Profiling Code Cache Behaviour via Events

Pablo Tesone
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, Pharo Consortium
Lille, France
pablo.tesone@inria.fr

Guillermo Polito
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France
guillermo.polito@univ-lille.fr

Stéphane Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France
stephane.ducasse@inria.fr

Abstract

Virtual machine performance tuning for a given application is an arduous and challenging task. For example, parametrizing the behaviour of the JIT compiler machine code caches affects the overall performance of applications while being rather obscure for final users not knowledgeable about VM internals. Moreover, VM components are often heavily coupled and changes in some parameters may affect several seemingly unrelated components and may have unclear performance impacts. Therefore, choosing the best parametrization requires to have precise information.

In this paper, we present Vicoca, a tool that allows VM users and developers to obtain detailed information about the behaviour of the code caches and their interactions with other virtual machine components. We present a complex optimization problem due to the heavy interaction of components in the Pharo VM, and we explain it using Vicoca. The information produced by the tool allows developers to produce an optimized configuration for the VM. Vicoca is based on event recording that are manipulated during off-line analysis. Vicoca not only allows us to understand this given problem, but it opens the door to future work such as automatic detection of application characteristics, identification of performance issues, and automatic hinting.

CCS Concepts: • **Software and its engineering → Runtime environments**.

Keywords: just-in-time compilers, virtual machines, profiling, performance tuning

1 Introduction

Virtual Machines use Just-in-time compilers to obtain a boost in performance compared to bytecode interpreter execution [3, 12, 16]. The generated machine code is stored in a *code cache* from where it can be re-executed many times [3, 16]. Such code caches contain not only machine code resulting from application compilations, but also all dynamically generated code (*e.g.,* VM routines, primitives/-native methods, polymorphic-in-line cache stubs).

The size of the cache is an important parameter of the VM execution and performance. The cache size limits the number of methods that are kept in the cache, meaning that VMs must adequately select what the optimal set of methods to compile to machine code is. Also, if the code cache does not have any more space for new compiled methods, some existing methods should be discarded, and the code cache should be compacted to allow new methods to be compiled [10, 19].

All the behaviour of the code cache is governed by a set of parameters: *e.g.,* cache size, initial set of VM routines and primitives, minimum size to recover after compaction, method retention policy, and maximum length of the methods to compile in the cache. In the scope of the paper, we name *specific configuration* a given set of parameter values. Due to their close interdependencies, we refer to them as a single entity.

In addition, the JIT compiler and its associated code cache interact with other components of the virtual machine such as the garbage collector. For example, modifications in the parameters of the code cache affect GC performance. If methods compiled to machine code reference objects and literals allocated in the object heap; the code cache must be traversed and eventually updated when a garbage collection triggers. The code cache size directly affects the execution time of the garbage collector.

Because of this close interaction, reaching an optimal performance tuning for a given application with a particular workload requires special attention. VM users need an extensive knowledge of the internals of the VM and how each VM component is related with each other. VM developers suffer the same complexity when trying to understand the impact of modifications in the components of the VM, and when measuring the performance of changes to them. Users and developers of the VM do not have detailed information to correctly understand the relationship and impact of their configurations or modifications. There is a need for detailed information about the different aspects of VM execution. Such detailed information can be overwhelming and tools should offer ways to extract adequate knowledge to guide application tuning.

After sketching the code zone architecture of the Pharo VM [28] (See Section 2), we present a detailed example showing the complex relationships between VM compoenent (Section 3). We plot and inspect the behaviour of the JIT and the management of its code cache for a given application and workload. Then we present Vicoca: a tool that allows developers and users to get insight about the behaviour of the VM and a given application (Sections 4 and 5). Vicoca is based in the recording of VM events for off-line analysis.

Vicoca provides a rich event model extensible supporting new analyses. Our approach adds a different point of view of a given execution complementing existing profiling tools. This solution does not replace existing tools but it provides users and developers with more information to take decisions. We implemented Vicoca on top of the Pharo Virtual Machine [9], a production level virtual machine used by the Pharo programming language, a continuation of the work performed in the OpenSmalltalk-VM (Section 6).

2 Code Cache In Pharo

The Pharo VM JIT compiler is a non-optimising method-based compiler that compiles as soon as possible with the objective of enter machine code fast and stay there as long as possible [28]. It stores the compiled machine code methods in a code cache. When a message is sent, the VM executes the method in the machine code if it is already compiled and it is located in the code cache.

To speed up the method look-up in the bytecode interpreter Pharo VM uses a method look-up cache [24]. This cache allows the VM to easily resolve recently used methods faster. If there is a cache miss, the full look-up is performed and the method is added to the cache. The method in the cache is a bytecode method, it should be compiled to machine code. Once there is a cache hit for a method, the VM compiles the method to machine code. So, the VM essentially compiles recently used methods. Moreover, the VM optimizes messages-sends using linked calls for monomorphic call-sites and polymorphic-inline-caches stubs (PIC) for polymorphic call-sites [20].

New methods are compiled if there is enough space in the code cache. Otherwise, if the occupation threshold has been reached, a code compaction is planned. In that case a compaction process executes during the next interruption point (*i.e.,* method activation, stack overflow or green thread switch), performing three phases: a link-reference-counting phase, a marking phase, and the final compaction.

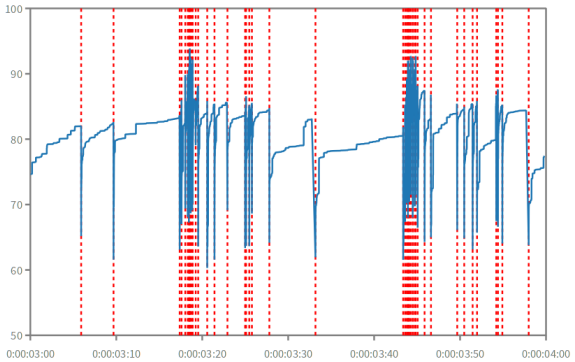
Link-reference-counting phase. It does a breadth-first traversal of the code cache methods starting from those present in the execution stack. When a method is traversed all linked call-sites are traversed to mark the target methods. If the call-site is monomorphic, the target method is traversed. If the call-site is polymorphic, the PIC stub is traversed. When a PIC is traversed, the referenced methods are traversed. Each time a method or stub is traversed a counter is updated in the method. This counter represents the number of incoming call-sites linked to the method.

Marking phase. Once the link-reference-counting phase finishes, a marking phase starts. It iteratively marks methods until a certain target amount of memory is reached. Starting with methods with counter zero the marking phase marks as to-be-released all methods with the given counter value. This means that methods that are not linked to any caller are the first candidates for removal, then the methods that are linked to a single caller, and so on.

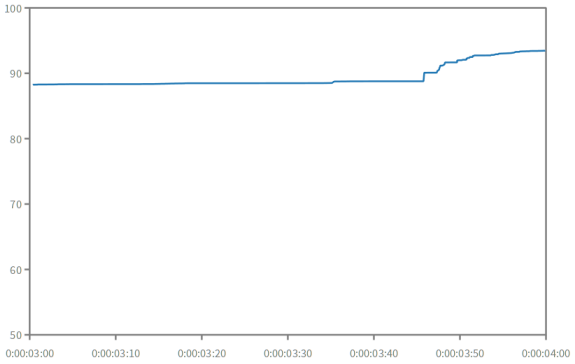
Compaction phase. Once the marking phase finishes, a compaction is performed, moving all the non-marked methods to lower addresses in the code cache. During this process all linked call-sites and method references from the object heap and the execution stack must be updated.

3 An Optimization Example

We present a common scenario where users want to tune Pharo VM behaviour. Our example is the typical loading of a large project. We took Moose a software analysis platform [1, 32]. The current compilation schema of Pharo requires that all code is loaded and compiled to bytecode. Loading Moose compiles 1,662 classes and 51,053 methods.



(a) With a cache size of 1.4MB.



(b) With a cache size of 10MB.

Figure 1. Code Cache occupation rate when the application is in steady-state. In blue, the occupation rate of the code cache. In red, the compaction events.

This example execution has two additional advantages that make it more representative: (1) it has a limited working set, as compiling a method uses all the time the same set of methods; and (2) the execution time is long enough to clearly present trashing patterns and discuss later performance improvements. The working set of this problem is 10,959 methods and closures (for practical purposes, the

Table 1. Code Cache Statistics for different Cache Sizes.

Cache Size	Working Set	Compactions	Compilations (Unneeded)	Compaction Time	Compilation Time
1.4 MB	10,959	780	583,787 (572,935)	3,825 ms	5,161 ms
2.8 MB	10,959	35	34,075 (23,116)	380 ms	601 ms
5 MB	10,959	5	11,790 (831)	43 ms	250 ms
10 MB	10,959	1	11,010 (51)	8 ms	252 ms

JIT compiler and the code cache compaction treat them in the same way). The analyses in this section are done using Vicoca and their building will be explained later.

Figure 1a presents a graph showing the occupation rate and the code cache compaction events during a minute of execution. This extraction of the execution is when the application is in steady-state and it is performing the compilation of classes and methods. This figure presents the number of code compactions performed during a sample of the execution and how the code cache occupation varies showing recompilation of some methods in the working set. The figure shows *code cache trashing*: If the machine code for the entire application working set does not fit in the code cache, the VM needs to compact numerous times the code cache. But if these methods are part of the working set, they will be recompiled shortly after.

Figure 1b shows the occupation rate and code cache compaction events using an increased code cache size. In this version, the code cache gets to a stable point where all the methods are there. The only code compaction is produced to discard the methods that are used during the application start-up. This set of methods is independent of the application as this start-up contains all methods executed during the standard Pharo base library initialization.

Table 1 presents a deeper analysis of this situation: there is a set of methods that are recompiled many times affecting the overall performance. Increasing the cache size minimizes the recompilation of methods and reduces the time spent in the JIT compiler. By looking only at these results, it seems that we may have an improvement in performance. However, next Section shows the contrary.

3.1 Method Cache Size / Garbage Collect Time

Figure 2 presents the result of testing the same application with three GC young space configurations (1MB, 10MB, and 100MB) and four code cache configurations (1.44MB, 2.88MB, 5MB, and 10MB). Again this analysis is made with Vicoca. This figure shows that increasing the code cache actually introduces a performance penalty. An increase of the *young space* size (one of the GC parameters) produces a performance improvement. However, by simply looking at the numbers it is not clear why increasing the code cache has a negative impact on the application performance, as we have seen that it minimizes the number of method recompilations.

Explanation. This application puts a strong pressure on the garbage collector, creating and discarding thousands of

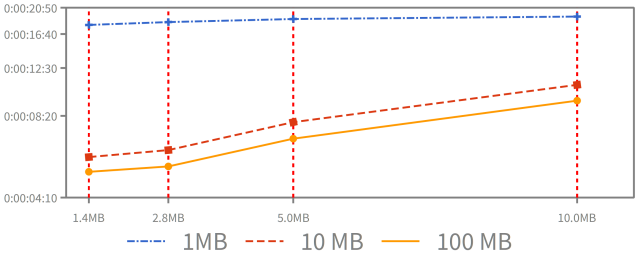


Figure 2. Execution time for different Young Space size (1MB, 10MB, 100MB) and Cache Sizes (1.44MB, 2.8MB, 5MB, 10MB)

objects per second. When a class or a method is compiled from source code, it requires generating multiples objects that are discarded once the compilation is performed (*e.g.*, AST nodes, intermediate representations nodes). As this application has an extensive use of memory, it produces numerous garbage collection (GC) passes. For a young space size of 1MB, it produces 28 full GCs; and for a young space of 100MB it is reduced to 6.

As the code cache has references to objects that may move during the GC execution, those references need to be updated. The update of the references is performed by traversing the code cache to look for and to update moved references. A larger code cache implies longer execution times during the GC. Moreover, in the machine code compiled methods, all references to objects are in-lined in machine instructions. This means that the GC requires to disassemble the instructions to check if there is a reference to a moved object, reassemble it and patch the existing code if the target object has been moved.

To improve the performance of the application we need to find a balance in the GC parameters (*e.g.*, object memory size, growth rate) and the code cache size. To correctly tune up the application, the user requires precise information clearly presenting the impact of a given configuration.

3.2 Raw Data Is Not Enough

Adequate VM performance tuning requires to have precise information describing the application execution as well as VM components. In our example, the required information includes JIT compiler events (method compilations, number of executions, PIC usage), garbage collector statistics (GC execution events, total execution time, average execution time), code cache management events (discarded methods/PICs, code cache compaction, compaction time), and general statistics (execution time). In addition, a tool should address the

three following requirements: (1) Correlating behaviour. All events are recorded independently, and they do not represent the relations between each other. (2) Usable. Raw events represent the execution, but they are not presented in a way the user can exploit them directly. (3) Scalable. The number of recorded events grows fast (in our example, the amount of events collected in each run ranges from 35,000 to 1,400,000 depending on the execution time and produced events).

4 Event Model and Collection

Vicoca processes a collection of events and generates several analyses from them.

4.1 Event Model

Vicoca loads the events in memory into a rich object-oriented model. This architecture allows us to implement different analysis algorithms on top of the same model incrementally. The output of these algorithms are new CSV files to be processed by other tools, interactive plots, and interactive inspectors to navigate the results and events. Figure 3 presents an overview of the architecture. It includes, the two main object classes used by the tool and users. The VMEventCollection is the entry point of any script or tool manipulating the events. Vicoca has a rich hierarchy of events that represent different operations and state.

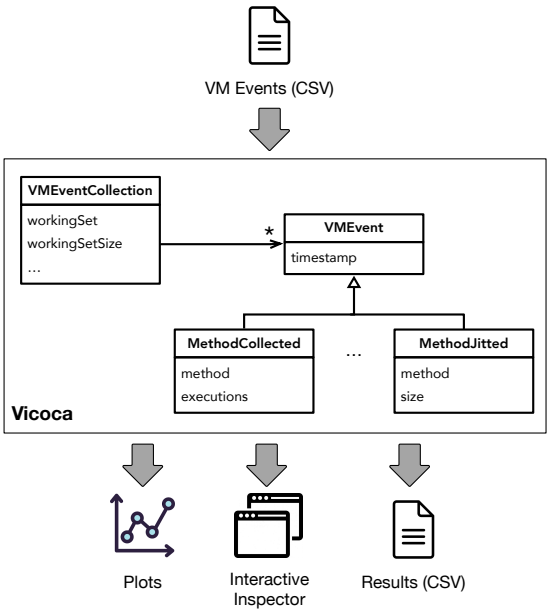


Figure 3. Vicoca's Overall Architecture showing input and existing output formats.

Moreover, by using the scripting and live programming tools of Pharo the tool exposes an extensible tool. Users are able to script the application, combine different algorithms to enriching the produced results and navigate them interactively to improve the user's ability to discover new relationships between events.

4.2 Event Generation

Vicoca is an event-based profiler [36] capturing all related VM events and storing them, to later reconstruct the execution history off-line.

We have modified the Pharo VM to record the events that are related to the JIT compiler, code cache management, and other VM components such as the Garbage Collector.

Also, we have added a primitive to the VM allowing the image to log other events that are important for the application, such as the end of the image start-up process.

Each event includes the parameters describing the event, for example, method compilation events record the method class and selector, and its size; method eviction events identify the method and the times it has been executed. Also, they include a timestamp so Vicoca reconstructs the execution history. Table 4.2 presents the complete details.

Each VM execution generates a set of collected events that are stored directly to the disk in a CSV file.

5 Getting Gold out of Raw Events

Once the events are loaded in memory, they are used through the interface exposed by the model. The main entry point is a VMEventCollection object. It represents the whole collection of events providing accessing messages.

5.1 Working Set Identification

To find the optimal size of the code cache (one allowing to have all the working set at once), we require to identify:

Working Set. It is the set of methods that is executed in the steady state of the application. Listing 1 shows the code to calculate it. It identifies the point the image has ended the start-up (event generated by the image), and uses it to get all methods after this event. Then, the found methods are sorted by number of executions. Presenting to the user the more popular methods.

Working Set Compiled Size. Once we have the correct set of methods executed by the application. We need to calculate the size of them. Listing 2 performs this calculation. It takes the working set, it calculates the max size of each method. As said before, code cache also includes the PICs used by the application, so they are included in the compiled methods.

Figure 4 shows the more popular methods in the execution of the application under tests as shown by the tool.

Table 2. Event Details: Detailed set of events collected from the application execution.

Event	VM Component	Description	Additional Information in the Event
GC Counters	*VM Shut-down*	When shut-down, the VM logs all GC counters and statistics (Used as validation)	total gc time, scavenger total time, scavenger runs, full gc total time, full gc runs
Shut-down	*VM Shut-down*	Last event produced by the VM, after this the VM process ends.	
Cog Counters	*VM Shut-down*	When shutdown, the VM logs all JIT counters and statistics (Used as validation)	total number of methods, total number of blocks, total time
Image Loaded	*VM Start-up*	This event is produced when the image is finally loaded, the image start running from here.	image name
Start-up Ended	*Image Start-up*	The image reports that the startup ends, the application start running.	
Scavenge GC	*Young Space GC*	A GC has run in the young objects space (Eden).	execution time, recovered space
Full GC	*Old Space GC*	A GC has run in the old objects space.	execution time, recovered space
Compact Code Cache	*Code Cache*	A compaction has been performed in the code cache after all methods have been collected.	execution time, recovered space
Method Collected	*Code Cache*	A method/block/PIC is removed from the code cache to free space.	method identifier, type, method executions
Method Compiled	*JIT Compiler*	A new method/block/PIC has been compiled to machine code.	method identifier, type, size
Ensure Method	*JIT Compiler*	VM infrastructure requires method in machine code, it might trigger a compilation (*e.g.*, debugging, stack mapping)	method identifier

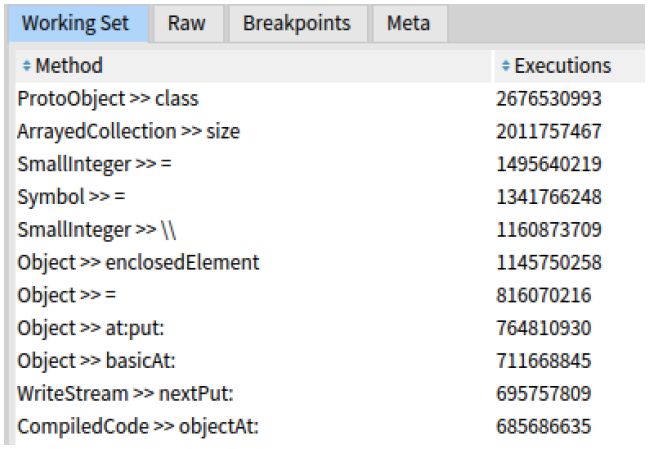
```
1  VMEventCollection >> workingSet
2    | endOfStartupEvent eventsAfterStartup
3      methods workingSet |
4    endOfStartupEvent := self endOfStartupEvent.
5    eventsAfterStartup := self eventsAfter: endOfStartupEvent.

6    methods := eventsAfterStartup select: [ :e | e
         isMethodCollected and: [ e isMethod ] ].
7    workingSet := (methods groupedBy: #fullMethodName)
8      associations
9        collect: [:aGroup | VMWorkingSetItem fromGroup:
         aGroup]
10       as: OrderedCollection.
11   ^ workingSet sorted: [ :a :b | a executions > b executions ].
```

Listing 1. Calculating Working Set

```
1  VMEventCollection >> workingSetSize
2    ^ self workingSet sumNumbers: [ :item | item
       lookupMethodSize: events ]
3
4  WorkingSetItem >> lookupMethodSize: events
5    ^ (events
6      select: [ :e | e isMethodJitted
7        and: [ e fullMethodName = methodName ] ])
8          max: #methodSize
```

Listing 2. Calculating Working Set Size



Working Set | Raw | Breakpoints | Meta

Method	Executions
ProtoObject >> class	2676530993
ArrayedCollection >> size	2011757467
SmallInteger >> =	1495640219
Symbol >> =	1341766248
SmallInteger >> \\	1160873709
Object >> enclosedElement	1145750258
Object >> =	816070216
Object >> at:put:	764810930
Object >> basicAt:	711668845
WriteStream >> nextPut:	695757809
CompiledCode >> objectAt:	685686635

Figure 4. Example of Working Set calculated by Vicoca

```
1  VMEventCollection >> cacheModifications
2    ^ (events select: [ :e | e isMethodJitted or: [e
       isMethodCollected])
3
4  VMEventCollection >> compactions
5    ^ (events select: [ :e | e isCompactionEnded ])
6
7  VMEventCollection >> fullGCs
8    ^ (events select: [ :e | e isFullGC ])
9
10 VMEventCollection >> scavengeGCs
11   ^ (events select: [ :e | e isScavengeGC ])
```

Listing 3. Selecting Events

5.2 Code Cache Trashing

Vicoca provides visualizations to detect code cache trashing. Figures 1b and 1a show the results of these visualizations. The generated information is used to show the evolution of the cache occupation and compactions during the execution.

Listing 4 calculates the cache occupation rate in each compilation of a method. Informing the evolution of the code cache. It extracts all events that are cache modifications (compiling a method, or removing a method). Then, it uses these methods to calculate the number of methods in the code cache. It keeps the state of the code cache in each event. Later, it produces objects that are represents this information. Listing 3 shows different filters on the events, as used by the code cache occupation.

```
1  VMEventCollection >> occupationRates
2    | total methods maxQty firstTimestamp |
3    total := 0.
4    methods := self cacheModifications.
5
6    "Update the number of elements in the code cache"
7    methods do: [ :e |
8      e isMethodJitted
9        ifTrue: [ total := total + 1 ]
10       ifFalse: [ total := total − 1 ].
11     e totalMethods: total ].
12
13   "Compute the max number of elements to estimate
       occupation %"
14   maxQty := methods max: [ :e | e total ].
15   firstTimestamp := methods first timestamp.
16
17   ^ methods collect: [ :e |
18     VMOccupationItem new
19       relativeTime: e relativeTime;
20       numberOfMethods: e total;
21       percentage: e total / maxQty;
22       yourself ]
```

Listing 4. Computing Code Cache Occupation

Listing 5 shows how the plots in Figures 1b and 1a are generated. It uses the generated data and Roassal (Pharo's plotting library).

5.3 Execution and Garbage Collector Statistics

Vicoca calculates extensive statistics about the execution of the Garbage Collector. Listing 6 presents some examples of generated statistics from the events collected. These statistics presents general results about the execution of the GC, and the whole application. All garbage collector statistics are presented for both generations of the garbage collector, as shown in the selected ones.

Vicoca provides flexibility to plot all information produced in one or many runs of the application. Listing 7 shows code to produce a plot that combines the 4 different executions of the application. Providing in such way a common point of comparison. A similar but more complex script has been used to produce Figure 2.

6 Implementation Decisions

We implemented Vicoca using an event-based approach because it produces a precise stream of events. While this technique affects the performance of the VM execution, the precision of the information is not affected by this performance impact. The VM performs exactly the same number of operations and events as if it is not logging; only event timestamps are affected. This technique allows the detection of existing relations between the events and complex interactions [8, 36, 42]. Sampling techniques [7, 30, 41] could minimize the performance impact at the cost of precision,

since they are only able to detect events that are less frequent that the sampling rate [7, 35, 46]. Moreover, our modified version of the VM is intended to be run only for profiling purposes and it is not intended to be used in productive environments.

We modified the execution runtime to generate the events instead of automatically instrumenting the application's code [4, 18, 23, 26]. This decision allows us to have a VM specific profiler that gives us low-level VM data that is not available at the application level. For example, from the application point of view there is not a hook to log when a method is compiled to code machine.

Even thought the application working set and its size might be computed ahead of time by approximation using an execution model, we believe the complexity of those required models makes our dynamic recording cheaper to implement. A static analysis requires to have knowledge not only of the execution characteristics of the application but specifics about the architecture and existing optimizations in the VM. Moreover, computing the size of methods and their corresponding PICs requires to exactly know the JIT compiler implementation. This requirement gets more and more complex with optimizations such as in-lined and PICs. Instead, our solution reuses the existing VM which already provides an efficient execution engine and the effort of implementing our instrumentation is neglectable.

```
VMEventCollection >> occupationRatePlot
  occupationRates := self occupationRates.
  chart := RSChart new.
  x := occupationRates collect: #relativeTime.
  y := occupationRates collect: #percentage.
  plot := RSLinePlot new x: x y: y.
  chart addPlot: plot.
  compactions do: [ :e |
    chart addDecoration: (RSXMarkerDecoration new
        value: e relativeTime;
        yourself) ].
  ^ chart build; canvas
```

Listing 5. Plotting Code Cache Occupation

```
VMEventCollection >> fullGCAverageRecolection
  gcs := self fullGCs.
  ^ (gcs sumNumbers: #reclamedMemory) / gcs size.

VMEventCollection >> fullGCTime
  ^ self fullGCs sumNumbers: #executionTime

VMEventCollection >> executionTime
  ^ events last timestamp − self startTime
```

Listing 6. Some GC and Execution Statistics

```
executions1M := #('1400K−1M.csv'
'2800K−1M.csv' '5000K−1M.csv' '10000K−1M.csv')
  collect: [:e |
    VMEventCollection
      readFromFileReference: e asFileReference].
plot1 := RSLinePlot new
  x: #(1.4 2.8 5.0 10.0)
  y: (executions1M collect:#executionTime);
  fmt: '+−.'.
chart addPlot: plot1.
chart yLog; build; canvas
```

Listing 7. Plotting Execution Times

We implemented our analysis tool in Pharo to take advantage of all its live-programming features and utilities to inspect live objects. Also, it includes tools for data analysis and plotting that we profit [2, 11, 34]. All results and plots of this paper has been generated using Pharo.

7 Related Work

Kaleba *et al.,* [22] have extended an existing profiling tool to collect and expose VM information. Different from ours, their solution is intended to detect performance issues in the compiled machine code and is not aware of the impact of the GC in the code cache, the code compaction or the time spent by the JIT compiler.

Indicium [44] is a profiling tool for the V8 Virtual Machine. This profiling tool allows the developer to see the interaction of the application with the generated machine code and PICs. However, this tool does not produce information about memory consumption, garbage collection, or code cache statistics. This tool is intended to improve the performance speed of the running application.

Other profiling solutions exists in the literature, although they center the analysis in improving the speed of the running application. So, they provide little to none information about the interactions between the VM components [21, 25, 33, 38].

To overcome some limitations of the event-based approach, some techniques use hardware specific features to measure. They use information available in the architecture such as execution counters, cache statistics, and memory manager events [13–15, 29, 31]. These same techniques have been extrapolated to get information from Virtual Machine executions [27, 43, 45].

Finally, there is a large body of work on the data analysis of profiling information. Bergel proposed to use profilers to identify where (application) caches get an impact [5]. Levin *et al.,* proposed a technique to improve the precision of sampling techniques. Sandoval [39, 40] worked on identifying speed regression between consecutive versions. Bertuli *et al.,* used metrics to quantify the mass of (often repetitive) generated information [6, 17, 37].

8 Future Work

Vicoca presents different points of extension to be done as future work. We identify three categories of possible future work. All of them are oriented to simplifying the VM tuning task.

Application Behaviour Recognition. By using the collected information is possible to recognize characteristics of the running application and create application profiles. The VM user would then be able a profile based on its application's behaviour. For example, by detecting if the application is memory intensive, larger heap configurations could be used. Also, this information is an excellent entry point to generate application benchmarks that correctly represent the behaviour of real world applications.

Automatic Performance Bottleneck Detections. Detecting performance bottlenecks requires to understand the inner working of the VM and their components. This knowledge is not common between all users. However, if such knowledge is expressed in terms of algorithms using the collected events as inputs, an extension of Vicoca would be able to provide optimizations hints for the configuration or the development of the application.

Automatic Performance Hinting. If the execution used to collect the events is a productive execution, the information collected provides a source to automatically extract performance hinting. An example of this is the detection of the working set of an application. If this information is then later provided to the JIT compiler, it would be able to perform different optimizations such as ahead of time compilation or avoiding discarding them during compaction.

9 Conclusion

In this paper, we presented a non-trivial configuration problem of a production level Virtual Machine. To correctly address this configuration problem, VM users need to have precise information that shows the existing complex relationships between VM components.

To give support to VM users, we present a first iteration of Vicoca; a tool that allows users to record events produced by the VM machine and to analyse these events to present useful information to the user.

We show that even in this initial state our tool helps to understand the described problem. Also, we present possible extensions to this work based in the information captured by Vicoca.

References

[1] Nicolas Anquetil, Anne Etien, Mahugnon Honoré Houekpetodji, Benoît Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatija Djareddir, Jèrome Sudich, and Mustapha Derras. 2020. Modular Moose: A new generation of software reengineering platform. In *International Conference on Software and Systems Reuse (ICSR'20) (LNCS)*. https://doi.org/10.1007/978-3-030-64694-3_8

[2] Vanessa Peña Araya, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. 2013. Agile Visualization with Roassal. In *Deep Into Pharo*. Square Bracket Associates, 209–239.

[3] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Laguë, and Kostas Kontogiannis. 2000. Advanced Clone-Analysis to Support Object-Oriented System Refactoring. In *Proceedings Seventh Working Conference on Reverse Engineering (WCRE'00)*, Françoise Balmas and Kostas Kontogiannis (Eds.). IEEE Computer Society, 98–107. http://nms.lcs.mit.edu/~mbalazin/publications/wcre2000Balazinska.ps

[4] Thomas Ball and James R. Larus. 1994. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems* 16, 3 (1994), 59–70. https://doi.org/10.1145/183432.183527

[5] Alexandre Bergel, Romain Robbes, and Walter Binder. 2010. Visualizing Dynamic Metrics with Profiling Blueprints. In *Objects, Models, Components, Patterns (TOOLS EUROPE'10) (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 6141. Springer Berlin / Heidelberg, 291–309. https://doi.org/10.1007/978-3-642-13953-6_16

[6] Roland Bertuli, Stéphane Ducasse, and Michele Lanza. 2003. Run-Time Information Visualization for Understanding Object-Oriented Systems. In *Proceedings of 4th International Workshop on Object-Oriented Reengineering (WOOR'03)*. University of Antwerp, 10–19.

[7] Walter Binder. 2006. Portable and accurate sampling profiling for Java. *Software: Practice and Experience* 36, 6 (2006), 615–650. https://doi.org/10.1002/spe.712 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.712

[8] Walter Binder, Jarle Hulaas, Philippe Moret, and Alex Villazón. 2009. Platform-Independent Profiling in a Virtual Execution Environment. *Softw. Pract. Exper.* 39, 1 (Jan. 2009), 47–79. https://doi.org/10.1002/spe.890

[9] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland. 333 pages. http://books.pharo.org

[10] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. 2006. Thread-shared software code caches. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 11–pp. https://doi.org/10.1109/CGO.2006.36

[11] Damien Cassou, Stéphane Ducasse, Luc Fabresse, Johan Fabry, and Sven Van Caekenberghe. 2015. *Enterprise Pharo: a Web Perspective*. Square Bracket Associates. 278 pages. http://books.pharo.org

[12] Craig Chambers, David Ungar, and Elgin Lee. 1989. An Efficient Implementation of SELF — a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, Vol. 24. 49–70. https://doi.org/10.1145/74878.74884

[13] Hyoun Kyu Cho, Tipp Moseley, Richard Hank, Derek Bruening, and Scott Mahlke. 2013. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–10. https://doi.org/10.1109/CGO.2013.6494982

[14] Thomas Conte, Kishore Menezes, Burzin Patel, and J. Cox. 1996. Hardware-Based Profiling: An Effective Technique for Profile-Driven Optimization. *International Journal of Parallel Programming* 24 (apr 1996). https://doi.org/10.1007/BF03356747

[15] J. Dean, J.E. Hicks, C.A. Waldspurger, W.E. Weihl, and G. Chrysos. 1997. ProfileMe: hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. 292–302. https://doi.org/10.1109/MICRO.1997.645821

[16] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 system. In *Proceedings POPL '84*. Salt Lake City, Utah. https://doi.org/10.1145/800017.800542

[17] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. 2004. High-Level Polymetric Views of Condensed Run-Time Information. In *Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*. IEEE Computer Society Press, Los Alamitos CA, 309–318. https://doi.org/10.1109/CSMR.2004.1281433

[18] Michael Factor, Assaf Schuster, and Konstantin Shagin. 2004. Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '04)*. ACM, New York, NY, USA, 288–300. https://doi.org/10.1145/1028976.1029000

[19] Kim Hazelwood and James E Smith. 2004. Exploring code cache eviction granularities in dynamic optimization systems. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 89–99. https://doi.org/10.1109/CGO.2004.1281666

[20] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *European Conference on Object-Oriented Programming (ECOOP '91)*. https://doi.org/10.1007/BFb0057013

[21] java [n.d.]. VisualVM: All-in-One Java Troubleshooting Tool. https://visualvm.github.io/. https://visualvm.github.io/

[22] Sophie Kaleba, Clément Béra, Alexandre Bergel, and Stéphane Ducasse. 2017. A detailed VM profiler for the Cog VM. In *International Workshop on Smalltalk Technology IWST'17*. Maribor, Slovenia. https://hal.inria.fr/hal-01585754

[23] Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. 2008. JavaScript Instrumentation in Practice. In *APLAS 2008*. https://doi.org/10.1007/978-3-540-89330-1_23

[24] G. Krasner. 1983. *Smalltalk-80: Bits of History, Words of Advice*. Addison Wesley, Reading, Mass.

[25] Lukáš Marek, Stephen Kell, Yudi Zheng, Lubomír Bulej, Walter Binder, Petr Tůma, Danilo Ansaloni, Aibek Sarimbekov, and Andreas Sewe. 2013. ShadowVM: Robust and comprehensive dynamic program analysis for the Java platform. *ACM SIGPLAN Notices* 49, 3 (2013), 105–114. https://doi.org/10.1145/2637365.2517219

[26] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD '12)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2162049.2162077

[27] Nevena Milojković, Clément Béra, Mohammad Ghafari, and Oscar Nierstrasz. 2016. Inferring Types by Mining Class Usage Frequency from Inline Caches. In *International Workshop on Smalltalk Technologies IWST'16*. Prague, Czech Republic. https://doi.org/10.1145/2991041.2991047

[28] Eliot Miranda. 2011. The Cog Smalltalk Virtual Machine. In *Proceedings of VMIL 2011*.

[29] Shirley V. Moore. 2002. A Comparison of Counting and Sampling Modes of Using Performance Monitoring Hardware. In *Computational Science — ICCS 2002*, Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 904–912. https://doi.org/10.1007/3-540-46080-2_95

[30] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. 2007. Shadow Profiling: Hiding Instrumentation Costs with Parallelism. In *International Symposium on Code Generation and Optimization (CGO'07)*. 198–208. https://doi.org/10.1109/CGO.2007.35

[31] H. Mousa and C. Krintz. 2005. HPS: hybrid profiling support. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 38–47. https://doi.org/10.1109/PACT.2005.24

[32] Oscar Nierstrasz and Stéphane Ducasse. 2004. Moose–a Language-Independent Reengineering Environment. *European Research Consortium for Informatics and Mathematics (ERCIM) News* 58 (July 2004), 24–25. http://www.ercim.org/publication/Ercim_News/enw58/nierstrasz.html

[33] Xuesong Peng, Barbara Pernici, and Monica Vitali. 2018. Virtual Machine Profiling for Analyzing Resource Usage of Applications. In *Services Computing – SCC 2018*, João Eduardo Ferreira, George Spanoudakis, Yutao Ma, and Liang-Jie Zhang (Eds.). Springer International Publishing, Cham, 103–118. https://doi.org/10.1007/978-3-319-94376-3_7

[34] polymath [n.d.]. PolyMath: Scientific Computing with Pharo. https://github.com/PolyMathOrg/PolyMath. https://github.com/PolyMathOrg/PolyMath

[35] Carl Ponder and Richard J. Fateman. 1988. Inaccuracies in program profilers. *Software: Practice and Experience* 18, 5 (1988), 459–467. https://doi.org/10.1002/spe.4380180506 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380180506

[36] S.P. Reiss. 2003. Event-based performance analysis. In *11th IEEE International Workshop on Program Comprehension, 2003.* 74–83. https://doi.org/10.1109/WPC.2003.1199191

[37] Jorge Ressia, Alexandre Bergel, Oscar Nierstrasz, and Lukas Renggli. 2012. Modeling Domain-Specific Profilers. *Journal of Object Technology* 11, 1 (April 2012), 1–21. https://doi.org/10.5381/jot.2012.11.1.a5

[38] Eduardo Rosales, Andrea Rosà, and Walter Binder. 2020. FJProf: Profiling Fork/Join Applications on the Java Virtual Machine. In *Proceedings of the 13th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS '20)*. Association for Computing Machinery, New York, NY, USA, 128–135. https://doi.org/10.1145/3388831.3388851

[39] Juan Pablo Sandoval Alcocer, Fabian Beck, and Alexandre Bergel. 2019. Performance Evolution Matrix: Visualizing Performance Variations Along Software Versions. In *2019 Working Conference on Software Visualization (VISSOFT)*. 1–11. https://doi.org/10.1109/VISSOFT.2019.00009

[40] Juan Pablo Sandoval Alcocer, Alexandre Bergel, Stéphane Ducasse, and Marcus Denker. 2013. Performance Evolution Blueprint: Understanding the Impact of Software Evolution on Performance. In *Vissoft 2013*. https://doi.org/10.1109/VISSOFT.2013.6650523

[41] S. Subramanya Sastry, Rastislav Bodík, and James E. Smith. 2001. Rapid Profiling via Stratified Sampling. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*. Association for Computing Machinery, New York, NY, USA, 278–289. https://doi.org/10.1145/379240.379273

[42] Gülfem Savrun-Yeniçeri, Michael L. Van de Vanter, Per Larsen, Stefan Brunthaler, and Michael Franz. 2015. An Efficient and Generic Event-Based Profiler Framework for Dynamic Languages. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. Association for Computing Machinery, New York, NY, USA, 102–112. https://doi.org/10.1145/2807426.2807435

[43] Peter F Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. 2004. Using Hardware Performance Monitors to Understand the Behavior of Java Applications.. In *Virtual Machine Research and Technology Symposium*. 57–72.

[44] V8 [n.d.]. Indicium: V8 runtime tracer tool. https://v8.dev/blog/system-analyzer. https://v8.dev/blog/system-analyzer

[45] Hernán Wilkinson. 2019. VM Support for Live Typing: Automatic Type Annotation for Dynamically Typed Languages. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming (Programming '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 3 pages. https://doi.org/10.1145/3328433.3328443

[46] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. 2019. Can We Trust Profiling Results? Understanding and Fixing the Inaccuracy in Modern Profilers. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*. Association for Computing Machinery, New York, NY, USA, 284–295. https://doi.org/10.1145/3330345.3330371