that is running the application. Scheduling new threads or varying the activity of existing threads (e.g., a refresh made by the user interface thread) is a source of perturbation since a smaller time share is granted by the scheduler to the profiled thread for the total profiled execution time.

Garbage collection is another significant source of perturbation since a memory scan (necessary when scavenging unused objects) suspends the computation, and thus augments the application execution time. Garbage collection occurs when memory is in short supply and is hence not exactly correlated with any particular execution sequence.

These problems are not Pharo specific. They are found in common execution platforms, as mentioned by Mytkowicz *et al.* [18]. There are numerous other source of measurement bias, for example the relation between the sampling period and the period of thread scheduling [18] and to some extent the room temperature since it affects the CPU clock speeds [17,5].

Non-determinism. Regularly sampling the execution of an application is so sensitive to the executing environment that it makes the activity non-deterministic. Profiling twice the very same piece of code does not produce exactly the same profile. Consider the Pharo expression `30000 factorial`. On an Apple MacBook Pro 2.26Ghz, evaluating this expression takes between 3 803 and 3 869 ms (ranges obtained after 10 executions). The difference may be partially explained due to the variation of the garbage collection activity. Computing the factorial of 30 000 triggers between 800 and 1000 incremental garbage collections in Pharo. The point we are making is not about the implementation of the factorial function that requires a garbage collector, but the variation in the memory activity an arbitrary code evaluation may have.

A common way to reduce the proportion of random perturbations is to ensure that the code to be profiled takes a long execution time. By doing so, the effect of the garbage collector is minimized. Having long profiling period gains in accuracy, however it makes code profiling an activity that may not be realized as often as a programmer would like to do.

Lack of portability. Profiles based on execution sampling are hardly reusable across different runtime execution platforms [4], virtual machines and CPUs. A profile realized on a platform A cannot be easily related to a similar profile realized on a platform B. For example, the first version of the Mondrian visualization engine [16] was released in 2005 for Visualworks Smalltalk[11]. In 2008 Mondrian development has been moved to Pharo. Since its beginning Mondrian has been constantly profiled to meet scalability and performance requirements. However, because of (i) the language change from Visualworks to Pharo, (ii) the constant

[10] e.g., `http://download.oracle.com/javase/1.4.2/docs/guide/jvmpi/jvmpi.html`
[11] `http://www.cincomsmalltalk.com/main/products/visualworks/`

evolution of Pharo and (iii) the continuously evolution of the physical machine and the Pharo virtual machine, profiles cannot meaningfully be related to each other.

Shared resources. In addition to the general issues mentioned above, a particular profiler implementation comes with its own limitations.

Memory is a persistent global shared resource. Executions that were realized before beginning the profiling may leave the memory in such a state that the application is prone to an excessive garbage collection triggering. In Pharo, the programming environment uses the same memory heap that is used to run applications. Activities that are realized before a profile may impact this one.

MessageTally, the standard profiler of Pharo, constructs a profile sharing the same memory space as the running application, which is a favorable condition for the Heisenburg effect. The longer the application execution is, the more objects are created by MessageTally to model the call graph and stock runtime information, thus exercising additional pressure on the memory manager.

3 Counting Messages as a Proxy for Execution Time

Almost all the computation in Smalltalk, and thus in Pharo, is realized via sending messages ("everything is an object"). Operations like conditional branching (*if*-like statement), arithmetic, class creation, method creation are essentially realized via sending messages.

Such a program execution platform suggests that CPU time consumption is likely to be related to the number of sent messages.

3.1 Execution time and number of message sends

Determining whether the number of messages sent during the execution of an expression is related to the time taken for the expression to execute is a bit trickier than it appears. Execution time measurements are hardly predictable. As any statistical measurement, a relation between the execution time and the number of message sends is realized by bounding the error margin in the measurement. The relation is established if this margin is "small enough". Determining a relation between two data sets requires a number of statistical tools [14]. We will follow the traditional steps of constructing a regression model.

Intuitively, we expect the number of messages sent during the execution of an expression to increase with an increase of the execution time: the longer an expression takes to execute, the more messages are sent. We will later discuss about native calls and other interactions with the operating system. This subsection answers to the research question *A*.

Measurements. From the Pharo ecosystem[12] we selected 16 Pharo applications. We selected these applications based on their coverage of Pharo abilities. Appendix A lists the applications and gives the rationale why we have chosen them.

[12] Principally available from the forge `http://www.squeaksource.com`

The experiment conducted has been realized on a MacBook Pro 2.26 GHz Intel Core 2 Duo with OSX 10.6.4 and 2Gb 1067 MHz DDR3 using the SqueakVM Host 64/32 Version 5.7b3 (this execution context is designated as c in the following sections).

Our measurements, used to relate the number of sent messages with the execution time, have to be based on representative application executions to be the closest to what programmers are facing. Running unit tests is convenient in our setting since unit tests are likely to represent common usages and execution scenarios [15]. We execute the unit tests associated to each of the 16 applications. None of the test we used in this paper manipulate randomly generated data or makes use of non-deterministic data input. The execution time and the number of message sends are measured for each test suite execution. As an illustration of the message send metric we are interested in, consider the following code (i.e., a simplified version of a test, part of the Moose application test suite):

```
ModelTest>> testRootModel
  self assert: MooseModel new mooseID > 0

Behavior>> new
  ^  self basicNew initialize

Behavior>> basicNew
  <primitive: 70>

Object>> initialize
  ^ self

MooseElement>> mooseID
  ^ mooseID
```

The test testRootModel sends 6 messages. The messages assert:, new, mooseID and > are directly sent by testRootModel. The message new sends basicNew and initialize. The total number of messages sent by testRootModel is 115. The message assert:, which belongs to the SUnit framework, does some checks on the argument and the method initialize is redefined in the class MooseElement.

The number of messages can easily skyrocket. Running the tests associated to the Pharo collection library [8] takes slightly more than 32 seconds. The test execution sends more than 334 million messages.

Linear regression. A scatter plot is drawn from our measurements (Figure 1). Each of the applications we have profiled is represented by a point *(execution time, number of message sends)* and is denoted with a cross in the scatter plot. These values form an almost straight line with a statistical correlation of 0.99. The correlation is a general statistical relationship between two random variables and observed data values: a value of 1 means the data forms a perfect straight line. This line, commonly called regression line, may be "deduced" from these values.

The general equation of a regression line is $\hat{y} = a + bx$ where a the constant term; b is the line slope; x is the independent variable; y is the dependent variable;

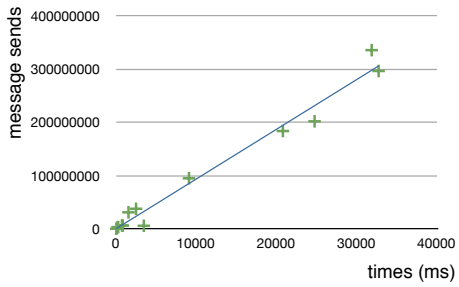

Fig. 1. Linear regression for the 16 Pharo applications.

$\hat{y}$ the predicted value of y for a given value of x. The independent variable is the execution time and the dependent variable is the number of message sends. We also put an additional constraint on the constant term: an execution time of 0 means that no message has been sent.

Using the material provided in Appendix A, we estimate the sample regression line to be $\hat{y} = 9\ 335.55\ x$. The line is drawn in Figure 1.

The slope of the regression line is 9 335.55. This value corresponds to the rate of message sends per millisecond for the applications that were run on our machine. We designate the average message rate (number of message sends per unit of time) as $MR_{\Gamma,c}$ where Γ is the set of the applications we profile and c the context in which the experiment has been realized. c captures all the variables for which the measurements depends on (e.g., computer, RAM, method cache implementation, temperature of the room).

We now have established the relation between the number of message sends and the execution time. We are not done yet however: only an approximation has been determined. The $MR_{\Gamma,c}$ value has been computed from an arbitrary set of applications. If we would have chosen a different set of applications, say Γ', $MR_{\Gamma',c}$ would have probably be slightly different from $MR_{\Gamma,c}$. $MR_{\Gamma,c}$ is said to be a random variable, and it possesses a probability distribution. Assuming that the applications we have chosen are representative of the all possible applications available in Pharo, the real value of $MR_{\mathbb{A},c}$, where $\mathbb{A}$ is the set of all Pharo applications, lives in an interval that can be easily calculated according to how confident we want to be in our findings.

The standard deviation of error tells us how widely the errors are spread around the regression line. This value is essential to estimate the confidence interval that includes $MR_{\mathbb{A},c}$. Appendix A details how the standard deviations (s_e and s_b) are computed. We have the standard deviation of error $s_e = 16\ 448\ 897$. The confidence interval is $[MR_{\Gamma,c} - t\ s_b; MR_{\Gamma,c} + t\ s_b]$ where $s_b = 350.84$ is the standard deviation of $MR_{\Gamma,c}$ and t is a value obtained from the standard t distribution table based on the confidence $(1 - \alpha)100\%$ we want to have.

For a 95% confidence interval, we have $\alpha = 0.05$ and therefore $t = 2.145$ according to the standard t distribution, which may be found in any statistical

text book. As a result, the confidence interval is [8 582; 10 089], which means that there is a probability of 95% that the real value $MR_{\mathbb{A},c}$ is within the interval.

The linear regression model enables the prediction of the execution time from the number of sent messages. Consider GitFS, an implementation of Git in Pharo. The tests of GitFS emit 28 096 569 messages to run. According to the regression model, this corresponds to a period of time $(28\,096\,569 - 418\,253)/9\,335.55 = 2\,965$. GitFS' tests actually run in 2 928 milliseconds, which is included in the time interval [2743; 3225].

3.2 Method invocation

This section compares the variation of the execution time with the number of sent messages, which answers to the research question B.

Hash values. Before we further elaborate on the precision of message counting, it is relevant to remark that executing multiple times the *same* code expression may not always emit the same amount of method invocations. For example, adding an element to a *set* does not always send the same amount of messages on evaluation. Consider the following code excerpt:

```
| s |
s := Set new.
Compteur numberOfCallsIn:
  [ 1000 timesRepeat: [ s add: Object new ] ]
```

Line 2 creates a new set. Line 3 invokes our library by sending the message numberOfCallsIn: which take a block as parameter (a block is equivalent to a lambda expression in Scheme and Lisp and an inner class in Java). Line 4 creates 1000 entries in the set. The hash value of the key objects are used for the internal indexing of the set. The virtual machine gives hash values and they cannot be predicted since they are based on the hardware clock which is used as a pseudo random number generator. Each execution of this piece of code gives a different values (e.g., 54 383, 55 997, 56 165) since the computation needed to add an object into a table depends on the object hash value pseudo-randomly provided by the virtual machine.

Even though the way hash values are assigned to objects is indeed a source of non-determinism, as we will subsequently see, it has a low impact on our measurement: for the applications we have profiled the number of message invocations varies significantly less than the execution time. Interactively acquiring data from the user, the filesystem or the network may also be another source of variation for the number of message sends.

Coefficient of variation. Each execution of the same piece of code results in a different execution time and a different number of message sent. We will now assess whether the number of sent messages is a more stable metric than the execution time over multiple executions. For each of the 16 applications we executed 10 times its tests and determined the standard deviation of execution time ($s_{TimeTaken}$)

and amount of sent messages ($s_{messages}$). To be able to compare these two standard deviations, we use the coefficient of variation, defined as the ratio of the standard deviation to the mean, resulting in c_{time} and $c_{messages}$, respectively. Appendix A gives our measurement and details how the variation is computed.

For the 16 applications we considered, our result shows that the stability of the execution time (the c_{time} column) varies significantly from one application to another. For example, the applications ProfStef, Glamour and Magritte are relatively constant in their execution time length. The variation may even be below 1% for ProfStef. However, execution time significantly changes at each run for a number of the applications. The execution time of XMLParser, DSM and PetitParser varies from 25% to 46%. The execution time of PetitParser may vary by 46% from one run to another. The reason for this is not completely clear. After some private discussion with the author of PetitParser about the cause of this variation, it seems to be caused by the intensive use of short methods on streams. These short methods, such as `peek` to fetch one character from a stream and `next` to move the stream position by one, have an execution time close to the elementary operations performed by the virtual machine to lookup the method cache.

On the opposite, message counting is a much more stable metric since its variation is usually below 1%. The greatest variations we have measured are with Mondrian and Moose. This is not surprising since these two applications intensively use non-deterministic data structures like sets and dictionaries to store their model.

The average value of c_{time} and $c_{messages}$ are 13.95 and 0.61, respectively. For the experimentation set up we have used, we have found that over multiple executions of the same piece of code measuring the number of sent messages is 22.86 (13.95 / 0.61) times more stable than measuring the execution time.

3.3 Effect of the execution context

We repeated the experiment on two different execution platforms: on the MacBook Pro using the Cog virtual machine (which supports a Just-In-Time compilation (JIT)) and a Linux Gentoo (2.6.34-gentoo-r6 running on an Intel Xeon CPU 3.06GHz GenuineIntel) using a non-jitted virtual machine.

On the Cog virtual machine we have $MR_{\Gamma,c'} = 58\ 384.75$, with a 95% confidence interval [55 325; 64 123]. On this platform, the ratio between c_{time} and $c_{messages}$ is 18.98. This is lower than what we obtained on the non-jitted virtual machine. The reason stems from the multiple method compilations, each being a resource consuming process on its own.

On the standard virtual machine running on Linux we obtained $MR_{\Gamma,c''} = 12\ 412.34$, with a 95% confidence interval of [9 615; 14 121]. The ration between c_{time} and $c_{messages}$ is 22.34, which corresponds to the ratio we have measured on the MacBook Pro machine.

3.4 Tracking optimizations

We identified a number of execution bottlenecks in the Mondrian visualization engine in our previous work [3]. We removed the bottlenecks by adding a "memo-

ization" mechanism which is a common technique applied to methods free of side effects to avoid unnecessary recalculations. Memoizing the method MOGraphElement>> bounds improved Mondrian performance by 43%. Another memoization of MOGraphElement>> absoluteBounds resulted in a speedup of 45% (for the UI thread this time). Comparing the amount of message sends with and without the optimization gives performance increases of the same range: the amount of messages sent with the bounds optimization is 42% less than the non-optimized version and the it is about 44% for the absoluteBounds optimization.

3.5 Cost of counting method invocation

Counting the number of executed send bytecode instructions is cheap. We measure the execution time of each of the 16 applications with and without the presence of message counting. Appendix B reports our results. Each measurement is the average of 5 executions. The overhead is computed as *overhead = (time on modified VM - time on normal VM) / time on normal VM * 100*.

The cost of message counting is almost insignificant. The execution time variation ranges from 0% to 0.02%. These results are not surprising actually. Message counting is simple to implement within the virtual machine. At each send bytecode a global variable is incremented. This is a cheap operation compared to the complex machinery to lookup method, interpret bytecode and to manage the memory. The execution time variation we have determined on a non-jitted virtual machine is of the same range on Cog.

4 Counting Messages to Identify Execution Bottlenecks

CPU profilers aim at identifying methods that consume a large share of the execution time. These methods are likely to be considered for improvement and optimization, aiming at reducing the total program execution time. This section considers counting message as a proxy to find runtime bottlenecks, which answer to the research question C.

4.1 A method as an execution bottleneck

A method is commonly referred as an execution bottleneck when it is perceived as taking a "lot of time", or more time that it should. The intuition we will elaborate on is that if a method is slow then it is likely to be the culprit of sending (directly and indirectly) "too many" messages.

Sending "too many" messages may not be the only source of slow down. An excessive use of the garbage collector and numerous invocations of the primitives offered by the virtual machine are likely to play a role in the time taken for a program to execute. A program that intensively uses files or the network may spend a significant amount of time for executing the corresponding primitives. In Pharo executing a primitive suspends the program execution and resumes it once the primitive has completed. It is easy to consider a program that sends few messages but makes a great use of primitives: the program can take a long time to execute with a few of sent messages. However, as we will see in the

coming sections, the perturbation that may be induced by primitive executions still makes message counting more advantageous than execution sampling for all the applications we have considered.

4.2 Method invocations per method

Counting the number of messages sent by a particular method is an essential step to compare the execution sampling with the message counting profiling techniques.

Example. Consider the following piece of code:

```
MOGraphElement>> addEdge: anEdge
  anEdge target addIncomingEdge: anEdge.
  anEdge source addOutgoingEdge: anEdge.
  self edges add: (anEdge setOwner: self)
```

The method addEdge: contains 7 message sends in its method body. Considering the calls made by the method calls shown in **bold**, just invoking addEdge: send 89 messages in total. Not all of these messages target a method in Mondrian, the application from which this code excerpt belongs to. The message edges returns the collection of defined edges.

Principle. Counting the number of emitted messages for each method implies to associate to each method the number of messages sent by the method and to increase it at each invocation. Most code instrumentation library and tools, including most aspect-oriented programming ones, easily realize this. The instrumentation we consider for each method of the application to be profiled is done as follows:

```
CompteurMethod>> run: methodName with: listOfArguments in: receiver
  | oldNumberOfCalls v |
  oldNumberOfCalls := self getNumberOfCalls.
  v := originalMethod valueWithReceiver: receiver arguments: listOfArguments.
  numberOfCalls := (self getNumberOfCalls - oldNumberOfCalls) + numberOfCalls - 5.
  ^ v
```

Compteur is the implementation of our message-based code profiler for Pharo. An instance of CompteurMethod is associated to each method of the application to be profiled. At each method invocation, the method run:with:in: is executed to increase the variable numberOfEmittedCalls defined in the CompteurMethod instance. The number of times a particular method is executed is associated to this method. Note that we do not instrument the whole system, but just the application we are interested in finding execution bottleneck in. The method getNumberOfCalls realizes a primitive operation defined in the virtual machine to obtain the current number of message sends. It is defined as:

```
CompteurMethod>> getNumberOfCalls
  <primitive: 556>
```

In Pharo, primitives may be identified with an integer. The primitive to retrieve from the virtual machine the amount of code sent messages is identified with 556. The instrumentation itself sends 5 messages: valueWithReceiver:arguments:, withArgs:executeMethod: and the second getNumberOfCalls, plus 2 messages sent by valueWithReceiver:arguments:, not presented here. We therefore need to subtract 5 to the number of calls.

4.3 Method execution time and number of message sends

The total execution time of an individual method is correlated with the number of messages that are directly and indirectly sent by the method. In this section, we focus on a single application, Mondrian. The result we have found equally applies to other applications.

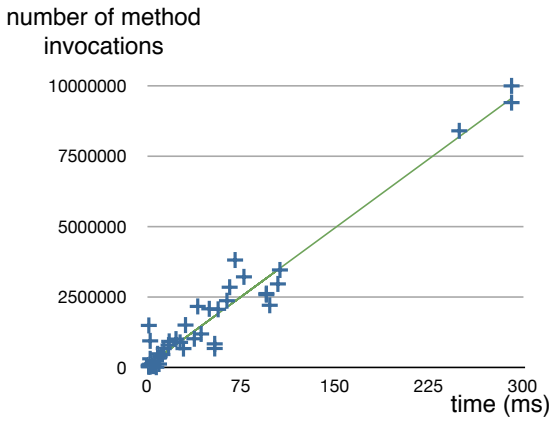

Fig. 2. Methods of Mondrian.

Figure 2 plots the methods of the Mondrian application according to their execution time in milliseconds with the number of sent messages. Note that we consider the total execution time and the total amount of message sends. This means that if a method is invoked 100 times for which each execution takes 2 ms and send 5 messages, then the method is plotted as the point $(200, 500)$. The graph shows that sending a message is almost constant in time: an increase of the amount of messages is related to a proportional increase of the method execution time.

Similarly to when we studied applications execution (Section 3.1), the regression model indicates that the large majority of methods forms a straight line (Figure 2), confirmed by a correlation of 0.97.

The equation of the regression line is $\hat{y} = 31\ 811.38\ x$. Figure 2 gives this line. We see that the slope of the regression line is about 3.4 greater than the slope we found when we studied application executions (Section 3.1). The reason

stems from the cumulative effect of recursive calls. To give a feeling of why this happens, consider the following two methods:

```
MOGraphElement>> bounds
  | basicBounds |
  boundsCache ifNotNil: [ ^ boundsCache ].
  self shapeBoundsAt: self shape ifPresent: [ :b |^ boundsCache := b ].
  ...

MONode>> startPoint
  ^ self bounds bottomCenter
```

The method bounds sends 239 direct and indirect messages. The method startPoint sends 2 direct calls. But since it invokes bounds and bottomCenter (which sends 27 messages), in total, startPoint sends $2 + 27 + 239 = 268$ messages. The method startPoint cumulated the amount of messages recursively sent.

4.4 Precision of message counting

To assess the stability and accuracy of message counting over execution sampling we will compare a list of profiles made with message counting and execution sampling. The idea is to numerically assess the variability of the method ranking against multiple profiles of the same code execution. We will then characterize a stable set of profiles with a constant method ranking.

Stability of profiles. We have profiled Mondrian 20 times: 10 using execution sampling and 10 using message counting. Each profile provides a ranking of the methods executed when running the unit tests. For space preservation, Table 1 gives an excerpt of our measurements only: the first 9 methods (name have been shortened into m1...m9) are ranked for 5 profiles. Methods are ranked against their execution time. The method ranked first is the one that has the greatest share of the CPU execution time. The method ranked last is the one that has consumed the CPU the less. The 5 profiles are obtained with MessageTally. As stated earlier (Section 2), due to the high and sensitive dependency on the running environment, not all the methods are equally ranked. Quantifying the variation of the method ranking for a set of profiles is the topic of this section.

For each method, we compute the standard deviation of the ranking (s_{es}) to estimate ranking variability. We have $s_{es}(\mathsf{m}) = 0$ if the method m is ranked always the same along the profiles. The greater s_{es} is, the greater the variability of the ranking.

The stability of a set of profiles depends on the method ranking variability. However, not all methods deserve to be considered the same way. We use the discounted cumulated gain [11] to give a discount to weight the ranking. The point of a discount is that the greater the ranked position of a method, the lesser valuable it is for the user, because the less likely it is that the user will ever consider the method as a bottleneck. A discounting function is needed which progressively reduces the method score as its rank increases. We weight a method ranked n as $w(n) = 1/ln\ (n + 1)$. We define the instability of the set of profiles

	m1	m2	m3	m4	m5	m6	m7	m8	m9
Profile 1	1	2	3	4	5	6	7	8	9
Profile 2	1	2	3	4	6	5	10	12	7
Profile 3	1	2	3	4	6	5	10	12	7
Profile 4	1	2	3	4	5	6	9	7	13
Profile 5	1	2	3	5	6	4	9	12	7
Average	1	2	3	4.1	5.4	5.5	8.9	10.4	8.2
Stand. Dev. s_{es}	0.000	0.000	0.000	0.316	0.516	0.707	1.197	1.955	1.989

Table 1. Ranking of the first 9 methods of Mondrian for 5 profiles (execution sampling).

P as $\psi(P) = \sum_{i=1}^{n} s_{es}(i) * w(n)$, the sum of the pondered deviations. According to the excerpt given in Table 1, we have $\psi(P) = 0\frac{1}{ln(0+1)} + ... + 0.316\frac{1}{ln(4+1)} + 0.516\frac{1}{ln(5+1)} + ... + 1.989\frac{1}{ln(9+1)} = 3.177$. A perfectly stable set of profiles P has the value $\psi(P) = 0$.

Experiment setting. We have profiled 20 times each application γ in the execution context c. We have $\gamma \in \Gamma$, where Γ is the list of applications given in Appendix A. 10 of these profiles were obtained using the standard execution sampling. We refer to these 10 profiles as $P_{\gamma,c}$. The 10 remaining profiles were obtained using message counting, referred as $Q_{\gamma,c}$. We have chosen to consider the same amount of methods for each applications since not all the applications have the same code size. We consider the first 100 ranked methods only. Taking the 100 ranked methods looks reasonable, since it is unlikely that a programmer will go over that amount of methods to find a bottleneck method. As previously, running the unit tests produces the profiles. We obtained surprising results as described below.

Poor stability of execution sampling. The method ranking against the execution time is not constant: each new profile gives a slightly different method ranking. For example, for 8 of the 10 profiles of $P_{PetitParser,c}$, the method ranked #5 that takes the greatest amount of time to execute is PPPredicateTest>> testHex. However, in the 2 remaining profiles, this method is ranked 35 and 36 (!). After an examination of the tests to make sure they do not randomly generate data, we speculate that this punctual odd ranking is due to a mixture of the problems phrased at the beginning of this article (Section 2). This kind of variability in the method ranking is hardly avoidable, even though we took a great care of garbage collecting the memory and releasing unwanted object references between each profile.

The greatest instability of the set of profiles we obtained is for PetitParser. We refer to this set of profiles as $P_{PetitParser,c}$obtained for the execution context c. The instability of this set is $\psi(P_{PetitParser,c}) = 1468$. About 3 times higher than the average of ψ for the remaining applications (they all vary between 500 and 600). PetitParser makes heavy uses of stream and string processing, which

perturbs MessageTally, the standard execution sampling profiler of Pharo, for the same reasons mentioned in Section 3.2 (use of short methods).

To give a reference point, we artificially build a random data set R on which we can compare ψ of the applications we profile: we randomly generate 10 random rankings for 100 methods. The instability of the random method ranking is $\psi(R) = 852$ (average obtained from 5 generations). The worse stability we obtained is 1468 which means that MessageTally is less stable than random method ordering when comparing the ranking of 100 methods for the weight function we use. About 80% of our set of profiles are around 550 which is close to a random method ranking limit.

Reducing the number of considered methods also reduce the variability of ψ. We define $\psi^{10}(P_{\gamma,c})$ over the first 10 methods given by a set of profiles P. By only considering the first 10 methods that consume the most resources, we intuitively expect execution sampling to be more stable. For our random set of profiles, we have $\psi^{10}(R) = 173$ and $\psi^{10}(P_{PetitParser,c}) = 11$. All the remaining ψ^{10} range from 3 to 5. Execution profiling behaves equally well: we have $\psi^{10}(P_{\_,c}) = 0$.

Perfect stability of message counting. The profiles obtained with message counting has a ψ of 0 for each of the applications we have profiled. This means that the 10 profiles we made for each application does not show a variation in the method ranking along the number of sent messages. Even though we have seen that the number of method invocations slightly varies (Section 4.2), the data we collected from this experiment shows that this does not impact the method ranking in all our experiments. Profiling multiple times always identically ranks the methods.

Even for a small amount of considered methods, the stability execution sampling does not equal the one of message counting. The stability of message counting clearly outperforms execution sampling.

4.5 Cost of the instrumentation

Determining the amount of sent messages for each method requires a complete instrumentation of the application to be profiled. This instrumentation induces an overhead. The cost of the instrumentation closely depends on the infrastructure used for code transformation. We have used the Spy framework [2]. To evaluate our implementation, we have performed two set of measurements. For each application, we run its associated unit tests twice, with and without the instrumentation. Appendix C presents our results.

Running the unit tests while counting message sends for each method has an overhead that ranges from 2% to 2524%. This overhead includes the time taken to actually instrument and uninstrument the application. When the unit test takes a short time to execute, then the instrumentation may have a high cost. The worse cases are with XMLParser and AST. AST's unit tests take 37 ms to execute. It takes 971 ms with the instrumentation, representing an overhead of 2 524%. The AST package is composed of 76 classes and 1 246 methods. XMLParser's unit tests take 36 ms to execute. The package is composed of 47 classes and 785 methods. Since XMLParser is smaller than AST, the overhead of the instrumentation is also smaller.

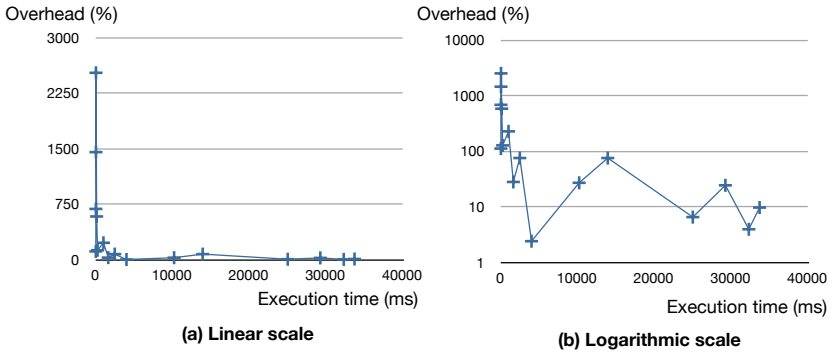

Fig. 3. Ratio between overhead and execution time.

The table given in Appendix C shows a general trend: the longer the unit tests take to execute, the smaller the instrumentation overhead is. Figure 3 represents the ratio of the overhead with the test execution time. The left hand-side presents this ratio with a linear scale. The right-hand side gives the same graphic, with a logarithmic scale for the overhead. Each cross is a couple *(execution time, overhead)*, representing an application. Above an execution time of approximately 5 seconds, determining the number of message sends per method has an overhead of less than 100%, which represents twice the execution time of the unit tests. In practice, this is acceptable in most of the situations we have experienced.

DSM has an overhead of 2.4%, the smallest overhead we measured. The reason for this low overhead is that most of the logic used by the DSM package is actually implemented in Famix, a different package. When DSM is the only package instrumented, the overhead is low since most of the work happens in a different package, itself uninstrumented.

5 Properties

We revise the issues encountered with execution sampling that we previously enumerated (Section 2) against the message counting technique described above.

No need of sampling. Message counting provides an exact measurement of a particular execution. The measurement is solely obtained by counting the amount of message sends. Message counting therefore does not depend on thread support or advanced reflective facilities (e.g., MessageTally heavily relies on threads and runtime call stack introspection) or sophisticated support of the virtual machine (e.g., the JVM offers a large protocol for profiling agents). As described in Section 6, adapting a virtual machine to count send bytecode instructions may require a few dozen lines of code for a non-jitted virtual machine.

Executing environment. Message counting is not influenced by the thread scheduling and memory management. A profile is determined by the amount of message sends, a metric that is related but not dependent on the time. The benefit is to be able to compare profiles obtained from different executing environments. For the applications we have considered, sending messages is correlated with the execution time. As we have shown, this means that the execution time can be easily approximated from the number of messages.

Stable measurements. Measurements obtained from message counting are significantly more stable than when obtained from execution sampling. Even though the exact amount of message sends may vary over multiple executions (partly due to the hash values given by the virtual machine), the metric is stable and reproducible as long as the potential sources of execution indeterminism are identified.

No need of long profiling time. Contrary to execution sampling, message counting is well adapted for short profiles since an exact value is always returned. One compeling application of this property is asserting execution time when writing tests. We have produced an extension of unit test that offers a new kind of assertion: assertIs:fasterThan: to compare execution time.

We have written a number of tests that define time execution invariant. One example for Mondrian is (the difference between the two executions is shown in **bold**):

```
MondrianSpeedTest>> testLayout2
  | view1 view2 |
  "Collection and all its subclasses"
  view1 := MOViewRenderer new.
  view1 nodes: (Collection allSubclasses).
  view1 edgesFrom: #superclass.
  view1 treeLayout.

  "All the subclasses of Collection"
  view2 := MOViewRenderer new.
  view2 nodes: (Collection withAllSubclasses).
  view2 edgesFrom: #superclass.
  view2 treeLayout.

  self assertIs: [ view1 root applyLayout ] fasterThan: [ view2 root applyLayout ]
```

The code above says that computing the layout of a tree of n nodes is faster than with $n + 1$ nodes. The difference between these two expressions is just the message sent to Collection. Being able to write test on short execution time is a nice application of message counting. As far as we are aware of, none of the mainstream testing frameworks is able to define assertions to compare short execution time.

6 Implementation

Compteur is an implementation of the message counting mechanism for Pharo. It comprises a new virtual machine and a profiler based on the Spy profiling framework [2].

Virtual machine. The modification made in the virtual machine is lightweight: a global variable initialized to 0 is incremented each time a send bytecode is interpreted. In the non-jitted Pharo virtual machine, the increment is realized in the part of the bytecode dispatch switch dedicated to interpret message sending. In the jitted Cog virtual machine, the preamble of the method translated in machine code by the JIT compiler realizes the increment.

The maximum value of a small integer in Pharo is 0x3F FF FF FF (∼1 073 M). Over this value, an integer is represented as an instance of the LargeInteger class, which is slow to manipulate within the virtual machine. The current Pharo virtual machine (5.7beta3) executes approximately 12 M message sends per second on micro benchmarks[13]. This means that the range of the Pharo integer values may be exhausted after 90 seconds (1, 073 / 12).

Using a 64 bits integer is not an option since Pharo is designed to run on 32 bit machines. We therefore use two small integers to encode the number of sent messages . The maximum amount of messages to be counted is $\sim 1.152 * 10^{18}$. Even at full interpretation speed, this value is not reached after 2 millions of hours, which is largely enough for the amount of profiling time we are considering.

The global message counter is made accessible within our profiler written in Pharo via primitives. The counter is reset via a dedicated primitive.

Bytecode Instrumentation. To obtain the amount of message sends per method, the application has to be instrumented to capture the value of the global counter before and after executing the method, as explained in Section 4.2. Using the Aspect-Oriented-Programming terminology [12], such instrumentation is easily realized with an around or a before and an after advice.

7 Discussion

The design of our approach is the result of a careful consideration of different points.

Modifying the virtual machine. Even though the modification we made in the virtual machine is relatively lightweight, we are not particularly enthusiast about producing a new virtual machine since it does not favor a large adoption among the Pharo community. People are often reluctant from using non-standard tools, even if the benefits are strong and apparent.

We have not found a satisfactory alternative. As an initial attempt before we realize the work presented in this paper, we made a profiler that counts only the messages sent by the application, and not by dependent libraries and the runtime.

[13] Result of the standard 0 tinyBenchmarks micro benchmark.

Only the application would be instrumented and the virtual machine would be left unmodified. We discovered that the information we gathered were not enough to demonstrate the properties presented in this paper. As soon as the execution flow leaves the application, no information is recorded until it returns to the application. Since it cannot be accurately predicted how long the execution flow will spent outside the application, we could not establish a correlation between the amount of messages solely sent by the application and execution time.

Instrumentation. Our approach requires an instrumentation of the application to be profiled: only the methods defined in the application we wish to improve its execution need to be instrumented.

Instrumenting the complete system has not proven to be particularly useful or possible in our situation: (i) if an execution slowdown is experienced, there is no need to look for its cause outside the application we are actually considering; (ii) instrumenting the whole system has a significant runtime cost; (iii) this easily leads to meta-circularity issues since our profiler shares the same runtime than the profiled application. Even if recent advances in instrumentation scoping have been proposed [19], this increases the complexity of the implementation without a clear benefit. Efficiently handling metacircularity is necessary to profile the profiler itself. However, since the implementation of Compteur is not particularly complex, we have not felt the need to do so.

Special messages. For optimization purpose, not all messages are sent in Pharo. According to the name of the message being sent, the Pharo compiler may decide to transform the message sent into a particular sequence of bytecode instructions. Simple conditions are realized with the message ifTrue:ifFalse: and two block arguments sent to a boolean. For example, the expression (1 < 2) ifTrue: ['Everything is okay'] ifFalse: ['Something wrong'] is translated into the sequence:

```
76 pushConstant: 1
77 pushConstant: 2
B2 send: <
99 jumpFalse: 27
21 pushConstant: 'Everything is okay'
90 jumpTo: 28
20 pushConstant: 'Something wrong'
87 pop
78 returnSelf
```

According to the design of the Pharo virtual machine, a jump bytecode is more efficient than a send. In the whole Pharo library, approximatively 62% of all message sends contained in the source code are translated into send bytecode instructions. The correlation we established between execution time and message sends is strong, even if 38% of message sends are not translated into send bytecode instructions.

8 Related Work

The work presented in this paper is not the first attempt at finding an alternative to execution sampling. We are however not aware of any work which studied the amount of method invocations.

Bytecode instruction counting. Camesi *et al.* [4] pioneered the field by investigating the use of bytecode instruction counting as an estimation of real CPU consumption. For all the platforms they have considered, there is an application-specific ratio of bytecode instructions per unit of CPU time. Such bytecode rate can be used as a basis for translating a bytecode instruction value into the corresponding CPU consumption.

We partly share some of their results. We indeed have identified a message rate, however this rate is attached to a particular executing platform, and not to an application.

Dynamic bytecode instrumentation. Instrumentation-based profile is known to have a high overhead. However, such overhead are reduced by instrumenting only the subset of the application, where a bottleneck is known to be for example. Dmitriev [6] proposes that for a given set of arbitrary "root" methods, instrumentation applies to the callsubgraph of the roots only.

Dmitriev observed that this approach generally works much better for large applications, than for small benchmarks. The reason is that additional code and data become negligible once the size of the profiled application goes above a certain threshold. Message counting follows a similar idea. Only a subset of the system needs to be instrumented. However, message counting behaves perfectly well for small benchmarks.

Hardware Performance Counters. Most modern processors have complex microarchitectures that dynamically schedule instructions. These processors are difficult to understand and model accurately. For that purpose, they provide hardware performance counters [1]. For example, Sun's UltraSPARC processors count events such as instruction executed, cycles executed and many more.

With message counting we exploit the same kind of information, but obtained from what is being executed on the Pharo virtual machine.

9 Conclusion

A code profiler provides high-level snapshots of a program execution. Theses snapshots are often considered as the only support to effectively identify and understand the cause of a slow execution. Whereas execution sampling is a widely used technique among code profilers to monitor at a low cost, it brings its own bag of limitations, including fragility against the execution environment, non-determinism and inability to relates profiles obtained from different platforms.

We propose counting method invocations as a more advantageous profiling criterion for the Pharo programming language. We have shown that having method invocation as the exclusive computational unit in Pharo makes it possible

to correlate message sending and execution time, both for application in a whole and individual methods.

We believe that code profiling has not received the attention it deserves: execution sampling uses stack frame identifiers, which essentially ignore the nature of object-oriented programming. In their large majority, code profilers profile object-oriented applications pretty much the same way that they would profile applications written in C. We hope the work presented in this paper will stimulate further research of the field to give more importance to objects than to low implementation considerations.

Acknowledgment. We thank Mircea Lungu, Lukas Renggli and Romain Robbes for their comment on an early draft of the paper. We particular thank Walter Binder for his multiple discussions and advices. Our thank also goes to Eliot Miranda for his help on porting COMPTEUR to Cog, the jitted virtual machine of Pharo. We thank Gilad Bracha for the fruitful discussion we had. We gracefully thank María José Cires for her help on the statistical part.

References

1. Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM.
2. Alexandre Bergel, Felipe Bañados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. In *Smalltalks 2010*, 2010. To appear.
3. Alexandre Bergel, Romain Robbes, and Walter Binder. Visualizing dynamic metrics with profiling blueprints. In *Proceedings of 48th International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE'10)*, volume 6141 of *LNCS*, pages 291–309. Springer, 2010.
4. Andrea Camesi, Jarle Hulaas, and Walter Binder. Continuous bytecode instruction counting for cpu consumption estimation. In *Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, pages 19–30, 2006. IEEE Computer Society.
5. Amer Diwan, Han Lee, and Dirk Grunwald. Energy consumption and garbage collection in low-powered computing. Cu-cs-930-02, University of Colorado, 2002.
6. M. Dmitriev. Profiling java applications using code hotswapping and dynamic call graph revelation. In *Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150. ACM Press, 2004.
7. Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, March 2006.
8. Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Damien Cassou. Reusing and composing tests with traits. In *Proceedings of 47th International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE'09)*, pages 252–271, jun 2009.
9. David Freedman, Robert Pisani, and Roger Purves. *Statistics, Third Edition*. W. W. Norton & Company, 1997.
10. Aloke Gupta and Wen-Mei W. Hwu. Xprof: profiling the execution of x window programs. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '92/PERFORMANCE '92, pages 253–254, 1992. ACM.

11. Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20:422–446, October 2002.
12. Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, 1997.
13. L. Kleinrock and R. R. Muntz. Processor sharing queueing models of mixed scheduling disciplines for time shared system. *J. ACM*, 19:464–482, July 1972.
14. Prem S. Mann. *Introductory Statistics*. Wiley, 2006.
15. Robert Cecil Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.
16. Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, 2006. ACM Press.
17. 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, 2009. ACM.
18. 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, 2010. ACM.
19. Éric Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 37–48, Rennes and Saint Malo, France, March 2010. ACM Press.
20. Jia Yu, Jun Yang, Shaojie Chen, Yan Luo, and Laxmi Bhuyan. Enhancing network processor simulation speed with statistical input sampling. In *Proceedings of High Performance Embedded Architectures and Compilers*, volume 3793 of *LNCS*, pages 68–83. Springer, 2005.

A Regression Linear Material

This section contains the relevant data and theoretical tools to conduct the regression linear model described in Section 3.1 and Section 4.3.

Measurements. The following table lists 16 representative Pharo applications. Each of these applications covers a particular aspect of the Pharo library and runtime. *Collections* is an intensively used library to model collections. *Mondrian*, *Glamour* and *DSM* make an intensive use of graphical primitives and algorithms. *Nile* is a stream library based on Traits [7]. *Moose* is a software analysis platform which deals with large models and files. Mondrian and Moose heavily employ hash tables as internal representation of their models. *SmallDude*, *PetitParser*, *XMLParser* heavily manipulate character strings. *Magritte* and *Famix* are metamodels. *ProfStef* intensively makes use of reflection. *Network* uses primitive in the virtual machine. *ShoutTest* and *AST* heavily parse and manipulate abstract syntax trees. *Arki* is an extension of Moose which performs queries over large models.

These applications cover the features of Pharo that are intensively used by the Pharo communities: most of the applications are either part of the standard

Pharo runtime or are among the 20 most downloaded applications. Not all the set of primitives offered by the virtual machines are covered by the applications. For example, none of them make use of sound. We are not aware of any application that intensively use Pharo musical support.

For each of these applications, we report the time taken to run its corresponding unit tests (**time taken (ms)**) and the amount of sent messages (# sent messages). These reported results are obtained after 10 runs. For each of these two measurements, we compute the standard deviation (not reported here) and normalize it. We have $c_{time} = s_{TimeTaken} * 100/TimeTaken$ and $c_{messages} = s_{messages} * 100/messages$. These applications were run on a virtual machine modified to support our message counting mechanism.

Application	**time taken (ms)**	**# sent messages**	c_{time} (%)	$c_{messages}$ (%)
Collections	32 317	334 359 691	16.67	1.05
Mondrian	33 719	292 140 717	5.54	1.44
Nile	29 264	236 817 521	7.24	0.22
Moose	25 021	210 384 157	24.56	2.47
SmallDude	13 942	150 301 007	23.93	0.99
Glamour	10 216	94 604 363	3.77	0.14
Magritte	2 485	37 979 149	2.08	0.85
PetitParser	1 642	31 574 383	46.99	0.52
Famix	1 014	6 385 091	18.30	0.06
DSM	4 012	5 954 759	25.71	0.17
ProfStef	247	3 381 429	0.77	0.10
Network	128	2 340 805	6.06	0.44
AST	37	677 439	1.26	0.46
XMLParser	36	675 205	32.94	0.46
Arki	30	609 633	1.44	0.35
ShoutTests	19	282 313	5.98	0.11
Average			*13.95*	*0.61*

The source code of each of these applications is available online on the SqueakSource Pharo forge.

Estimating the sample regression line. For sake of completeness and providing easy-to-reproduce results, we provide the necessary statistical material. Complementary information may be easily obtained from standard statistical books [9].

For the least squares regression line $\hat{y} = a + b\,x$, we have the following formulas for estimating a sample regression line:

$$b = \frac{SS_{xy}}{SS_{xx}}$$
$$a = \overline{y} - b\,\overline{x}$$

where $\overline{y}$ and $\overline{x}$ are the average of all y values and x values, respectively. The y variable corresponds to the **# sent messages** column and x to **time taken (ms)** in the table given above.

$$SS_{xy} = \sum xy - \frac{(\sum x)(\sum y)}{n}$$
$$SS_{xx} = \sum x^2 - \frac{(\sum x)^2}{n}$$

where n is number of samples (i.e., 16, the number of applications we have profiled). *SS* stands for "sum of squares."

The standard deviation of error for the sample data is obtained from:

$$s_e = \sqrt{\frac{\sum SS_{yy} - b\ SS_{xy}}{n-2}}$$
$$where\ SS_{yy} = \sum y^2 - \frac{(\sum y)^2}{n}$$

In the above formula, $n-2$ represent the degrees of freedom for the regression model.

Finally, the standard deviation of b is obtained with:

$$s_b = \frac{s_e}{\sqrt{SS_{xx}}}$$

B Cost of the Virtual Machine Modification

Application	**Normal VM (ms)**	**VM with Compteur (ms)**	**overhead (%)**
Collections	32 317	32 323	0.02
Mondrian	33 719	33 720	0
Nile	29 264	29 267	0.01
Moose	25 021	25 023	0.01
SmallDude	13 942	13 944	0.01
Glamour	10 216	10 218	0.02
Magritte	2 485	2 485	0
PetitParser	1 642	1 642	0
Famix	1 014	1 015	0.1
DSM	4 012	4 013	0.02
ProfStef	247	247	0
Network	128	128	0
AST	37	38	2.7
XMLParser	36	36	0
Arki	30	30	0
ShoutTests	19	19	0

C Cost of the Application Instrumentation

Application	**No inst (ms)**	**Inst. (ms)**	**overhead (%)**
Collections	32317	33590	3.94
Mondrian	33719	36983	9.68
Nile	29264	36387	24.34
Moose	25021	26652	6.52
SmallDude	13942	24467	75.49
Glamour	10216	12976	27.02
Magritte	2485	4361	75.51
PetitParser	1642	2102	28.01
Famix	1014	3327	228.07
DSM	4012	4108	2.40
ProfStef	247	562	127.47
Network	128	875	583.87
AST	37	971	2524.00
XMLParser	36	559	1452.78
Arki	30	236	685.71
ShoutTests	19	40	111.76