A detailed VM profiler for the Cog VM

Sophie Kaleba, Clément Bera, Alexandre Bergel, Stéphane Ducasse

A detailed VM profiler for the Cog VM

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

Abstract

Code profiling enables a user to know where in an application or function the execution time is spent. The Pharo ecosystem offers several code profilers. However, most of the publicly available profilers (MessageTally, Spy, GadgetProfiler) largely ignore the activity carried out by the virtual machine, thus incurring inaccuracy in the gathered information and missing important information, such as the Just-in-time compiler activity.

This paper describes the motivations and the latest improvements carried out in VMProfiler, a code execution profiler hooked into the virtual machine, that performs its analysis by monitoring the virtual machine execution. These improvements address some limitations related to assessing the activity of native functions (resulting from a Just-in-time compiler operation): as of now, VMProfiler provides more detailed profiling reports, showing for native code functions in which bytecode range the execution time is spent.

1. Introduction

Although computers tend to get faster and faster, improving software performance, especially in terms of execution time, remains a major goal to be pursued when developing a software. This statement applies of course to virtualised environments and assuring the performance of **v**irtual **m**achine (VM) is critical when you aim for an overall good performance. Thus, it is crucial to know where the time is spent in the VM during execution: indeed, it helps identifying where to tune its settings to actually get better results.

To get a precise idea of the program behavior, critical information can be collected by profiling code. Such profiling tools are already available in Pharo, like MessageTally [BCDL13] and VMProfiler [Mir08b]: 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 [BCDL13]. However, VMProfiler, unlike MessageTally, provides statistical data about the time spent in the Cog VM. Cog [Mir08a] is a virtual machine designed for Smalltalk, and currently used for other similar languages, such as Pharo [BDN+09] and Squeak [BDN+07]. Cog features a bytecode interpreter and a **j**ust-**i**n-**t**ime compiler (JIT). A careful monitoring of the interpreter and the JIT is crucial to adequately estimate the time taken to execute a portion of code.

The existing VMProfiler cannot track down precisely where the time is spent when executing the code generated by the JIT. It tracks 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, assuming there is a frequently used method with multiple loops, VMProfiler mentions that most of the time is spent in this method (it is indeed frequently used), but it cannot mention 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 [HU94]. The development branch of the JIT now features speculative inlining. In this context, the JIT generates a single machine code method for multiple unoptimised 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 on. So while we get a faster and more performant VM, the profiler mostly ignores optimisations when computing and reporting its analysis.

To increase the level of detail of the profile, the existing VMProfiler has to be enhanced to show specifically where the time is spent in a method. To do so, we use the API usually used for debugging, that maps machine code program counter (pc) to bytecode program counter. This way, we can tell for each method appearing in the report in which bytecode range most of the time is spent.

In this paper, we will first discuss the existing Squeak VMProfiler, how it works and how the granularity of its reports

when optimised native code functions are profiled raises a problem. Then, we describe the proposed solution, a bytecode level profiling, to address this problem. Eventually, we mention other profiling tools in Smalltalk and other programming languages and compare them against VMProfiler.

2. Profiling jitted code

This section first defines the terminology used in the paper, then describes the existing VMProfiler available in the Cog VM clients such as Squeak or more recently Pharo, and the debugger mapping 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 refer to executable code, in our case, method or block closures.

Bytecode function. The term *bytecode function* is used to refer specifically to 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. These functions are executable by the VM.

Native function. We use the term *native function* to refer to 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. These functions are directly executable by the processor.

2.2 Existing VM profiler

VMProfiler has been available for almost a decade in Squeak and has been recently ported to Pharo. VMProfiler allows one to track down where the time is spent in the VM when executing a specific portion of code. VMProfiler 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 JIT.

Machine code zone. As depicted in Figure 1, the machine code zone is composed of three areas (in order, the numbers match the numbers on the figure):

1. The first area includes all the trampolines and enilopmarts generated at VM start-up. Trampolines are native code routines. Some of them are used as discovery routine at VM start-up to know which instructions the processor supports. The other trampolines are called from native functions, either to switch to the C runtime of the VM or just to execute specific uncommon code. Enilopmarts (trampoline written backwards) are native code routines called from the C runtime of the VM to switch to native functions generated by the JIT.
2. The second area is composed of native functions (CogMethods or CogFullBlocks, depending on if a method or a block closure is compiled) and **p**olymorphic **i**nline **c**aches (PIC) (ClosedPICs, PICs represented as a jump table up to 6 cases or OpenPICs, PICs represented as a hash map search with an 8 entry hash map). [HCU91]
3. The last area is a linked list of native functions or PICs referencing young objects. This list is used by the scavenger.

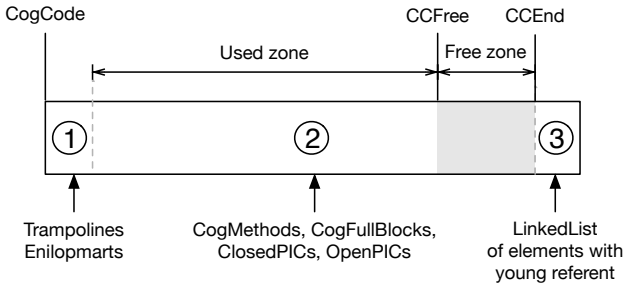

Figure 1: Machine code zone layout

During the runtime, part of the machine code zone is unused. When the machine code zone is full and a new function needs to be compiled, a code compaction happens, freeing a quarter of the machine code zone, using a least recently used naive algorithm. Hence, while running a normal application, up to a quarter of the machine code zone is free.

We use three keywords to identify different addresses in the machine code zone. *CogCode* is the beginning of the machine code zone, before the trampolines and enilmoparts. *CCFree* is the beginning of the unused part of the machine code zone. *CCEnd* is the last address of the machine code zone before the linked list of young referrers.

Implementation. Implementation-wise, VMProfiler is a sampling profiler. When profiling is started, a separate high-priority OS thread is started and collects the instruction pointers of the VM OS thread in a large circular buffer at an approximate cadence of 1.4kHz. Once the profiling ends (*i.e.,* once the profiled code has been executed), a primitive method is available to gather the samples from the VM into a Bitmap. To understand to which functions the samples correspond to, VMProfiler 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.

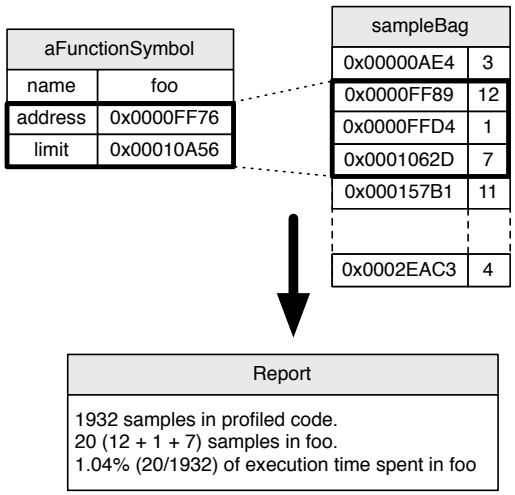

Figure 2: Mapping in original code

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. VMProfiler uses these ranges to find out in which function each profiling sample corresponds to, as shown in Figure 2. Then, the profiler generates a report, either in the form of a string or through a user interface.

Primitive Cog Constituents. The `primitiveCollectCogCodeConstituents` provides VMProfiler with the description of the native functions currently present in the machine code zone it needs. This primitive answers an array of pair-wise elements as shown in Figure 3. The first element of the pair is either :

- the name of a trampoline/enilopmart, or
- a function pointer, or
- the name of a selector (for PICs), or
- annotations (*i.e.,* CCFree, CCEnd).

The second item of the pair is the start address of this element in the machine code zone.

`primitiveCollectCogCodeConstituents` is called once the profiling samples has been gathered. These samples are mapped to the data answered by the primitive. For instance, if a sample is equal to `0x89012F0`, one can find out it refers to `Behavior>>new` thanks to the primitive, as showed in Figure 3.

2.3 Debugger mapping

To be able to debug native functions as if they were executed as bytecode functions by the bytecode interpreter, when Cog's JIT generates a native function for a given bytecode function, it generates a list of annotations [Bér16] allowing one to reconstruct the interpreter state of the function activation at any point where the code execution can be interrupted (message sends, conditional branches 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 activations and provide it to Pharo.

2.4 Problem

The VM development team is currently working on the implementation of an optimising JIT for the Cog VM. These optimisations include speculative inlining, as described in the work of Hölzle and Ungar [HU94], in a similar way to production VMs such as Java's hotspot VM (The hotspot VM is the default virtual machine for Java [PVC01]) and Javascript's V8 engine (V8 is mostly used as the Javascript engine for Google Chrome and NodeJS [Goo08]). The optimising JIT is designed as a bytecode functions to bytecode functions runtime optimising compiler, re-using the existing JIT as a back-end to generate native functions. In this context, optimised functions are present both in the form of

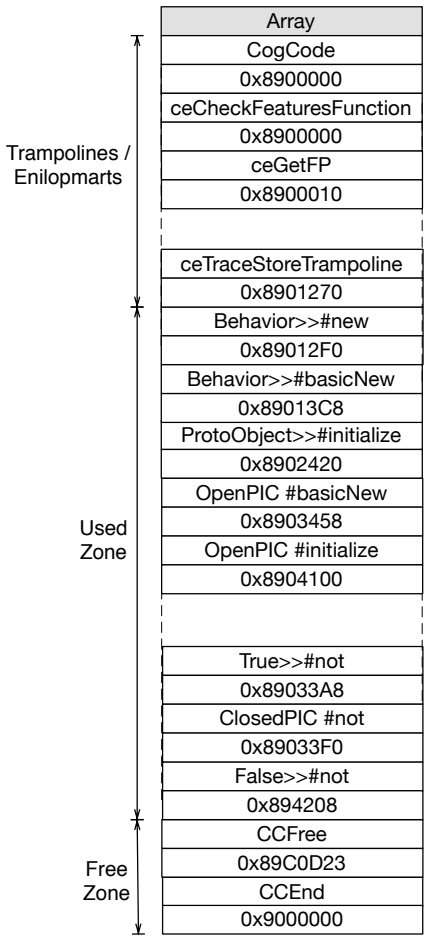

Figure 3: Cog constituents as answered by the primitive

Source code	Bytecode
foobarbaz	25 <4C> self
self foo.	**26 <80> send: foo**
self bar.	27 <D8> pop
self baz.	28 <4C> self
	29 <81> send: bar
	30 <D8> pop
	31 <4C> self
	32 <82> send: baz
	33 <D8> pop
	34 <58> returnSelf

Figure 4: Example method

bytecode functions and native functions, using the extended bytecode set described in the work of Béra *et al.* [BM14]. When profiling optimised code for benchmarks such as the Games benchmark [GB04], VMProfiler now shows that all the time is spent in a single function (the 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 functions, for example, in which range of bytecodes the time is spent.

Problem statement. *How to provide detailed profiling information in large native functions profiled?*

To address this problem, we propose an implementation that takes 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 accurately profile 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.

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.

For example, the function `foobarbaz` in Figure 4 sends 3 messages. Once translated to bytecode, there are indeed 3 send bytecodes, each responsible for the sending of a message (on bytecode pc 26, 29 and 32, in bold in Figure 4). For this function, the original primitive was answering 2 elements: the pointer to the native function and its start address in the machine code zone

As you can see in Figure 5, the improved primitive still answers 2 elements, but, while the first one remains unchanged and still refers to the name of the function, the second one is an array, because the 3 send bytecodes are mapped. The first element of this array is the starting address of the function in the machine code zone. The other elements come in pairs: the first one is the machine code pc, the second one is the bytecode pc. The results answered by the improved primitive are then used to determine bytecode ranges in the function. In Figure 5, there are 4 bytecode ranges, each delimited by a mapped bytecode pc.

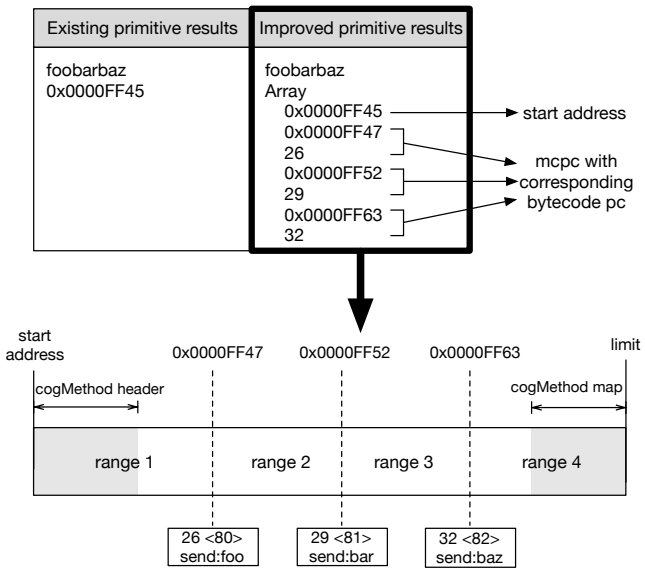

Figure 5: Example of primitive results

3.2 Accurate mapping and report

To compute the profiling statistics, the profiler uses the primitive to create a description of the native functions currently present in the machine code zone. Each function is represented by a functionSymbol object, characterized by the name of the function and its starting and limit addresses in the machine code zone. A new field in `FunctionSymbol` has been added to take the results of the modified primitive into account: *mcpcbcpcmap*, standing for machine code program counter - bytecode program counter map. This dictionary associates a machine code pc with a bytecode pc.

As shown in Figure 6, this new field helps with identifying where the execution time is spent. For instance, we know that `foobarbaz` starts at `0x0000FF45`, and that the first mapped bytecode pc (26) is at `0x0000FF47`: it means that the 12

samples within this address range will refer to the bytecode instructions between 1 and 26. The same applies for the other entries: the unique sample within the range `0x0000FF47` and `0x0000FF52` will refer to the bytecode instructions between 26 and 29.

In the Figure 6, 1932 samples were gathered in total and 20 were referring to `foobarbaz`. Among these 20 samples, 12 were referring to foo's bytecode pc between 1 and 26. Therefore, 60% of the time spent in `foobarbaz` was spent between these bytecode pcs.

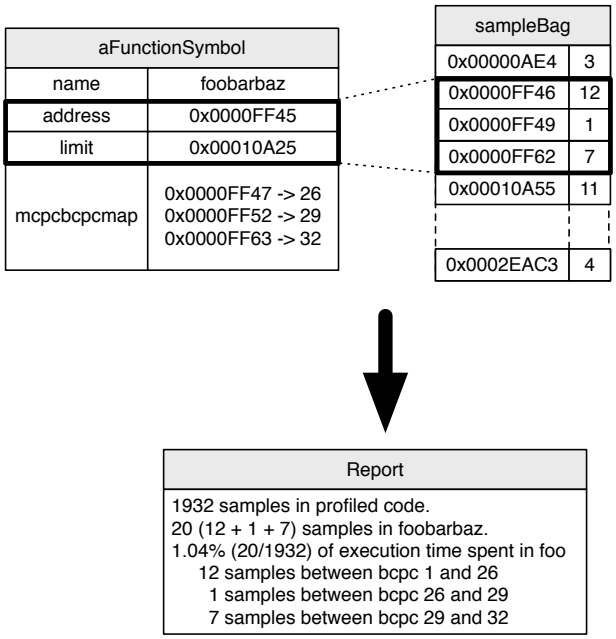

Figure 6: Mapping with new feature

4. Example

In this section, we present a concrete example of profiling a benchmark.

We profiled the following benchmark: `10 tinyBenchmarks` using first the existing VMProfiler, and using then the detailed VMProfiler. Figure 7 puts the two profiling reports side by side. Among the jitted methods, `Integer>>benchFib` was the one in which most of the execution time was spent (around 45% of the total time).

In the original version of the profiler (left-hand side of Figure 7), one cannot identify where 45% of the total execution time is spent. In the detailed version (right-hand side), however, the method is decomposed in 8 bytecode ranges: one can then identify in which bytecode range(s) most of the time is spent. Here, 57.95% of the time is spent in the entry. The next significant part of time is spent in the last bytecode instructions (12% starting from bytecode pc 38).

In the `Integer>>benchmark` function, most of the time is spent in the 74 -> 78 bytecode range, referring to the following bytecode instructions:

- `75 <6B> popIntoTemp: 3`
- `76 <13> pushTemp: 3`
- `77 <10> pushTemp: 0`
- `78 <B4> send: <=`

In the next bytecode instructions of the `Integer>>benchmark` method, one can find the `90 <A3> jumpTo: 76` instruction. It indicates that there is a loop between the 76 and 90 bytecode instructions. Thus, we can assume that the time is mostly spent in the 76, 77 and 78 bytecode instructions.

5. Related Work

The field of execution profiling is vast and has received a large attention from the research community. This section presents the different works related to the effort presented in this paper.

5.1 Standard Pharo profilers

Smalltalk, and therefore Pharo, offers a sophisticated reflective API. Threads are openly exposed and the stack for each active thread may be introspected. In addition, the next thread in the execution queue may be accessed. `MessageTally` and `AndreasSystemProfiler` are two standard profilers in Pharo that exploit this advanced reflective API. Both follow the same principle: a thread of high-priority is run and regularly it samples the thread queue. The frequency of the samples typically ranges from 1 to 10 milliseconds. After the program execution, frequency of method context frames is determined. Such frequency is used to estimate the time spent in each frame.

Both profilers essentially rely on the Smalltalk reflective API. Since most of the computation happens within the image, the overhead of the profiler is therefore likely to be expensive and intrusive (*e.g.,* time to execute an application is longer when being profiled). It is known that getting a sample profiler with a high-precision is difficult to achieve and prone to error [MSHD08, MDHS10, Ber11].

5.2 Support in the Virtual Machine

The Java Virtual Machine Tool Interface (JVM TI) is a native programming interface offered by the JVM to build dedicated profiling and debugging tools [Ora02]. JVM TI provides both a way to inspect the state and to control the execution of Java applications.

A JVM TI client, defined as an *agent*, can be notified of particular events emitted by the JVM. In total, a client may receive 31 different kinds of JVM events. These events cover breakpoints, class loading, garbage collection, method execution, monitor, and virtual machine start up and destruction.

JVisualVM [Ora08] is a visual profiling tool and framework. JVisualVM offers a large set of features, including

Existing profiler report	Accurate profiler report
/media/sophie/Data/GSOC/Part2-Precision/VM_Test/pharo-vm/lib/pharo/ 5.0-201706131152/pharo 2017-06-19 22:21:34 eden size: 3,801,936 stack pages: 50 code size: 1,048,576 7.126 seconds; sampling frequency 1450 hz 10305 samples in the VM (10332 samples in the entire program) 99.74% of total 10007 samples in generated vm code 97.11% of entire vm (96.85% of total) 298 samples in vanilla vm code 2.89% of entire vm (2.88% of total) % of generated vm code (% of total) (samples) (cumulative) **46.55% (45.08%) Integer>>benchFib (4658) (46.55%)** **20.40% (19.75%) Integer>>benchmark (2041) (66.94%)** 17.21% (16.67%) Object>>at:put: (1722) (84.15%) 10.10% (9.79%) SmallInteger>>+ (1011) (94.25%) 5.68% (5.50%) Object>>at: (568) (99.93%) 0.03% (0.03%) Sequenceab...m:to:put:(3) (99.96%) 0.02% (0.02%) Array>>repla...startingAt:(2) (99.98%) 0.02% (0.02%) ...others... (2) (100.0%) % of vanilla vm code (% of total) (samples) (cumulative) 47.65% (1.37%) primitiveStringReplace (142) (47.65%) 27.18% (0.78%) instantiateClassindexableSize (81) (74.83%) 8.72% (0.25%) scavengeReferentsOf (26) (83.56%) 5.70% (0.16%) copyAndForward (17) (89.26%) 3.36% (0.10%) doScavenge (10) (92.62%) 2.01% (0.06%) addressAfter (6) (94.63%) 1.68% (0.05%) heartbeat_handler (5) (96.31%) 1.34% (0.04%) bytesInObject (4) (97.65%) 0.67% (0.02%) shouldRemapObj (2) (98.32%) 1.68% (0.05%) ...others... (5) (100.0%) **Memory** old +157,616 bytes free +0 bytes **GCs** full 0 totalling 0ms (0.0% elapsed time) scavenges 182 totalling 57ms (0.8% elapsed time), avg 0.313ms tenures 0 root table 0 overflows **Compiled Code Compactions** 0 totalling 0ms (0.0% elapsed time) **Events** Process switches 38 (5 per second) ioProcessEvents calls 350 (49 per second) Interrupt checks 3745 (526 per second) Event checks 3743 (525 per second) Stack overflows 209 (29 per second) Stack page divorces 0 (0 per second)	/media/sophie/Data/GSOC/Part2-Precision/VM_Test/pharo-vm/lib/pharo/ 5.0-201706131152/pharo 2017-06-19 22:16:11 eden size: 3,801,936 stack pages: 50 code size: 1,048,576 6.948 seconds; sampling frequency 1495 hz 10368 samples in the VM (10389 samples in the entire program) 99.80% of total 10067 samples in generated vm code 97.10% of entire vm (96.90% of total) 301 samples in vanilla vm code 2.90% of entire vm (2.90% of total) % of generated vm code (% of total) (samples) (cumulative) **47.03% (45.58%) Integer>>benchFib (4735) (47.03%)** **57.95% 1->23 (2744) (57.95%)** 5.89% 24->30 (279) (63.84%) 2.89% 30->31 (137) (66.74%) 8.24% 31->34 (390) (74.97%) 3.38% 34->35 (160) (78.35%) 6.59% 35->36 (312) (84.94%) 3.06% 36->38 (145) (88.00%) 12.00% 38->6667 (568) (100.0%) **20.62% (19.98%) Integer>>benchmark (2076) (67.66%)** 0.05% 44->50 (1) (0.05%) 11.32% 52->60 (235) (11.37%) 3.90% 61->65 (81) (15.27%) 4.05% 65->66 (84) (19.32%) 6.17% 66->70 (128) (25.48%) 2.94% 70->74 (61) (28.42%) **35.36% 74->78 (734) (63.78%)** 8.24% 79->84 (171) (72.01%) 8.62% 84->88 (179) (80.64%) 8.67% 88->90 (180) (89.31%) 2.12% 90->94 (44) (91.43%) 5.15% 94->98 (107) (96.58%) 3.37% 98->100 (70) (99.95%) 0.05% 100->104 (1) (100.0%) 15.36% (14.88%) Object>>at:put: (1546) (83.01%) 100.0% 1->65 (1546) (100.0%) 10.78% (10.44%) SmallInteger>>+ (1085) (93.79%) 100.0% 1->22 (1085) (100.0%) 6.07% (5.88%) Object>>at: (611) (99.86%) 100.0% 1->57 (611) (100.0%) 0.04% (0.04%) Sequenceab...m:to:put:(4) (99.90%) 0.03% (0.03%) Array class>>new: (3) (99.93%) 0.02% (0.02%) Array>>repla...startingAt:(2) (99.95%) 0.02% (0.02%) SmallInteger>>- (2) (99.97%) 0.03% (0.03%) ...others... (3) (100.0%) % of vanilla vm code (% of total) (samples) (cumulative) 45.18% (1.31%) primitiveStringReplace (136) (45.18%) 30.56% (0.89%) instantiateClassindexableSize (92) (75.75%) 9.30% (0.27%) scavengeReferentsOf (28) (85.05%) 3.99% (0.12%) copyAndForward (12) (89.04%) 1.99% (0.06%) addressAfter (6) (91.03%) 1.99% (0.06%) doScavenge (6) (93.02%) 1.99% (0.06%) heartbeat_handler (6) (95.02%) 1.99% (0.06%) moveFramesInth...toPage.isra.74(6) (97.01%) 0.66% (0.02%) handleStackOverflow (2) (97.67%) 2.33% (0.07%) ...others... (7) (100.0%) **Memory** old +2,425,712 bytes free +0 bytes **GCs** full 0 totalling 0ms (0.0% elapsed time) scavenges 182 totalling 53ms (0.763% elapsed time), avg 0.291ms tenures 0 root table 0 overflows **Compiled Code Compactions** 0 totalling 0ms (0.0% elapsed time) **Events** Process switches 38 (5 per second) ioProcessEvents calls 340 (49 per second) Interrupt checks 3656 (526 per second) Event checks 3660 (527 per second) Stack overflows 15183 (2185 per second) Stack page divorces 0 (0 per second)

Figure 7: Comparison of profiling reports for `10 tinyBenchmark`

remote debugging / profiling, thread analysis, heap snapshot, and garbage collection monitoring.

5.3 Generic profilers

The profiling tool described in this paper is intended to be used to address performance issues. The software engineering community has produced software execution monitoring techniques to profile various aspects related to an execution [RBN12, RBNR12].

A common profiling technique is to use instrumentation instead of sampling [MLG05]. For example, Spy [BBRR11] is a framework to build domain-specific profilers, with application ranging from memory consumption analysis [IB15] to test coverage [BP12].

6. Conclusion and Future Work

In this paper, we have presented the evolutions carried out in VMProfiler, a profiler enabling to identify where the execution time is spent in the VM side, i.e. identify where the time is spent in the C code of the VM (interpreter, garbage collector) and in the jitted functions.

As this kind of tool is typically used to know where to boost performance, the existing VMProfiler could be improved: it could not provide detailed profiling data for large native code functions. Indeed, it could report that most of the execution time was spent in a function, but not *where* in this function the time was spent. This problem was getting significant as more and more optimisations were performed by the JIT ; inlining, especially, makes the jitted functions harder to accurately profile.

This paper describes a way to address this problem: an API used for debugging purposes offers a mapping between machine code pc and bytecode pc. This mapping is then used to determine bytecode ranges in a large native code function, and thus identifies how many samples are included in one or the other range. Now, VMProfiler provides detailed profiling statistical reports.

Further improvements are currently being considered:

- As for now, VMProfiler is only available headless in Pharo. A graphical user interface could be implemented to provide profiling data from another perspective.
- Sometimes, customers in a Windows environment request profiling, yet the VMProfiler is currently available for Mac and Linux only. The VMProfiler could then be implemented for Windows to tackle this problem.
- Currently, VMProfiler shows PIC disregarding if the PIC is a closed PIC or an open PIC. It would be nice to extend it to show this information (it requires changes in `primitiveCollectCogCodeConstituents`).

Acknowledgments

We thank Eliot Miranda for the original implementation of VMProfiler and his support during the evolution.

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

References

[BBRR11] Alexandre Bergel, Felipe Bañados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. *Journal of Computer Languages, Systems and Structures*, 38(1), December 2011.

[BCDL13] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

[BDN+07] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Squeak by Example*. Square Bracket Associates, 2007.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[Ber11] 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.

[Bér16] Clément Béra. Smalltalk, Tips 'n Tricks. CogMethod's Maps, 2016. https://clementbera.wordpress.com/2016/09/19/cogmethods-maps/.

[BM14] Clément Béra and Eliot Miranda. A bytecode set for adaptive optimizations. In *International Workshop on Smalltalk Technologies 2014*, IWST '14, 2014.

[BP12] Alexandre Bergel and Vanessa Peña. Increasing test coverage with hapao. *Science of Computer Programming*, 79(1):86–100, 2012.

[GB04] Isaac Gouy and Fulgham Brent. The Computer Language Benchmarks Game, 2004. http://benchmarksgame.alioth.debian.org/.

[Goo08] Google. V8 repository, 2008. https://github.com/v8/v8.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *European Conference on Object-Oriented Programming*, ECOOP '91, London, UK, UK, 1991.

[HU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Programming Language Design and Implementation*, PLDI '94, pages 326–336, New York, NY, USA, 1994.

[IB15] Alejandro Infante and Alexandre Bergel. Efficiently identifying object production sites. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (ERA Track)*, March 2015.

[MDHS10] 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.

[Mir08a] Eliot Miranda. Cog Blog: Speeding Up Terf, Squeak, Pharo and Croquet with a fast open-source Smalltalk VM, 2008. http://www.mirandabanda.org/cogblog/.

[Mir08b] Eliot Miranda. Cog Blog: Speeding Up Terf, Squeak, Pharo and Croquet with a fast open-source Smalltalk VM. The Idee Fixe and the Perfected Profiler, 2008. http://www.mirandabanda.org/cogblog/2008/12/30/the-idee-fixe-and-the-perfected-profiler/.

[MLG05] Edu Metz, Raimondas Lencevicius, and Teofilo F. Gonzalez. Performance data collection using a hybrid approach. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 126–135, New York, NY, USA, 2005. ACM.

[MSHD08] Todd Mytkowicz, Peter F. Sweeney, Matthias Hauswirth, and Amer Diwan. Observer effect and measurement bias in performance analysis, 2008.

[Ora02] Oracle. Java Virtual Machine Tool Interface, 2002. https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html.

[Ora08] Oracle. Java VisualVM, 2008. https://visualvm.github.io/.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java hotspotTM Server Compiler. In *Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01. USENIX Association, 2001.

[RBN12] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceedings of the 34rd international conference on Software engineering*, ICSE '12, 2012.

[RBNR12] Jorge Ressia, Alexandre Bergel, Oscar Nierstrasz, and Lukas Renggli. Modeling domain-specific profilers. *Journal of Object Technology*, 11(1):1–21, April 2012.