# Memory Profiling Blueprint

Alexandre Bergel[1], Mariano Abel Coca[2], Gabriela Arevalo[3], Dale Henrichs[4], Jannik Laval[5]

[1] Pleiad Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile
[2] Universidad Tecnológica Nacional, Buenos Aires, Argentina
[3] Universidad Nacional de Quilmes, Buenos Aires, Argentina
[4] Gemstone, USA
[5] INRIA Lille Nord Europe, France

www.bergel.eu
gabriela.b.arevalo@gmail.com
dhenrich@vmware.com
jannik.laval@inria.fr

**Abstract**

Keeping track of the memory consumption along the execution of an application is a difficult and delicate operation. In this paper, we present an innovative visual representation of the memory consumption and the execution time at the method granularity. Our visualizations were effective to identify an important resource consumption issue in Metacello, a widely used package mechanism for Pharo. We were able to easily fix this issue. Our visualization are implemented in Memory Profiler.

*Keywords:* Smalltalk, Pharo, Mondrian, memory profiling

## 1. Introduction

Nowadays, keeping track of the memory consumption in systems is not a trivial task. Considering object-oriented ones, there are essentially two reasons for this problem. Firstly, code execution profiling tools found in common platforms usually focus on the method execution time and extracting the method call graph. Memory consumption is not a main analysis issue. Specifically in Smalltalk platforms, reflective methods such as Behavior>> allInstances are commonly called to the rescue. However, the number of instances of a given class is rather a crude information which is difficult to exploit. Tools, such as Spac-

eTally[1] in Pharo helps getting information about system space usage. However, it is rather slow to operate since the whole memory is scanned for each class under analysis. Reflection and SpaceTally have a global scope, meaning that no distinction is made between an object created by the programming environment or the application under investigation. Secondly, it is difficult to identify the actions required to reduce the memory usage. Reducing the object creation frequency is not always that easy.

The need to reduce the memory consumption is usually perceived by a software developer only when the usage of the application is compromised with the slow response when, for example, the application execution is overwhelmed with garbage collector activations. Thus, we have designed a general approach to keep track of memory consumption.

In this paper, we describe a simple but effective profiler that keeps track of memory consumption at the method granularity level. The profiling operates on any arbitrary Smalltalk expression. Profiling information is graphically rendered to easily spot methods responsible for larger memory consumption.

The application to profile is first instrumented to capture runtime execution. For each method execution, the amount of memory before and after the execution is computed. The memory consumption of each method is computed along the application execution. After the application execution, the collected information is easily rendered using the Mondrian visualization engine[2].

Our profiler has been validated on Metacello[3]. We identified a number of issues related to memory consumption. Several methods were responsible for high memory consumption. We significantly reduced the memory needed by Metacello by introducing a number of memory caches.

An implementation is freely available[4] for the Pharo Smalltalk under the MIT license.

## 2. Memory Profiling

### *2.1. Memory profiling in a nutshell*

As most code execution profiler, our memory profiler works with four sequential steps:

1. *Instrumentation of the application to profile* – A meta-object is associated to each method of the application to profile. This meta-object is responsible to keep track of its memory consumption. The meta-object intercepts a method execution and computes the consumed memory by subtracting the amount of free memory after and before the method execution.

[1] SpaceTally is a tool implemented as a unique class, available in every distribution of Pharo.

[2] http://moosetechnology.org/tools/mondrian

[3] http://code.google.com/p/metacello

[4] http://www.squeaksource.com/Spy.html

2. *Application execution* – In addition to keeping track the memory consumption, identifying methods that are constant on their return value gives good hints on where to insert memory cache.

3. *Uninstalling the profiling instrumentation* – Once the application execution has finished, the profiled application is de-instrumented.

4. *Visualization of the profiling result* – The information collected is rendered via a polymetric view [1] in Mondrian[5]. The metrics we are focussing on are memory usage and execution time for each method.

Our profiler runs on any standard Pharo virtual machine.

### *2.2. Memory blueprints*

The information obtained during the application execution is suitably exploited using two visualizations:

- *Structural Distribution Blueprint* represents the distribution of the memory consumption along the application structure, expressed in terms of classes and methods;

- *Behavioral Distribution Blueprint* depicts the distribution along the method call graph of the application.

*Metacello.* The experiment described in this paper is driven by the need for optimization we faced when building the metacello browser. Metacello is a package management system for Pharo. The expressiveness and flexibility of its domain specific language quickly propelled Metacello within a few months as the main package system of Pharo, Squeak and Gemstone. However, Metacello remains quite greedy on memory and relatively slower than one would expect. In this paper, we will use Metacello as our motivating and running example. The blueprints presented in this paper were all obtained evaluating the following expression:

```
MemoryProfiler
  viewProfiling: [ (1 to: 10) collect: [ :i |  ConfigurationOfMetacello project currentVersion ] ]
```

The message viewProfiling: takes a block closure as argument. We perform 10 iterations to artificially increase the resources consumption. The experiment described in this paper is based on Version 1.0-beta.27.2 of Metacello.

[5] http://www.moosetechnology.org/tools/mondrian

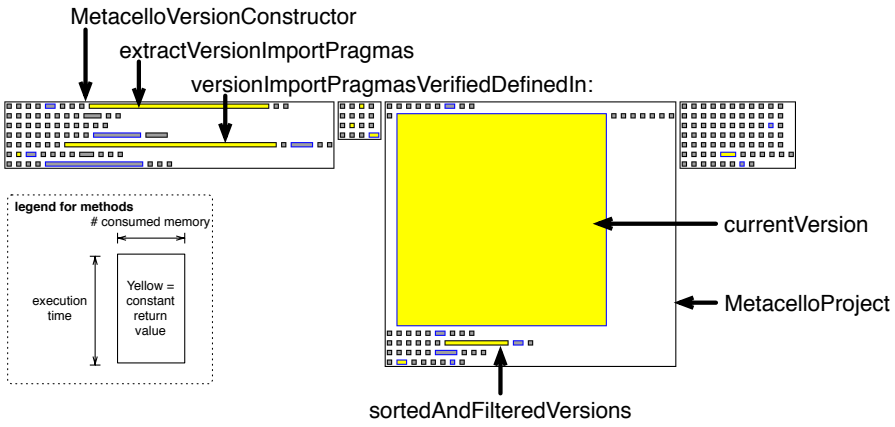


Figure 1: Structural memory distribution blueprint on Metacello.

*Structural Distribution.* The evaluation of the expression generates the memory distribution structural view (Figure 1). This visualization is intended to immediately spot methods that are greedy in memory. The visualization is structured according to the program static structure, in terms of classes and methods. The width and height of a method box represents the number of allocated bytes and the total excecution time, respectively. Figure 1 shows that asking for the current version of a Metacello project is costly both in time and memory since the method is represented as a tall and wide box. In addition, it is constant on its return value. We define by being constant when for each method execution the return value is equal (i.e., answers true to the message =) with the return value of its previous execution.

Figure 1 also shows that some other methods have a high memory consumption. This is the case for example of the methods extractVersionImportPragmas and versionImportPragmasVerifiedDefinedIn: of the MetacelloVersionConstructor class and sortedandFilteredVersions of MetacelloProject.

Invocations between methods are represented by changing the border color of the boxes that represent the methods calling and called by the selected method. It shows us that extractVersionImportPragmas is invoked by versionImportPragmasVerifiedDefinedIn:.

The specification of this view is given by the table below:

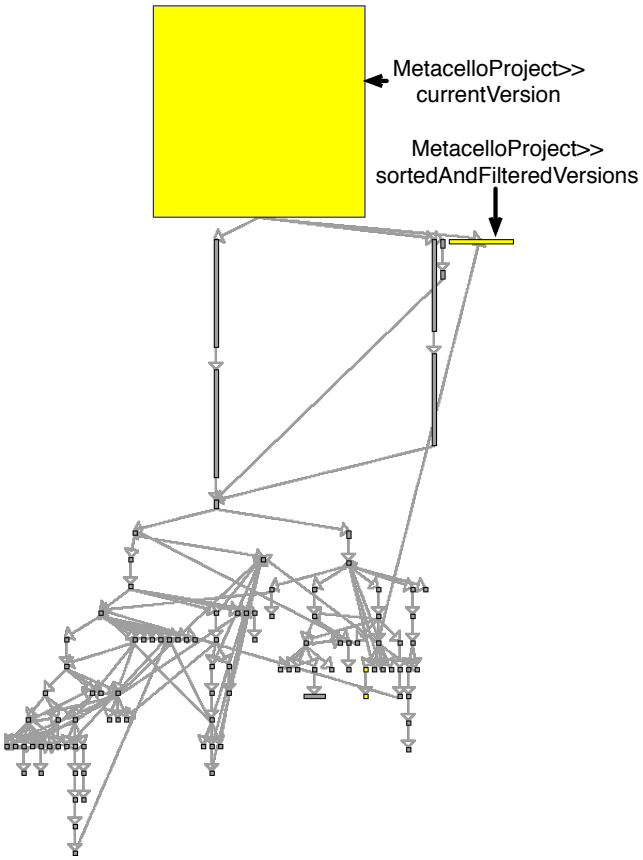


Figure 2: Behavioral memory distribution blueprint on Metacello.

| ***Memory structural distribution blueprint*** | |
|---|---|
| *Scope* | full system execution |
| *Edge* | class inheritance (upper is superclass of below) |
| *Layout* | tree layout for outer nodes and gridlayout for inner nodes (inner nodes are ordered by increasing height) |
| *Metric scale* | linear |
| *Node* | outer node is a class, an inner node is a method |
| *Inner node color* | Yellow indicates a method constant on its return value; Gray otherwise |
| *Inner node height* | total execution time of a method |
| *Inner node width* | number of allocated bytes |
| *Example* | Figure 1 |

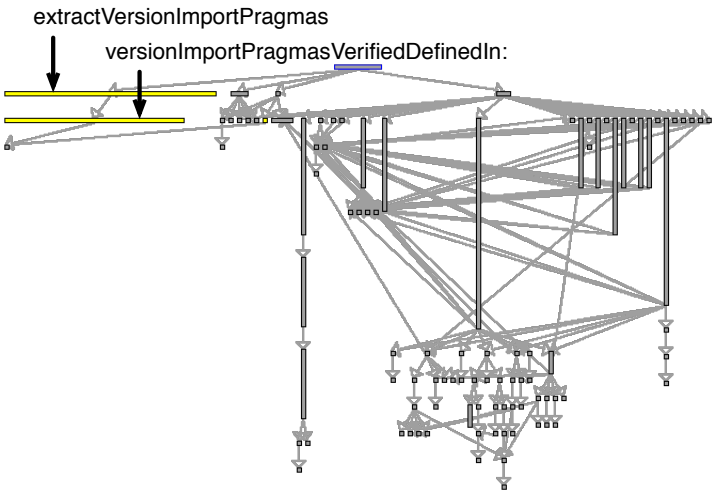


Figure 3: Second behavioral memory distribution blueprint.

*Behavioral distribution.* Right clicking on the method currentVersion of the class MetacelloProject, it opens the behavioral memory distribution, a complementary view that shows the consumption along the method call graph. Figure 2 depicts the call graph that involves the methods currentVersion and sotedAndFilteredVersions identified in the previous blueprint.

Figure 3 provides a further example of the behavioral blueprint by showing the call graph that involve extractVersionImportPragmas and versionImportPragmasVerifiedDefinedIn:.

The specification of the blueprint is defined as:

| ***Memory behavioral distribution blueprint*** | |
|---|---|
| *Scope* | full system execution |
| *Edge* | method invocation sequence |
| *Layout* | tree layout |
| *Metric scale* | linear (except for node width) |
| *Node* | method |
| *Node color* | Yellow indicates a method constant on its return value; Gray otherwise |
| *Node height* | total execution time of a method |
| *Node width* | number of allocated bytes |
| *Examples* | Figure 2, Figure 3 |

## 3. Optimizing Metacello

The methods in yellow in the previous blueprints are good indicator on where to optimize Metacello.

To keep track of our progress, we first measure the metrics we are likely to improve.

| ConfigurationOfMetacello project currentVersion | | |
|---|---|---|
| **# repetitions** | **time taken (ms)** | **memory consumed (bytes)** |
| 5 | 7, 096 | 2, 914, 700 |
| 10 | 14, 241 | 4, 790, 860 |
| 15 | 21, 801 | 7, 091, 124 |

The execution time has been obtained using timeToRun. The memory consumption is obtained with:

```
Smalltalk garbageCollect.
(MPMemoryConsumptionSnapshot
  during: [
        | project |
        project := ConfigurationOfMetacello project.
        (1 to: X) collect: [ :i |  project currentVersion ] ]) numberOfAllocatedBytes
```

Note that MPMemoryConsumptionSnapshot is a class part of our memory profiler.

From executing 10 times the expression ConfigurationOfMetacello project currentVersion, we measured the amount of time taken and the amount of memory used for the four methods we mentioned previousluy:

| **method** | **time (ms)** | **memory (bytes)** |
|---|---|---|
| currentVersion | 12912 | 111, 276 |
| sortedAndFilteredVersions | 20 | 34, 076 |
| versionImportPragmasVerifiedDefinedIn: | 20 | 112, 476 |
| extractVersionImportPragmas | 20 | 95, 240 |

The memory distribution blueprints (Figure 1 and Figure 2) clearly indicate that getting the current version of a configuration project is costly, both in memory and in time. As indicated by the blueprints, currentVersion is the culprit method. This is not a surprise considering the algorithm used to compute a project version number. The "best" version number is determined according to the version of each individual installed packages.

The source code of this method is:

```
MetacelloProject>> currentVersion
  | cacheKey |
  cacheKey := self configuration class.
  ^MetacelloPlatform current
        stackCacheFor: #currentVersion
        at: cacheKey
        doing: [:cache |  |cv versions |
        ...
        ].
```

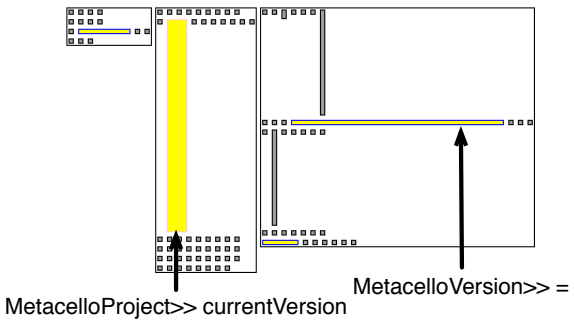


Figure 4: Structural distribution after the optimization.

Apparently, a rudimentary cache mechanism has been provided. However, it seems to be ineffective for the benchmark we are interested in. We augmented the class MetacelloProject with a new instance variable currentVersionCache. The currentVersion is then rewritten as:

```
MetacelloProject>> currentVersion
  | cacheKey |
  currentVersionCache ifNotNil: [ ^currentVersionCache ].
  cacheKey := self configuration class.
  ^currentVersionCache := MetacelloPlatform current
        stackCacheFor: #currentVersion
        at: cacheKey
        doing: [:cache | | cv versions |
        ...
        ].
```

For a given project object, calling twice currentVersion executes the whole body just once. We now have to make sure that asking multiple times a configuration for a project return the same object. The project method is defined as:

```
ConfigurationOfMetacello classproject
  ^self new project
```

We have rewritten this method into:

```
ConfigurationOfMetacello classproject
  project ifNotNil: [ ^ project ].
  ^project := self new project
```

Where project is a class variable of ConfigurationOfMetacello.

The new blueprint realized after these small changes is eloquent compared to the original excecution times:

| ConfigurationOfMetacello project currentVersion | | |
|---|---|---|
| **# repetitions** | **time taken (ms)** | **memory consumed (bytes)** |
| 5 | 1, 469 | 1, 644, 964 |
| 10 | 1, 477 | 1, 647, 252 |
| 15 | 1, 508 | 1, 723, 792 |

Executing 10 times the expression ConfigurationOfMetacello project currentVersion produces the following measurement:

| **method** | **time (ms)** | **memory (bytes)** |
|---|---|---|
| currentVersion | 1, 292 | 6, 000 |
| sortedAndFilteredVersions | 1 | 0 |
| versionImportPragmasVerifiedDefinedIn: | 1 | 0 |
| extractVersionImportPragmas | 1 | 0 |

Metacello performances have been significantly improved. The time taken to compute the current version of a Metacello configuration is now almost constant in time and in memory.

## 4. Conclusions

This short paper describes a real situation where excessive memory consumption and execution time were perceived as a critical issue in Metacello, a widely used mechanism to manage Pharo packages. As far as we are aware of, State of the Art code profilers are good at identifying what the problem is, but fixing the situation requires tremendous effort to understand what the situation is and how to address is. To that very purpose, we propose two visual representations of the program execution that relate the memory consumption with the execution time. Indication about the side effect is further provided.

We used our visualization to successfully address a serious memory consumption issue in Metacello. Our implementation is freely available for the Pharo programming language and is available on Squeaksource, the Pharo source forge[6].

## References

[1] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[6] http://www.squeaksource.com/Spy.html