Accurate VM profiler for the Cog VM

Sophie Kaleba[1], Clément Béra[1], Alexandre Bergel[2]
[1]INRIA- Lille Nord Europe, France
[2]Pleiad Lab, DCC, University of Chile, Santiago, Chile

Abstract

* A REECRIRE *

Code profiling is critical when it comes to improve the performance of your applications. This applies of course for Smalltalk applications and a lot of different profiling tools are already available, such as MessageTally, GadgetProfiler, VMProfiler. The last one can especially provide significant data to help you tune the virtual machine (VM) settings for performance. However, this VM profiler is only available on Squeak. [ncessaire? : This is getting critical as the number of Pharo users is growing]. Besides, new optimisations are added to the Just-In-Time (JIT) compiler, that could compromise the relevance of the data provided by this profiler.

In this paper, we discuss these two problems and then we introduce the VM profiler for Pharo and the solution proposed to keep collecting relevant profiling data despite the enhancements of the JIT.

* A REECRIRE *

1. Introduction

Even though the technical specs of computers get more and more impressive, performance, both in terms of execution time and XXX, remains a major goal to be pursuied (atroce). [besoin d'une transition non moisie vers les VM] This applies of course to virtualised environments : assuring the performance of the virtual machine (VM) is critical when you aim for overall good performances. These performances can be assessed by metrics such as bytecodes-per-second and sends-per-second [reference : Modern Problems for the Smalltalk VM]

Cog [**?**] is a virtual machine designed for Smalltaltk, and currently used for various Smalltalk-based languages such as Pharo [**?**] or Squeak [**?**]. It features (what exactly? JIT, optimisations such as inline-caching...?) [reference - article about the features of cog?] Aiming for performance, it is crucial to know where the time is spent in the VM during execution : indeed, it helps identifying hotspots and where to tune the VM settings to actually get better results.

These critical informations can be collected by profiling code, to get a precise idea of the program behaviour. Such profiling tools are already available in Smalltalk, like MessageTally and VMProfiler : they provide statistical/graphical reports, showing the methods in which most of the execution time is spent, and how much of this time is spent in garbage collection **[reference needed]**. However, the VM profiler, unlike MessageTally, provides statistical data about the time spent in the VM, ie. time spent in the JIT-generated methods, and in the GC and interpreter.

Right now, the VM profiler cannot track down precisely where the time is spent when executing the code generated by the JIT. It can track down in which methods the time is spent, but it cannot track down in which part of those methods the time is spent. For example, let's say there is a frequently used method with multiple loops. The VM profiler can tell us that most of the time is spent in this method (it is indeed frequently used), but it cannot tell in which loop the time is spent.

This problem is more and more significant as new optimisations are added to the JIT, based on the work of Hölzle and Ungar [**?**]. The development branch of the JIT now features speculative inlining. In this context, the JIT generates a single machine code method for multiple bytecode methods (the method optimised and the inlined methods). The VM profiler shows that most of the time is spent in optimised code, but it is currently not possible to know in which inlined method most of the time is spent. So while we get a faster and more performant VM, we lose the ability to profile it in a relevant way.

To get relevant profiling data again, we have to tweak the VM profiler so it shows where specifically the time is spent in a method. After porting the profiler to Pharo **[justifier avant pourquoi Pharo? ou juste retirer cette partie]**, we use an API usually used for debugging, that maps machine code

pc to bytecode pc. This way, we can tell for each method appearing in the report in which bytecode range most time is spent.

2. Accurate profiling of jitted code

This section first defines the terminology used in the paper, then describes the existing VM profiler available in the Cog VM clients such as Squeak or Pharo and lastly states the problem analysed in the rest of the paper.

2.1 Terminology

Function. In the paper, we use the term *function* to discuss about executable code, in our case, method or block closures.

Bytecode function. The term *bytecode function* is used to talk specifically about the compiled function in the form of bytecode, for example, instances of `CompiledMethod` in the case of methods. Bytecode functions are regular objects accessible from the Squeak/Pharo runtime and are present in the heap with all other objects.

Native function. We use the term *native function* to discuss about the representation of a function generated by Cog's JIT, which includes the native code. Native functions are not regular objects and are allocated in a specific memory zone (called the *machine code zone*, which is executable.

2.2 Existing VM profiler

A VM profiler has been available for almost a decade in Squeak and has been recently ported to Pharo. The VM profiler allows to track down where the time is spent in the VM when executing a specific portion of code. The VM profiler computes where the time is spent in the compiled C code of the VM, in the VM plugins and in the native functions. All the results are available as a statistical report. A typical report includes two main sections, the time spent in generated code, *i.e.,* in native functions and the time spent in the compiled C code. The time spent in the compiled C code includes the time spent in the bytecode interpreter, in the garbage collector and in the just-in-time compiler.

Implementation. Implementation-wise, the VM profiler is a sampling profiler. When profiling is started, a separate high-priority OS thread is started and collect the instruction pointers of the VM OS thread in a large circular buffer at a cadence of 1.3GHz. Once profiling stops, a primitive method is available to gather the samples from the VM into a Bitmap. To understand to which functions the samples correspond to, the VM profiler requests:

- The symbol table of the VM executable and all external plugins.
- A description of the native functions currently present in the machine code zone

Each function (indifferently, a native function or a C function) is represented as a function symbol (the name of the function, either the Smalltalk function name with the method class and the selector or the C function symbol), a start address and the last address. The VM profiler uses these ranges to find out in which function each sample is, as shown in Figure 1. Then, the profiler generates a report, either in the form of a string or through a user interface.

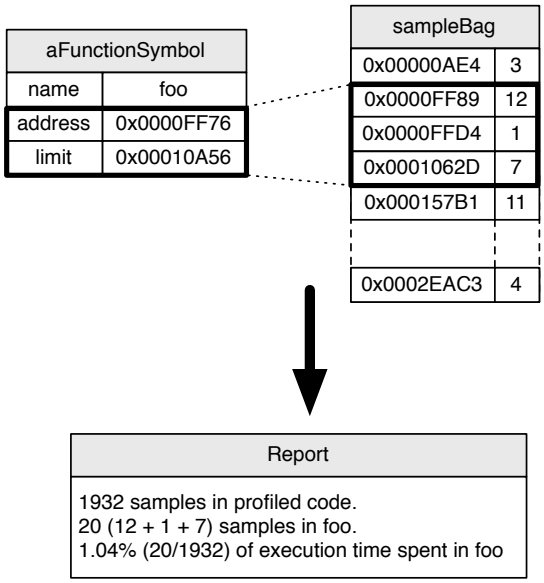

Figure 1: Mapping in original code

Primitive Cog Constituents. The primitiveCollectCogCodeConstituents provides the VM profiler with the description of the native functions currently present in the machine code zone it needs. This primitive answers an array of pairwise elements as shown in Figure 2. The first element of the pair is the name of the native function, the second one is its starting address in the code zone. It is typically called once the profiling has stopped and the samples has been gathered : the data provided will be mapped with the samples collected.

2.3 Debugger mapping

To be able to debug native function as if they were executed as a bytecode function by the bytecode interpreter, when Cog's JIT generates a native function for a given bytecode function, it generates a list of annotations [**?**] allowing to reconstruct the interpreter state of the function activation at any point where the code execution can be interrupted (message sends, conditional branch or back jumps). Each annotation includes a mapping between the pc in the machine code and the pc in the bytecode of the method. The VM is able to use these annotations to reify function activation and provide it in Squeak/Pharo.

Source code

```
foo
    self foo.
    self bar.
    self baz.
```

Bytecode

```
f17 FF self
```

Figure 3: Example method

Existing primitive result	Improved primitive result
result 1	result 2

Figure 4: Example of primitive results

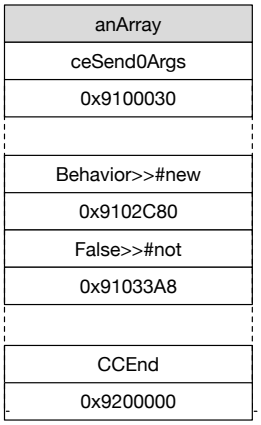

Figure 2: Cog constituents as answered by the primitive

2.4 Problem statement

The VM development is currently working on the implementation of an optimising Just-in-time compiler for the Cog VM, with optimisations such as speculative inlining, as described in the work of Hölzle and Ungar [**?**], in a similar way to production VMs such as Java's hotspot VM[1] [**?**] and Javascript's V8 engine[2] [**?**]. The optimising is planned as a bytecode functions to bytecode functions runtime optimising compiler, re-using the existing Just-in-time compiler as a back-end to generate native functions. In this context, optimised functions are present in the form of bytecode functions, using the extended bytecode set described in the work of Béra *et al.* [**?**]. When profiling optimised code for benchmarks such as the Games benchmark [**?**], the VM profiler now shows that all the time is spent in a single bytecode function (the bytecode function where the rest of the functions used by the benchmark are inlined). To improve performance and tune the optimising JIT, the VM development team requires more information about where the time is spent in optimised function, for example, in which range of bytecodes the time is spent.

How to provide accurate profiling information in large native functions profiled ?

To address this problem, we propose an implementation that take advantage of an API used for debugging, to map machine code pc to bytecode pc, to be able to identify in which bytecode range the time is spent in a function. The implementation is specific to the Cog VM, with a working version in both Squeak and Pharo. A similar design could apply in other VMs featuring similar debugging capabilities.

3. Solution

To profile accurately large native functions, we re-use the API available for debugging to identify in which section of the function the time is spent. The solution is split in two steps. First, we enhanced the primitive providing the description of the native functions present in the machine code zone to provide a mapping between machine code pc and bytecode pc in addition to the start address of the function. Second, we used the mapping to identify in which range of bytecodes each sample is.

[1] The hotspot VM is the default virtual machine for Java.

[2] V8 is mostly used as the Javascript engine for Google Chrome and NodeJS.

3.1 Improved primitive

In the improved version of the primitive, if the native function has at least one machine code pc mapped to a bytecode pc, the primitive answers for the function, instead of the start address, an array starting with the start address followed by pairs of machine code pc and the corresponding mapped bytecode pc.

+ EXAMPLE: decrire la methode ca fait 3 sends. Figure **??** decrit le bytecode. decrit la table. Figure **??**

code d'une methode. avec 3 sends. (4 mapped points)

Bytecode de la methode.

3.2 Accurate mapping and report

Expliquer en 1 paragraph

decrire figure

4. Example

Intro: this is a concrete example on a benchmark. (tinyBench ?)

montrer avant / apres.

Figure **??** describes bla.

5. Related Work

intro

5.1 Smalltalk profilers

*profiling in Pharo-Smalltalk In Squeak, 2 (?) profiling tools are available in the image: MessageTally and VM profiler. Both are sampling profilers, but they are not used for the

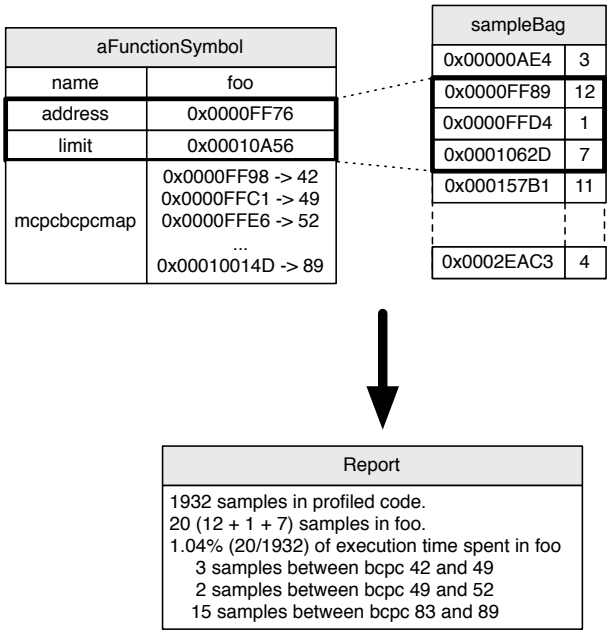

Figure 5: Mapping with new feature

Existing profiler report	Accurate profiler report
rapport 1	rapport 2

Figure 6: Example of profiling reports

same purpose. MessageTally = image side, VM profiler = vm side.

*messageTally [reference] description, comparison *spy [reference] description, comparison *gadgetProfiler [reference] description, comparison

5.2 Other VM profilers

jvisualvm

6. Conclusion and Future Work

Future work: UI, Windows. -¿ engineering.