Execution profiling blueprints

Alexandre Bergel¹, Felipe Bañados¹, Romain Robbes¹, Walter Binder²

¹*Pleiad Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile*

http://bergel.eu
http://www.dcc.uchile.cl/~fbanados
http://www.dcc.uchile.cl/~rrobbes

²University of Lugano, Switzerland

http://www.inf.usi.ch/faculty/binder

SUMMARY

While traditional approaches to code profiling help locate performance bottlenecks, they offer only limited support for removing these bottlenecks. The main reason is the lack of detailed visual runtime information to identify and eliminate computation redundancy. We provide three profiling blueprints which help identify and remove performance bottlenecks. The *structural distribution blueprint* graphically represents the CPU consumption share for each method and class of an application. The *behavioral distribution blueprint* depicts the distribution of CPU consumption along method invocations, and hints at method candidates for caching optimizations. The *behavioral evolution blueprint* compares profiles of different versions of a software system and highlights performance-critical changes in the system. These three blueprints helped us to significantly optimize Mondrian, an open source visualization engine. Our implementation is freely available for the Pharo development environment and has been evaluated in a number of different scenarios. Copyright © 2010 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Profiling; Visualization; Pharo

1. INTRODUCTION

Improving system performance is a permanent focus in research and in industrial software development. While the applicability of static program analysis to identify performance bottlenecks and hotspots is limited, particularly for software written in dynamic object-oriented languages, dynamic program analysis allows gathering detailed execution statistics for individual program executions. Profiling techniques are widely used for collecting dynamic metrics in order to locate hotspots that are the target for subsequent program optimizations.

However, whereas the profiles produced by many prevailing profilers offer detailed execution time statistics for individual methods, they often fail to explain the reason why a certain method is slow, e.g., which particular method arguments cause an expensive computation. For instance, gprof, which appeared in 1982, offers a number of textual reports focused on "how much time was spent executing directly in each function" and on call graphs*. Almost 30 years later, it is interesting to notice that this output has not evolved much. Consider JProfiler, a award winning profiler for Java which is actively developed. JProfiler essentially produces the same output, using a graphical rendering instead of a textual one[†]. Whereas JProfiler uses tremendously more

*http://sourceware.org/binutils/docs/gprof/Output.html#Output

[†]http://www.ej-technologies.com/products/jprofiler/screenshots.html

sophisticated profiling techniques than its predecessors, it still considers the output of its profiling activity as a mere tree widget to indicate the CPU consumption. It appears that a large part of the research conducted in the field of code profiling focuses on reducing the overhead triggered by the code instrumentation and observation [1, 2].

The set of dynamic metrics and visualizations used to profile object-oriented applications are very similar to those used to profile procedural applications. When we retrospectively look at the history of execution profilers, we see that tool usability and profiling overhead reduction have steadily improved, but the offered visualizations have not changed much. Prevailing profile visualizations do not properly address the specific structure of object-oriented software. Dynamic metrics are presented in a similar form when profiling an application written in Java or in C. In both cases, the call frame is the main element of the generated profiles.

Tracing application performance across different versions of an evolving application is another weak point in prevailing execution profilers. Profiles are usually gathered for a given snapshot of a program. Questions like "Which method of a particular software version is faster than in the current version?" can hardly be answered with current profilers, without resorting to manual screening and differentiation of the profiles. Most profilers do not offer any dedicated support for comparing profiles (e.g., hprof[‡], JProbe[§]).

In this article we apply some visualizations that have been previously used in static software analysis in order to display dynamic metric for profiling purposes. We describe visualizations for rendering dynamic information that effectively enables comparison of different metrics related to a program execution. *Structural distribution blueprint* and *behavioral distribution blueprint* are two visualizations intended to identify bottlenecks and to give hints on how to remove them. The first blueprint represents the distribution of the CPU effort along the program structure. The second blueprint directs the distribution along method invocations and identifies methods prone to one class of optimization, namely caching.

This article builds on our previously published work on blueprint profiling [3]. Since then, we have constantly used our blueprints to address performance issues in our software development activities. Our experience made us realize that exploiting the history of a software plays an important role in understanding performance of a particular software version. Identifying where the speedup gained from a particular optimization in the past got lost in subsequent software versions requires considering the evolution of the source code. We address this issue with the new *behavioral evolution blueprint*, which is the original scientific contribution of this article.

The work presented in this article aims at complementing existing profilers with new visualizations that help specific optimization tasks. We obtained the results presented in this article using Pharo[¶], an open-source Smalltalk-dialect programming language. We apply our profiling techniques to the visualization framework Mondrian^{||} [4], our running example.

Intuitively the blueprint profiling techniques presented in this article may be applied to applications written in any other object-oriented programming language as well. However, the strategy we used to draw our conclusion is characterized as idiographic [5], meaning that we focus only on the phenomenon in the context of Pharo and the applications in which we found performance issues. The points we fix in our work are Pharo, its scheduling strategy and the size of the applications we considered for our experiment (topping at 200 classes and 2000 methods). Even though profiling long living applications, including large scale servers, is indeed a major challenge [6], our work focuses on medium-sized applications (\sim 20 000 lines of code) with rather short profiling execution time (less than 1 minute).

The profiler providing the visualizations presented in this article is publicly available in Pharo** under the MIT license.

[‡]http://java.sun.com/developer/technicalArticles/Programming/HPROF.html

^{\$}http://www.quest.com/jprobe

[¶]http://www.pharo-project.org/home

http://www.moosetechnology.org/tools/mondrian

^{**}http://www.squeaksource.com/Spy.html



Figure 1. Principle of polymetric view.

Note that the structural distribution blueprint is made of gray and black only. The behavioral distribution blueprint and behavioral evolution blueprint uses colors.

This article is structured as follows: We first describe the structural and behavioral distribution blueprints (Section 2). We then identify and implement opportunities for optimization in Mondrian (Section 3). Section 2 and Section 3 are taken from our previous publication [3]. Next, we investigate factors impacting the performance evolution of two systems, Mondrian and GitFS, along their development history (Section 4) and discuss the experience we gained (Section 5); this part constitutes the novel contribution of this article. We then review related work (Section 6) and conclude (Section 7).

2. PROFILING BLUEPRINTS

2.1. Profiling blueprint in a nutshell

Time profiling blueprints are graphical representations meant to help programmers (i) assess the time distribution and (ii) identify bottlenecks and give hints on how to remove them for a given program execution. The essence of profiling blueprints is to enable a better comparison of elements constituting the program structure and behavior. To render information, these blueprints use a graph metaphor, composed of nodes and edges.

The size of a node hints at its importance in the execution. In the case that nodes represent methods, a large node may say that the program execution spends "a lot of time" in this method. The expression "a lot of time" is then quantified by visually comparing the height and/or the width of the node against other nodes.

Color is used to either transmit a property (*e.g.*, a yellow node represents a method that always returns the same value) or a metric (*e.g.*, a color gradient is mapped to the number of times a method has been invoked).

We propose two blueprints that help identify opportunities for code optimization. They provide hints to programmers to refactor their program along the following two principles: (i) make oftenused methods faster and (ii) call slow methods less often. The metrics we adopted in this article help finding methods that are either unlikely to perform a side effect or return always the same result, good candidates for simple caching optimizations.

2.2. Polymetric views

The blueprints we propose are graphically rendered as *polymetric views* [7]. A *polymetric view* is a lightweight software visualization enriched with software metrics. It has been successfully used to provide "software maps" intended to help software comprehension and visualization. Figure 1 depicts the general aspect of a polymetric view.

Given two-dimensional nodes representing entities, we can map up to 5 metrics on the node characteristics: position (X and Y), size (width and height), and color:

• *Position*. The X and Y coordinates of the position of a node may reflect two measurements.

Structural distribution blueprint			
Scope	full system execution time		
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 (except for node width, which is logarithmic)		
Node	outer node is a class, an inner node is a method		
Inner node color	Number of different receivers		
Inner node height	total execution time of a method		
Inner node width	number of executions (logarithmic scale)		
Example	Figure 2		

Table I. Specification of the structural distribution blueprint.

- *Size*. The width and height of a node can render two measurements. We follow the intuitive notion that the wider and the higher the node, the larger the associated metric.
- *Color*. The color interval between white and black may render one measurement. The convention that is usually adopted [8] is that the higher the measurement, the darker the node. Thus light gray represents a smaller measurement than dark gray.

Edges may also render properties along a number of dimensions (width, color, direction, etc.). However, for the purpose of this work, all edges are identical.

2.3. Structural distribution blueprint

The execution of an object-oriented program yields a large amount of information [9] (*e.g.*, number of objects created at runtime, total execution time of a method). Unfortunately, all these dimensions cannot be visually rendered in a meaningful fashion. The *structural distribution blueprint* displays a selected number of metrics indicating the distribution of the execution time along the static structure of a program (*i.e.*, classes, methods and class hierarchy). Table I gives the specification of the *structural distribution blueprint*. The blueprint renders a program in terms of classes, methods and inheritance relations. Each method representation exhibits its corresponding CPU time profiling information along three metrics:

- *number of different receivers*: amount of different object receivers the method has been invoked on. Due to implementation limitations, this is at the moment a lower bound estimate.
- *total execution time of a method*: time for which a call frame corresponding to the method is present on the stack at runtime. The precision depends on the underlining profiler used to collect runtime information.
- *number of executions*: number of times the method has been executed, independently of the object receiver.

Actual metric values, and additional information, are accessible through a contextual popup window.

Example. Throughout this article, we use the graph visualization framework Mondrian as a case study. The blueprints described in this article are also rendered using Mondrian. An example of the structural distribution blueprint is given in Figure 2. Four classes are represented: MOGraphElement, MOViewRenderer, MONode and MORoot. This figure is a small part of a bigger picture obtained by evaluating the following code snippet, which renders a simple visualization of 100 nodes, each containing 100 nodes:



Figure 2. Example of a structural blueprint.

```
ProfilingPackageSpy

viewProfiling: [

| view |

view := MOViewRenderer new.

view nodes: (1 to: 100)

forEach: [:each | view nodes: (1 to: 100)].

view root applyLayout ]

inPackage: 'Mondrian'
```

The code being profiled is indicated using a **bold font** in the example source code. The profiling is realized from the perspective of one package, Mondrian in our case. MOGraphElement inherits from MONode, MORoot from MOGraphElement, and MOViewRenderer from Object. Since Object does not belong to Mondrian (but to the Kernel package), it is not rendered in the blueprint.

The height of a method node is proportional to the total execution time taken by the method (e.g., 53%) of the code execution is spent in the method #applyLayout and 38\% in #bounds). The width is proportional to the number of times the method has been executed. A logarithmic scale is used. The method node color represents the number of different objects this method has been executed on (more than 3 732). The scope of the blueprint is global, which means that the darkest method corresponds to the method that has been executed on the greatest number of object receivers, system-wide.

Moving the mouse over a method node pops up additional contextual information. In the example, the contextual window says that the method #applyLayout defined in the class MOGraphElement has been executed 10 100 times, and has been executed on more than 3 732 distinct receiver objects (*i.e.*, instances of MOGraphElement or one of its subclasses). It is also indicated that this method returns always the same value for a given object receiver. While the blueprint emphasizes the three metrics indicated above, the contextual information provides useful data when one wants to know more about a particular method.

Within a class, methods are ordered along their height. This helps quickly spot the amount of costly methods. For example, it is clear that among MOGraphElement's methods, 3 are dominating with respect to execution time.

Interpretation. Classes represented in Figure 2 illustrate part of a scenario that totals 11 classes. Among the 111 classes that define Mondrian, these 11 classes are the only classes involved in the code snippet execution given above. Only classes that are covered by the execution, even partially, are depicted in the blueprint.

MOGraphElement contains "many large and dark" methods. This indicates that this class is central to the code snippet execution: these large and black methods consume a lot of CPU time and are invoked on many different instances. Almost all of MOGraphElement's methods are executed a large number of times: in the visualization, they are quite wide compared to methods in other classes. For most of them, this is not a problem because they are thin and horizontal: even if these methods are executed many times, they do not consume CPU time. On the left of #applyLayout stands the #bounds method. This method takes 38% of the CPU time and is invoked 70 201 times on more than 3 732 object receivers. The third costliest method on MOGraphElement, #shapeBoundsAt:ifPresent: takes 33% of the CPU time. MONode contains a black and relatively large method: MONode>>translateTo: consumes 22% of the total CPU time. The method has been invoked 10 100 times on at least 3 732 receivers.

Comparing to MOGraphElement, we find that classes are not involved in the computation as much. The representation of MOViewRenderer quickly says that its methods are invoked a few times without consuming much CPU. Moreover, methods are white, which tells that they are invoked on few instances only. The contextual information obtained by moving the mouse over the methods reveals that these methods are executed on a unique receiver. This is not surprising since only one instance of MOViewRenderer is created in the code example given above.

MORoot also does not seem to be the cause of a bottleneck at runtime. The few methods of this class are not frequently executed since they are relatively narrow. MORoot also defines a method #applyLayout. This method is the tall, thin and white method. The contextual information reveals that this method is executed once and on one object only. It consumes 97% of the execution time; this means that the CPU spent 97% of the time in the the method #applyLayout or in one of the methods recursively called by #applyLayout. The method MORoot>>applyLayout invokes MOGraphElement>>applyLayout on each of the nodes. The relation between these two #applyLayout methods is indicated by a fly-by-highlighting (not represented in the picture) and the *behavioral distribution blueprint*, described below.

All in all, a large piece of the total CPU time is distributed over four methods: MONode>>translateTo: (24%), MOGraphElement>>bounds (32%), MOGraphElement>>shape-BoundsAt:ifPresent: (33%), MOGraphElement>>applyLayout (53%). Note that at this stage, we cannot say that the CPU time share of these three methods is the sum of their individual share. We have 24 + 32 + 33 + 53 = 142. This indicates that some of these methods call each other since their sum cannot exceed 100%.

2.4. Behavioral distribution blueprint

In a pure object-oriented setting, computation is essentially performed through message sending between objects. The CPU time consumption is distributed along method executions. Assessing the runtime distribution along method invocations complements the structural distribution described in the previous section. To reflect this profiling along method invocations, we provide the *behavioral distribution blueprint*. Table II gives the specification of the figure.

The goal of this blueprint is to assess runtime information alongside method call invocations. It is intended to find optimization opportunities, which may be tackled with caching. In addition to the metrics such as the number of calls and execution time, we also show whether a given method returns constant values, and whether it is likely to perform a side effect or not. As shown later, this information is helpful to identify a class of bottlenecks.

Behavioral distribution blueprint		
Scope	all methods directly or indirectly invoked for a given	
	starting method	
Edge	method invocation (upper methods invoke lower ones)	
Layout	tree layout	
Metric scale	linear (except for node width)	
Nodes	methods	
Node color	gray: return always self; yellow: same return value per	
	object receiver; white: remaining methods	
Node height	total execution time	
Node width	number of executions (logarithmic scale)	
Example	Figure 3	

Table II. Specification of the behavioral distribution blueprint.

Classes do not appear on this blueprint. Methods are represented by nodes and invocations by directed edges. The blueprint uses the two metrics described in the previous blueprint for the width and height of a method. In addition to the shape, node color indicates a property:

- the gray color indicates methods that return self, the default return value. When no return value is specified in Pharo, the object receiver is returned. This corresponds to void methods in a statically typed language. No result is expected from the method, strongly suggesting that the method operates via side effects.
- the yellow color (which appears as light gray on a black and white printout) indicates methods that are constant on their return value, this value being different from self.
- other methods are white.

As shown in the coming section, keeping track of the returned value in such a way is beneficial in our experience setting. We will later on detail

A tree layout is used to order methods, with upper methods calling lower methods. We illustrates this blueprint on the MOGraphElement>>bounds method that we previously saw, a candidate for optimization.

Example. In the previous blueprint (Figure 2), right-clicking on the method MORoot>>applyLayout opens a behavioral distribution blueprint for this method. The complete picture is given in Figure 3. The picture has to be read top-down. Methods in this blueprint have the same dimensions as in the behavioral blueprint. We recognize the tall and thin MORoot>>applyLayout at the top. All methods in Figure 3 are therefore invoked directly or indirectly by MORoot >>applyLayout. MORoot>>applyLayout invokes 3 methods, including MOGraphElement>>applyLayout (labelled in the figure). MOGraphElement>>applyLayout calls MOAbstractLayout>>applyOn:, and both of these are called by MORoot>>applyLayout.

Interpretation. As the first blueprint revealed, #bounds, #applyLayout, #shapeBoundsAt:ifPresent:, #translateTo: are expensive in terms of CPU time consumption. The behavior blueprint highlights this fact from a different point of view, along method invocations. In the following we will optimize #bounds by identifying the reason of its high cost and provide a solution to fix it. Our experience with Mondrian tells us that this method has a surprisingly high cost. Where to start a refactoring among all potential candidates remains the programmer's task. Our blueprint only says "how it is" and not "how it should be", however it is a rich source of indication of what's going on at runtime.

The return value of MOGraphElement>>bounds is constant over time, hence it is painted in yellow. This method is involved in a rich invocation graph (presented in Figure 3). In general,



Figure 3. Example of a behavioral blueprint.



Figure 4. Detailed view of MOGraphElement>>bounds.

understanding the interaction of a single method is likely to be difficult when a complete call graph is used. The contextual menu obtained by right-clicking on a method offers a filtered view on the entity of interest. Figure 4 shows a detailed view of a behavioral blueprint, centered on MOGraphElement>>bounds. This method is at the center of the picture. Above are located the methods calling #bounds. Below, the unique method that is called by #bounds. Among the 5 methods that call #bounds, 3 always return the same value when executed. The method called by #bounds also remains constant on its return value. Figure 4 renders #bounds and #shapeBoundsAt:ifPresent: with the same width. It is therefore likely that these two methods are invoked the same number of times. The contextual window indicates that each of these two methods is invoked 70 201 times. We can deduce the following:

- #bounds belongs to several execution paths in which each method is constant on its return value. This is indicated in the upper part of Figure 4.
- #bounds calls #shapeBoundsAt:ifPresent:, which is constant on return value.
- #bounds and #shapeBoundsAt:ifPresent: are invoked the same number of times.

The following section addresses this bottleneck by adding a cache in **#bounds** and unveils another bottleneck in Mondrian.

3. OPTIMIZING MONDRIAN

The combination of the structural and behavioral blueprints helped us to identify a number of bottlenecks in Mondrian. In this section, we address some of these bottlenecks by using memoization^{††}, i.e. we cache values to avoid redundant computations.

3.1. Bottleneck MOGraphElement>>bounds

As we saw earlier, the behavioral blueprint on the method MOGraphElement>>bounds reveals a number of facts about the program's execution. These facts are good hints that #bounds will benefit from a caching mechanism since it always returns the same value and calls a method that is also constant. We inspect its source code:

```
MOGraphElement>>bounds

"Answer the bounds of the receiver."

| basicBounds |

self shapeBoundsAt: self shape ifPresent: [ :b | ^ b ].

basicBounds := shape computeBoundsFor: self.

self shapeBoundsAt: self shape put: basicBounds.

^ basicBounds
```

The code source confirms that #shapeBoundsAt:ifPresent: is invoked once each time #bounds is invoked. The method #shape is also invoked at each invocation of #bounds. The contextual window obtained in the structural blueprint reveals that the return value of #shape is constant: It is a simple variable accessor ("getter" method), so it is fast. #bounds calls #computeBoundsFor: and #shapeBoundsAt:put: in addition to #shapeBoundsAt:ifPresent: and #shape. However, they do not appear in Figure 3 and 4. This means that #bounds exits before reaching #computeBoundsFor:. The block [:b | $^{\circ}b$], which has the effect of exiting the method, is therefore always executed in the considered example.

We first thought that the last three lines of **#bounds** may be removed since they are not executed in our scenario. However, the large number of tests in Mondrian indicate that these lines are indeed important in other scenarios, although not in our particular example.

We elected to upgrade #bounds with a simple cache mechanism. Differences with the original version are indicated using a **bold font**. The class MOGraphElement is extended with a new instance

^{††}http://www.tfeb.org/lisp/hax.html#MEMOIZE



Figure 5. Upgrading #bounds has a global structural impact.

variable, **#boundsCache**. In addition, the cache variable has to be reset in 5 methods related to graphical bounds manipulation of nodes, such as translating and resizing.

```
MOGraphElement>>bounds

"Answer the bounds of the receiver."

| basicBounds |

boundsCache ifNotNil: [ ^ boundsCache ].

self shapeBoundsAt: self shape ifPresent: [ :b | ^ boundsCache := b ].

basicBounds := shape computeBoundsFor: self.

self shapeBoundsAt: self shape put: basicBounds.

^ boundsCache := basicBounds
```

There is no risk of concurrent accesses of **#boundsCache** since this variable is set when the layout is being computed. This occurs before the display of the visualization, which is done in a different thread.

Result. Adding a statement boundsCache ifNotNil: [^ boundsCache] significantly reduces the execution time of the code given in Section 2.3. Before adding this simple cache mechanism, the code took 430 ms to execute (on a MacBook Pro, 2Gb of RAM (1067 MHz DDR3), 2.26 GHz Intel Core 2 Duo, Squeak VM 4.2.1beta1U). With the cache, the same execution takes 242 ms only, which represents a speedup of approximately 43%.

This gain is reflected on the overall distribution of the computational effort. Figure 5 provides two structural blueprints of the code snippet given in Section 2.3. The comparison between the two version is done "manually", in Section 4 we will introduce a new visualization meant to support such comparison systematically. The upper blueprint has been produced before upgrading the method MOGraphElement>>bounds. Figure 2 is a part of it. The lower one has been produced after upgrading #bounds as described above. Many places are impacted. We annotated the figure with the most significant changes:



Figure 6. Upgrading #bounds has a global behavioral impact.

- the size of the **#bounds** method and the methods invoked by it (C) have seen their height significantly reduced. Before the optimization, **#bounds** used 38% of the total CPU consumption. After the optimization, its CPU use fell to 5%.
- the 5 methods denoted by the circle A and B have seen their height increased and their color darkened. The height increase illustrates the augmentation in relative CPU consumption these methods are subject to, now that #bounds has been improved.

The evolution of the behavioral blueprint is presented in Figure 6. We can clearly see the reduced size of #bounds and #shapeBoundsAt:ifPresent: (Circle B) and the increase of the #applyLayout method (Circle A).

3.2. Bottleneck in MONode>>displayOn:

We fixed an important bottleneck when computing bounds in Mondrian. We push our analysis of bounds computing a step further. We inspect the User Interface (UI) thread of Mondrian. Most applications with a graphical user interface run in at least 2 threads: one for the program logic and another in charge of receiving user events (e.g., keystrokes, mouse events) and virtual machine/OS events (e.g., window refreshes). Mondrian is no exception. The blueprints presented earlier focused on profiling the application logic.

Step 1. Figure 7 shows the structural profiling of the UI thread for the Mondrian script given in Section 2.3. The blueprint contains many large methods, indicating methods that received a significant CPU share. Among these, our knowledge of Mondrian leads us to #absoluteBounds. This method is very similar to #bounds that we previously saw. It returns the bounds of a node using absolute coordinates (instead of relative). The UI thread spends most of the time in MONode>>displayOn: since it is the root of the thread's computation.

Figure 8 shows the behavioral blueprint opened on MONode>>displayOn:. The blueprint reveals that #absoluteBounds and #absoluteBoundsFor: call each other. Return values of these two methods are constant as indicated by their yellow color. They are therefore good candidates for caching:



Figure 7. Profiling of the UI thread in Mondrian.



Figure 8. Profiling of the UI thread in Mondrian.

MOGraphElement>>absoluteBounds

Result. Without the cache in **#absoluteBounds**, the scenario takes 356 ms to run. With the cache, it takes 231 ms. We therefore gained 35% when displaying the visualization.

Step 2. By adding the cache in #absoluteBounds, we significantly reduced the cost of this method. We can still do better. As shown in Figure 8, there is another caller of #absoluteBounds. MORectangleShape>>display:on: is 85 lines long and begins with:

```
MORectangleShape>>display: aFigure on: aCanvas
| bounds borderWidthValue textExtent c textToDisplay font borderColorValue ... |
bounds := self absoluteBoundsFor: aFigure.
c := self fillColorFor: aFigure.
...
```

We saw in Step 1 that #absoluteBounds calls the expensive and uncached #absoluteBoundsFor:. Replacing the call to #absoluteBoundsFor: by #absoluteBounds improves performance further:

```
MORectangleShape>>display: aFigure on: aCanvas
| bounds borderWidthValue textExtent c textToDisplay font borderColorValue ... |
bounds := aFigure absoluteBounds.
c := self fillColorFor: aFigure.
...
```

Result. The execution time of the code snippet has been reduced to 198 ms. A speedup of 14% from Step 1, and of 45% overall.

Blueprint evolution. Figure 9 summarizes the two evolution steps described previously. Differences with a previous step are denoted using a circle. The effect of caching #absoluteBounds considerably diminished the execution time of this method. This is illustrated by Circle C. It has also the effect of reducing the size of MOShape's methods and increasing MORectangleShape>>display:on:. The share of the CPU consumption increased for this method. Step 2 reduced the size of MOShape's method. Their execution times became so small, that they does not appear in the behavioral blueprint (since we use a sampling-based profiler to obtain the runtime information, methods having less than 1% of the CPU do not appear in this blueprint).

3.3. Summary

The cache value of MOGraphElement>>bounds (Section 3.1) is implemented and has been finalized in the version 341 of Mondrian^{‡‡}. The improvement of #absoluteBounds and #display:on: may be found in the version 352 of Mondrian. The complete experiment led to a 43% improvement in creating the layout of a view, and of 45% in displaying the same view.

We identify and remove a number of bottlenecks. From this experience, it is tempting to identify and look after some general patterns that would easily expose fixable execution bottleneck. Unfortunately, we haven't see the opportunity to deduce some general rules. The visualization we provide clearly identify costly methods and classes, potentially being candidates for optimization. Whether the optimization can be easily realized or not depends heavily on a wide range of parameters (*e.g.*, algorithm, architecture, data structure).

4. BEHAVIORAL EVOLUTION BLUEPRINT

We have presented profiling blueprints as a graphical representation of a software execution for a particular version of the software. Although useful to identify execution bottlenecks, comparing two

^{‡‡}The source code is available at: http://www.squeaksource.com/Mondrian.html



Figure 9. Profiling of the UI thread in Mondrian.

Softw. Pract. Exper. (2010) DOI: 10.1002/spe



Figure 10. Behavioral evolution blueprint for the version 416 and 425 of Mondrian.

or more versions becomes hard as it requires an exhaustive manual check for every software entity. Focusing on the version differences instead of the actual values is the topic of this section.

To compare the profiles over two software versions, we propose the *behavioral evolution blueprint*. This additional behavioral blueprint depicts the method call graph in which nodes are methods, and edges invocations. The height of a node is the share of the total execution time taken by the represented method. The width shows the number of times the method has been invoked, based on a logarithmic scale. The differences between methods are usually so great that a linear scale is not effective. Colors—or grayscale equivalents in this version—convey the difference of a metric over different software versions, and border thickness refers to changes in the source code. This blueprint can be used to graphically validate when a source change improves the performance of the code. The specification of this visualization is given in Table III.

Figure 10 exemplifies the behavioral evolution blueprint on the method MORoot >>applyLayout for two versions of Mondrian, versions 416 and 425. The profiling is realized on version 425, and colors show the result of a comparison with a similar profiling based on a previous version, version 416 in this case. Each called method is colored like so:

- Green (light gray, thick borders in grayscale): the method source code has been modified from the previous software version and its total execution time decreased.
- Light green (light gray, thin borders): the method's source code is unchanged, but its total execution time decreased.
- Yellow (black): the method was not present in the previous software version, but is present in the current one.
- Red (dark gray, thick borders): the method source code has been modified and its total execution time increased.
- Light red (dark gray, thin borders): the method is unchanged, but its execution time increased.

Behavioral evolution blueprint				
Scope	all methods directly or indirectly invoked for a given			
	starting method			
Edge	method invocation (upper methods invoke lower ones)			
Layout	tree layout			
Metric scale	linear (except for node width)			
Nodes	methods			
Node color	Green (light gray in grayscale): method execution			
	time is less than previous version; Red (dark gray			
	in grayscale): method execution time is greater than			
	previous version; Yellow (block in grayscale): method			
	wasn't implemented in previous version; White:			
	method execution time is identical.			
Node height	total execution time			
Node width	number of executions (logarithmic scale)			
Node border thickness	thin: source code identical; thick: source code is			
	different than previous version			
Example	Figure 10			

Table III. Specification of the behavioral evolution blueprint.

• White: No change in execution time.

The red and light red colors quickly identify the methods that are slower in the profiled version than they previously were. Green and light green colors identify methods that are faster in the new version. However, the performance difference might not be directly related to the particular method, as it might be a side effect of changes in other methods being called. Therefore, we need to distinguish methods whose source code *and* performance has changed, which are the candidates to cause the overall performance difference. The use of strong and light colors—complemented by thick and thin borders—helps highlighting these particularly interesting methods. A strong color is assigned when the source code is different from its previous version. Light colors mean the source code is the same. In the case of yellow, as the method is not part of the call graph of the previous version, there is no need for a light color property.

The depicted blueprint (Figure 10) is obtained from the following steps:

- (i) profile a piece of code for Version 416 of Mondrian (we used the code snippet given in Section 2.3 for the figure), and store source code information for every method reached by the profile;
- (ii) repeat step (i) for a second profile of the same code snippet for Version 425 of Mondrian;
- (iii) for each method covered in Version 425, we subtract its total execution time with the metric obtained when profiling version 416;
- (iv) for every method in step (iii) we compare source code versions for changes.
- (v) we generate the blueprint using the call graph obtained from the execution of the piece of code in Version 425.

Only the methods that are defined in Version 425 are represented. Methods that are present in Version 425 but not in Version 416 are indicated in yellow. Methods not present in 416 are simply not represented.

As the source code of MORoot >>applyLayout does not change between both versions but its performance in terms of execution time does, we want to use the Behavioral Evolution Blueprint in

Figure 10 to trace the origin of this change.

Searching for the origin of the change. The coloring in the blueprint is helpful to find the main source of the change in performance. As seen in Figure 10, since the root node border is thin, the source code of MORoot >>applyLayout is the same in Version 425 than in Version 416. However, its execution time has changed, since it is colored in dark gray.

The first thick bordered node in the second branch of children of the original method corresponds to the method MOGraphElement >>translateBy:bounded:. The version 425 of this method is slower then the version 416 as the dark gray color indicates. The method name—along with the current displayed version and metric values—is available in a contextual popup window when the mouse hovers upon the method. Upon browsing the source code of the method, we see that the slowdown is due to moving the call to **#resetCachelnEdges** above in the method:

```
translateBy: aPoint bounded: bounded
"It moves the element by aPoint.
If bounded is true and the owner is not the root,
then the bounds are limited by the owner bounds.
If the element is placed in the root, then the root's bounds are updated"
|realStep newRelativePosition allShapes wasMyShapeUpdated |
```

self resetCacheInEdges. "line present in Version 425, not in 416"

self shapeBounds isNil ifTrue: [^ self bounds]. self shapeBounds isEmpty ifTrue: [^ self bounds].

self allNodesDo: [:n |n translateAbsoluteCacheBy: aPoint]. self resetCacheInEdges. "line present in Version 416, not in 425"

Revisiting the Mondrian optimizations. We used the behavioral evolution blueprint to compare the performance of Mondrian before and after the cache implementation for the bottlenecks discussed in Section 3.1 and Section 3.2. We can use the visualization to track the performance improvements directly to the cache implementation changes. From a global overview we found that the performance in MOEdge>>displayOn: improved by almost 20% between versions 300 and 352, while its source code did not change. As seen in Figure 11, that improvement is trackable to an over 300% improvement in MOGraphElement>>absoluteBounds, which coincides with the implementation of Section 3.2.

The optimization described in the previous sections are part of a major effort to optimize Mondrian. This effort started after Version 200. To compare our optimizations, we use Version 200 as the referential. When compared with Version 352, we identify an optimization implemented somewhere between Version 200 and 352: method MORoot>>applyLayout is faster in Version 352 and its source code has slightly changed (some methods have been renamed), changing from the version on the left hand side to the one on the right hand side (**bold font** indicates differences):

self shapeBounds	self shapeBoundsAt: self shape
at: self shape put: (0@0 corner: 0@0).	put: (0@0 corner: 0@0).
self do: [:each	self do: [:each
each applyLayout].	each applyLayout].
self layout applyOn: self.	self layout applyOn: self.
self do: [:each	self nodesDo: [:each
self bounds corner:	self bounds corner:
(self bounds corner max:	(self bounds corner max:
(each extent + each origin)).	(each extent + each origin)).
].].

With the Behavioral Evolution Blueprint in Figure 12 only three methods remain candidates to the source of the improvement. First, MOAbstractLayout >>initializeConnectionPositions, but its execution time is too small to cause the global improvement. Second, MORectangleShape >>computingExtentHavingChildrenFor and third MONode>>translateTo. The third method



Figure 11. Behavioral evolution blueprint of the UI thread over versions 300 and 352, rooted in MOEdge>>displayOn:.



Figure 12. Behavioral Evolution Blueprint over versions 200 and 352 rooted in MORoot>>applyLayout.

decreases from 64 to 1ms. Its source code was changed to include memoization, hence calling less times MOGraphElement>>bounds—a method that later added caching as seen in Section 3.2. Comparing the profiling obtained with Version 353 against the one obtained with Version 200

retrospectively assess the performance of Mondrian after 7 months (Version 200 was released on May 14, 2009 and Version 352 on January 25, 2010).

Navigating through history. The Behavioral Evolution Blueprint easily compares two execution profiles, each profile supposedly obtained from a particular software version. The profile that is compared with the anterior profile may be chosen by means of a contextual menu. This menu allows for moving through the history in a frame-by-frame basis. Clicking the "next" comparison button (not shown in the figures) for a particular method renders a new blueprint where the second profile of the previous blueprint becomes the first, and is compared to the profile associated with the next version available. A "previous" button is also available to navigate in the opposite direction. Note that the analysis data for the previous versions are stored and reused. Navigating through the history of the profile is therefore without noticeable lag.

In order to navigate the profiling history, we require a set of profiles taken over the execution of a particular code snippet throughout several versions of a software system. After selecting a particular set of versions where profiles are to be taken, we sort them from the oldest to the newest. The set of profiles begins with the first version's profile. Steps (ii) to (iv) are then repeated for each subsequent version of the system, loading, profiling, and comparing it with the last available. The profiling information is then stored in a similar approach to the one proposed in [10] for source code history.

Experience on Mondrian. We applied the previous algorithm to obtain a behavioral evolution blueprint analyzing the method MORoot >>applyLayout through a set of 22 versions, ranging from 400 to 570.

Navigation through the set of blueprints obtained made visible variations in the execution time associated with the cache implementation. Between versions 450 and 460, the call of a method MONode >>resetCacheInEdges in method MONode >>translateBy:bounded: was introduced as an effort to homogenize the various cache mechanisms Mondrian provides. This new method produced an increase of the execution time for MORoot >>applyLayout.

The time increase of MORoot >>applyLayout is, in contrast, reduced in Version 589 compared to Version 510. This is caused by changes in the method MOGraphElement >>applyLayout. Version 589 of this method contains an extra initialization of the shape, which significantly reduce the cost of layouting nodes:

```
MOGraphElement >>applyLayout
| b |
self resetMetricCachesResursively.
b := self hasChildren
ifTrue: [ 0 @ 0 extent: 0 @ 0 ]
ifFalse: [ self shape computeBoundsFor: self ].
self shapeBoundsAt: self shape put: b. "Added in Version 589"
self hasChildren
ifTrue: [
self do: [ :each | each applyLayout ].
self layout applyOn: self ].
```

Complementary case study. Beside Mondrian, we have used the behavioral evolution blueprint on a number of different Pharo projects. For example, GitFS^{*} is an implementation of Git in Pharo. At the time this article is being written, 91 different versions of GitFS are published. Git is a distributed revision control system with an emphasis on speed[†]. As a representative benchmark for a revision control system, we measured a sequence of 500 commits made on a file present in memory (no system primitives are therefore involved).

The method FSGitFilesystem>>commit: is responsible to commit a change into a new file. This method is the entry point to realize a Git commit operation. For Version 70, our benchmark made

^{*}http://www.squeaksource.com/GitFS.html
thttp://git-scm.com



Figure 13. Behavioral evolution blueprint of Version 80 compared with Version 70 of GitFS.

the #commit: method takes 2 541 milliseconds to execute. For Version 80, #commit: took 4 255 milliseconds, which represents a slowdown of $67\%^{\ddagger}$.

Figure 13 visually represents the cause of the slowdown. The source code of #commit: hasn't changed from Version 70 to Version 80. However, #commit: invokes #basicCommit:to: and the source code of this method has changed. It went from:

"Version 70"	"Version 80"
basicCommit: aMessage to: branchName commit parents self assert: modManager hasModifications. modManager processBlobs.	basicCommit: aMessage to: branchName commit parents modManager hasModifications ifFalse: []. modManager processBlobs.
 revision ifNotNil: [parents add: revision].	 revision ifNotNil: [parents add: (repository objectWithSignature: revision signature)]
revision := commit.	revision := repository objectWithSignature: commit signature.

Of the two modifications, the addition of the signature when committing is responsible of the slowdown. To confirm our findings, we contacted the author of GitFS to determine whether he agreed on our findings; he answered positively. This additional example further illustrates the efficiency of the behavioral evolution blueprint to track down the cause of a speedup or a slowdown.

Summary. Behavioral evolution blueprint is proposed as an effective and intuitive visualization that complements *Structural distribution blueprint* and *Behavioral distribution blueprint*, two visualizations of a unique profile snapshot. This third blueprint focuses on differencing profiles of two or more snapshots. A visualization always differentiates two profiles of the same application

 $^{^{\}ddagger} = (4255 - 2541) * 100/2541$

and benchmark, but for different software versions. Behavioral evolution blueprint helps tracking down the cause of a performance increase or decrease.

The effectiveness of behavioral evolution blueprint depends on the structural difference between the profiled version. Versions need to be "close enough" to give a useful meaning to the comparison. This point is discussed further in the following section (Section 5).

5. DISCUSSION

We discuss crucial points about the design of our blueprints.

Profiler. Profiling blueprints were first implemented for KAI. KAI is a profiler for the Pharo Smalltalk programming language. The graphical engine that renders profiling blueprint is not tied to KAI and Pharo. Once serialization and deserialization of a profiling has been agreed upon, then the blueprints may be generated for different profilers.

A number of requirements must be met by a profiler to produce profiling blueprints. First, it has to compute for each methods a number of metrics: number of different receivers per method, total execution time of a method and number of executions for each method (cf Section 2.3). Then, whether their return value is constant over multiple invocations (Section 2.4).

Gathering these information is not free. KAI execute twice the application to profile. The first time to gather using an execution sampling technique the call graph and the execution time for each method, and the second time to compute the method properties. Without being optimized, KAI produced satisfactory results for the situation in which it has been employed. The advances in the field of bytecode instrumentations and aspect-oriented techniques[§] are likely to be useful to reduce the cost of profiling.

Transformation verification. The yellow color used in the behavioral blueprint indicates possible locations for a memoization cache insertion. The cache is inserted by modifying the class definition, the method to be optimized and adding a method for clearing the cache, if necessary. There is no guaranty that such a transformation will not produce ripple effects, which may conduct the application to a complete different behavior.

The right semantic of all our code transformations have been validated using an extended set of unit tests. Before doing the transformation, we make sure that our tests are green. After the transformation, tests must remain green. Using unit testing as a reference for the right application behavior is not ideal since having a test coverage for *all* the execution is challenging [11]. However, having a robust set of unit tests has many advantages that go well beyond the semantics preservation of our code transformation.

Difficulty of profiling different version. A smooth integration of the Behavioral Evolution Blueprint (Section 4) in the Pharo programming environment is obtained by performing all the profiling on the same virtual machine. However, this has a number of drawback: Usage of sampling profiling may produce different execution times when repeating a profile over the same source code. As this problem is repeated over each version, comparing profiles gets more challenging. This problem is not unique to our approach: Mytkowicz *et al.* have shown that state-of-the-practice Java profilers such as xprof, hprof, jprofile and yourkit exhibit the same issues [12]. The usual workaround is to repeat the sampling process several times, and average the results of these execution. As future work, we are considering different metrics that could be used instead of time execution, such as message counting, or bytecode counting [13, 14]; these measures may prove more stable over different runs, compared to the time information obtained by sampling.

^{\$}http://www.eclipse.org/aspectj



Figure 14. Comparing distant versions.

Comparing distant versions. *Behavioral evolution blueprint* highlights differences between different profiles. This comparison makes sense only for profiles that are structurally comparable. By structurally, we refer to the application static structure, in terms of classes and methods. To illustrate this point, we obtained the evolution blueprint for the MetacelloBrowser application[¶] with 3 different versions (0.0, 1.1 and 1.41). Version 0.0 is a version that does not define any method or class. We use 0.0 as the extreme case for comparison.

Figure 14 shows that Version 1.41 defines the method #infos and 5 additional methods that are directly and indirectly called by #infos. These 6 methods are new in 1.41 since they are painted in yellow. In addition, 2 methods were redefined in 1.41 and are indirectly called by #infos (these are the small red methods). The call graph of #infos is almost made of yellow nodes.

In Version 1.1, the call graph rooted in MetacelloBrowser>>initialize is entirely painted in yellow since these methods do not exist in Version 0.0. In Version 1.41, #initialize is red, meaning that the method is slower and its source code has changed. The method #configurationSelection: is present in both versions. The cause of the slow behavior is due to the new version of #initialize.

The situation given in the figure clearly shows one limitation of the behavioral evolution blueprint. It is ineffective when the versions to compare are structurally "very" different. Whereas one can still refer to the cause of a slowdown in the evolution of the #initialize method, the call graph of #infos is pretty useless to determine what is the cause of its high execution time (#infos takes about 62% of the execution time).

From our personal experience, identifying what causes a slowdown in a call graph that has more than 30% of new methods is difficult at best. Naturally, this phenomena is highly dependent on the software analyzed. Note that the root method of the call graph has to be present in the previous version. In practice, the cause of a slowdown is easier to find with a small structural difference only.

Keeping track of returned values. The optimizations we realized stem from keeping track of values returned when invoking methods. The intuition that we exploited is that if a method always return the same computed value, then it may be worth adding a memoization mechanism to avoid redundancy. Our blueprint indicates such methods with the yellow color.

We analyzed the presence and relevance of these candidate methods. We took 5 representative applications of the Pharo ecosystem. These applications are available from the Pharo SqueakSource forge. For each of these applications, we run their associated unit tests and report our monitoring. Unit tests often describes typical execution scenarios [15], it therefore looks reasonable to base our analysis on the unit test execution.

[¶]http://metacellobrowser.dcc.uchile.cl

Application	# total methods	# constant methods
Mondrian	2098	39
Pier	3238	106
Grease	1046	12
Magritte	2056	66
XMLParser	1125	18

The first column gives the total mount of methods defined in the applications. The second column gives the amount of constant methods that return a value other than the receiver itself. We filtered out methods that are invoked less than 2 times and which perform less than 2 calls to avoid methods that simply do a delegation or return a primitive value. The amount of constant methods in these applications ranges from 1.1% and 3.2%.

We now focus on the case of Mondrian. We see that there are 39 constant methods that are potential candidates for a memoization mechanism. Each of them has been manually reviewed. We can classify each of these 39 methods along 3 sets: *caching*, *lazy initialization* and *candidate methods*.

Caching. Keeping satisfactory performances in Mondrian is a major concern for its development. Mondrian contains a number of sophisticated mechanisms to cache metric values. The method #colorFor: and #computeColorFor: is a typical example:

```
MOLineShape>>colorFor: anElement

^anElement

cachedNamed: #cachecolorFor:

ifAbsentInitializeWith: [ self computeColorFor: anElement ]
```

```
MOLineShape>>computeColorFor: anElement 
^color moValue: anElement model
```

Among the 39 methods, 3 groups of 2 methods follow the pattern *#metricFor:* and *#computeMetricFor:*. These 6 methods are therefore false positive since they already implement a memoization mechanism.

Lazy initialization. Delaying the object creation is often employed in Mondrian. The method #fontCache and #default are representative:

```
MOBoundedShape>>fontCache

^fontCache ifNil: [fontCache := Dictionary new]
```

MONodeShape>>default ^Default ifNil: [Default := MORectangleShape new]

11 methods use lazy initialization. These methods are considered as false positive.

Candidate methods. From the 39 methods, only 22 methods return the same value at each invocation and does not use a cache or a lazy initialization. 4 out of these 22 are due to incomplete testing. For example, the method #selectBoxBounds is defined as:

```
MORoot>>selectBoxBounds

"Return the selection box, note that the two corners cannot be nil"

self

assert: [selectionBoxCorner1 notNil and: [selectionBoxCorner2 notNil]]

description: 'Corners cannot be nil'.

^Rectangle encompassing: {selectionBoxCorner1 . selectionBoxCorner2 }.
```

This method is tested just once. This method is an example of a poor testing.

After a scrutinization, only 18 out of the 39 methods are considered candidates for implementing the memoization. These 18 methods represent less than 1% of the total amount of method in Mondrian. None of these methods is responsible for a slowdown, as we fixed them along the paper.

From numerical values to visual patterns. Outputs of traditional profilers make heavy use of numerical values to describe the execution of the profiled software. The execution is expressed in terms of percentage of the global execution time for each individual method. Profiling blueprints are a graphical representation of these metrics. By moving from a textual to graphical representation of a profiling, we gained the ability to compare many different metrics at the same time. Numerical comparison has been traded off for visual pattern identification. Instead of seeing that #applyLayout takes 53% of the total time execution and #bounds 40%, we see that the square representing #applyLayout is "slightly" larger than the one representing #bounds. Even though a visualization conveys less accurate values since we cannot precisely quantify distances and color intensity, we gained the ability to easily perform comparisons on a large amount of entities at once, allowing us to pinpoint the cause of the performance issues slower.

Scalability. The experiments we used for this work were realized on medium-sized applications (Mondrian totals 2 000 methods and 20 000 lines of codes approximately). The visualization were produced on a screen with a resolution of 1440×900 pixels. The largest execution was three screens large. Zooming out is hardly a practical solution: classes and methods needs to be significantly reduced to make more than 200 classes fit in one screen. Bertin [16] assessed that a good practice is to offer a visualization that can be grasped at one glance, without need to scrolling or moving around. The layout we have adopted are optimal for medium-sized applications.

Note that, strictly speaking, our blueprints are not the only visualization which suffer from scalability. YourKit and JProfiler output their results using a textual table or simple graphics, two visualizations fragile to scalability.

Multi-threading. Thread activity is not represented in our profiling blueprints. The experience on Mondrian threads (Section 3.2) has been realized by solely profiling the code executed by the UI thread. The logic thread was not started. Profiling concurrent and multi-threaded applications has not been considered in our work. The problems solved we solved in our experiment did not exhibit such a situation.

6. RELATED WORK

Our related work review discusses first the various dynamic information visualizations that have been proposed, before reviewing the various approaches that have been proposed to compare executions of two different versions of a program.

6.1. Visualization of Dynamic Information

Profiling capabilities have been integrated in IDEs such as the NetBeans Profiler^{||} and Eclipse's Tracing and Profiling Project (TPTP)^{**}. The NetBeans Profiler uses JFluid [17], which offers a Calling Context Tree (CCT) [18] augmented with the accumulated execution time for individual methods. The CCT is visualized as an expandable tree, where calling contexts are sorted by their execution time and can be expanded (respectively collapsed) in order to show (or hide) callees. However, as CCTs for real-world applications are often large, comprising up to some million nodes, an expandable tree representation makes it difficult to detect hotspots in deep calling contexts.

The Calling Context Ring Chart (CCRC) [19, 20] is a CCT visualization that eases the exploration of large trees. Like the Sunburst visualization [21], CCRC uses a circular layout. Callee methods are represented in ring segments surrounding the caller's ring segment. In order to reveal hot calling contexts, the ring segments can be sized according to a chosen dynamic metric. Recently, CCRC has been integrated into the Senseo plugin for Eclipse [22], which enriches Eclipse's static source

http://profiler.netbeans.org/

^{**}http://www.eclipse.org/tptp/performance/

Execution traces may be used to analyze dynamic program behavior. Execution traces are logged events, such as method entry and exit, or object allocation. However, the resulting amount of data can be excessive. In Deelen *et al.* [23] execution traces are visualized with nodes representing classes and edges representing method calls. Node size and edge thickness are mapped to properties (e.g., number of method invocations). A time range can be selected in order to limit the data to be visualized. Another approach to visualizing execution traces has been introduced in Holten *et al.* [24]. It uses the concept of hierarchical edge bundles [25], where similar edges are put together to improve the visualization of larger traces. Execution traces allow keeping calls in sequences and selecting a precise time interval to be visualized, which helps understanding a particular phase in the execution of a program. Blueprint profiling offers a global map of the complete execution without focusing on sequentiality in time. However, they offer hints about the behavior of individual methods that help to solve a class of optimization problem, namely introducing caches.

Tree-maps [26] visualize hierarchical structures. Nodes are represented as rectangular areas sized proportionally to a metric. Tree-maps have been used to visualize profiling data. For instance, in [27] the authors present KCacheGrind, a front end to a simulator-based cache profiling tool, using a combination of tree-maps and call graphs to visualize the data. Our blueprint use polymetric view to render data. A tree-map solves a problem in a different way that a polymetric view would solve it. A polymetric view enables one to compare several different metrics, whereas a tree-map is dedicated to showing a single metric (besides color) in a compact space.

Greevy *et al.* enhanced polymetric views with a third dimension in order to visualize the runtime behavior of systems [28]. They focus on the execution of individual features, use the third dimension to overlay dynamic information about instanciation and message sending over a polymetric view. Their system allows one to replay each of the steps of the trace, and is geared towards program comprehension, while our approach focuses on performance profiling, and provides a comprehensive view of performance metrics.

Sevitsky *et al.* present an information visualization tool specialized in the performance analysis of Java programs, Jinsight EX [29]. JInsight EX uses execution slices to narrow down the data to analyse down to a more manageable size, and proposes visualizations centered on the slices. Previous work by De Pauw *et al.* [30, 31] also abstract dynamic information to the instance, method and class levels, proposing visualizations such as instance-level histograms, and functions-instances or inter and intra-classes call matrices. We however abstract away more information as we show metrics instead of the more execution-event based approach employed in these works.

Reiss and Eddon adopt an approach radically different than ours, as they visualize extremely finegrained information of programs as they are executing [32]. JIVE visualizes for instance the state of the java heap, file I/O operations, or informations on which class is used in which thread of the program. Its successor, JOVE, further maps the information to source code locations that are being executed at any given moment [33].

6.2. Comparison of Versions

Several approaches exist that compare the execution of two versions of a program.

Zhang and Gupta [34] present an approach to match two programs that behave similarly, although they appear to be statically different. The approach matches the execution at the level of binary instructions, and is geared at scenarios such as debugging—for instance matching of an optimized and an unoptimized version of a program—, or piracy detection—detecting similar behavior despite obfuscation—, and not explicitly performance. The authors focus on the performance and accuracy of the matching algorithm. Nagajaran *et al.* later extended the approach, in order to support more aggressive transformations by working at the control flow level rather than the instruction level [35].

Zhuang *et al.* presented a framework for differencing execution profiles named PerfDiff [36]. The approach is based on matching CCTs. The algorithm identifies variations in the layout of the trees or the weight—the weight being a performance metric in their case—of the nodes. The difference algorithm is applied to selected benchmarks of programs, running under different platforms. The

authors report on the accuracy of the algorithm. Mostafa and Krintz propose a similar approach, aimed this time at tracking performance across different revisions of the same program [37]. They also employ CCTs, and perform an empirical evaluation of their matching algorithm.

We see that most approaches aiming at matching different executions of a program have primarily been evaluated in terms of their matching accuracy. Our focus is more on how to exploit this information: We presented visualizations summing up the differences between two versions of a given program. Since the dynamic information we collect is higher-level than CCTs (as it is aggregated at the level of methods and classes), we argue that the accuracy of the matching algorithm is not as critical, hence our focus in exploiting the information.

7. CONCLUSION

In this article we presented three visualizations helping developers to identify and remove performance bottlenecks. Providing visualizations that are intuitive and easy to use is our primary goal. Our graphical blueprints follow simple principles such as "big nodes are slow methods", "gray nodes are methods likely to have side-effects", "yellow nodes remain constant on return values". Our visualizations helped us to significantly improve Mondrian, a visualization engine. We described a number of optimizations we realized. The last version of Mondrian contains an improved version of the #applyLayout method, thus mitigating the bottleneck caused by this method, and other optimizations.

We also introduced a visualization aimed at the retrospective evaluation of the evolution of the performance of a system over time. The behavioral evolution blueprint allowed us to pinpoint changes to the source code that caused performance degradation in two instances, for the Mondrian and GitFS projects; the developer of GitFS reviewed and validated our findings.

A number of conclusions may be drawn from the experiment described in this article. First, bottleneck identification and removal are significantly easier when side-effects and constant return values are localized. Second, an extensive set of unit tests remains essential to assess whether a candidate optimization can be applied without changing the behavior of the system. Third, one must pay attention that further changes to a program may render previous optimizations less useful than they used to be, and keep an eye over this phenomenon over time.

As future work, we plan to focus on architectural views by adopting coarser grain than methods and classes. We also plan to further our work on the evolution of performance by proposing visualizations that handle more versions at once.

Acknowledgment. We gratefully thank Max Leske for his feedback on our finding related to GitFS.

REFERENCES

- 1. Tuduce I, Majo Z, Gauch A, Chen B, Gross TR. Asymmetries in multi-core systems or why we need better performance measurement units. *Proceedings of the Exascale Evaluation and Research Techniques Workshop (EXERT)*, 2010.
- Cornea B, Bourgeois J. Performance prediction of distributed applications using block benchmarking methods. *PDP'11, 19-th Int. Euromicro Conf. on Parallel, Distributed and Network-Based Processing*, IEEE Computer Society Press: Ayia Napa, Cyprus, 2011.
- Bergel A, Robbes R, Binder W. Visualizing dynamic metrics with profiling blueprints. *Objects, Models, Components, Patterns, Lecture Notes in Computer Science*, vol. 6141, Vitek J (ed.), Springer Berlin / Heidelberg, 2010; 291–309, doi:10.1007/978-3-642-13953-6_16.
- 4. Meyer M, Gîrba T, Lungu M. Mondrian: An agile visualization framework. ACM Symposium on Software Visualization (SoftVis'06), ACM Press: New York, NY, USA, 2006; 135–144, doi:10.1145/1148493.1148513. URL http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf.
- Benbasat I, Goldstein DK, Mead M. The case research strategy in studies of information systems. *MIS Q.* Sep 1987; 11:369–386, doi:10.2307/248684. URL http://portal.acm.org/citation.cfm?id=35194.35201.
- 6. Reiss S. Visualizing the java heap to detect memory problems. *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on, 2009; 73–80, doi:10.1109/VISSOF.2009. 5336418.*

- 7. Lanza M, Ducasse S. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)* Sep 2003; **29**(9):782–795, doi:10.1109/TSE.2003.1232284. URL http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf.
- 8. Gîrba T, Lanza M. Visualizing and characterizing the evolution of class hierarchies. WOOR 2004 (5th ECOOP Workshop on Object-Oriented Reengineering), 2004. URL http://scg.unibe.ch/archive/papers/Girb04aHierarchiesEvolution.pdf.
- Ducasse S, Lanza M, Bertuli R. High-level polymetric views of condensed run-time information. Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04), IEEE Computer Society Press: Los Alamitos CA, 2004; 309–318, doi:10.1109/CSMR.2004.1281433. URL http://scg.unibe.ch/ archive/papers/Duca04aRuntimePolymetricViews.pdf.
- Gîrba T, Ducasse S. Modeling history to analyze software evolution. Journal of Software Maintenance and Evolution: Research and Practice May 2006; 18(3):207-236, doi:10.1002/smr.325. URL http://doi.wiley. com/10.1002/smr.325.
- Yang Q, Li JJ, Weiss DM. A Survey of Coverage-Based Testing Tools. *The Computer Journal* 2009; 52(5):589-597, doi:10.1093/comjnl/bxm021. URL http://comjnl.oxfordjournals.org/content/ 52/5/589.abstract.
- Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF. Evaluating the accuracy of java profilers. Proceedings of the 31st conference on Programming language design and implementation, PLDI '10, ACM: New York, NY, USA, 2010; 187–197, doi:10.1145/1806596.1806618. URL http://doi.acm.org/10.1145/1806596. 1806618.
- Binder W. Portable and accurate sampling profiling for Java. Software: Practice and Experience 2006; 36(6):615–650.
- Binder W, Hulaas J, Moret P, Villazón A. Platform-independent profiling in a virtual execution environment. Software: Practice and Experience 2009; 39(1):47–79.
- 15. Martin RC. Agile Software Development. Principles, Patterns, and Practices. Prentice-Hall, 2002.
- 16. Bertin J. Graphische Semiologie. Walter de Gruyter, 1974.
- 17. Dmitriev M. Profiling Java applications using code hotswapping and dynamic call graph revelation. WOSP 2004: Proceedings of the Fourth International Workshop on Software and Performance, ACM Press, 2004; 139–150.
- Ammons G, Ball T, Larus JR. Exploiting hardware performance counters with flow and context sensitive profiling. PLDI 1997: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, ACM Press, 1997; 85–96, doi:http://doi.acm.org/10.1145/258915.258924.
- Moret P, Binder W, Ansaloni D, Villazón A. Visualizing Calling Context Profiles with Ring Charts. VISSOFT 2009: Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, IEEE Computer Society: Edmonton, Alberta, Canada, 2009; 33–36.
- Moret P, Binder W, Villazón A, Ansaloni D, Heydarnoori A. Visualizing and exploring profiles with calling context ring charts. Software: Practice and Experience 2010; 40:825–847.
- Stasko J. An evaluation of space-filling information visualizations for depicting hierarchical structures. Int. J. Hum.-Comput. Stud. 2000; 53(5):663–694, doi:http://dx.doi.org/10.1006/ijhc.2000.0420.
- Röthlisberger D, Härry M, Villazón A, Ansaloni D, Binder W, Nierstrasz O, Moret P. Augmenting Static Source Views in IDEs with Dynamic Metrics. *ICSM 2009: Proceedings of the 25th IEEE International Conference on* Software Maintenance, IEEE Computer Society: Edmonton, Alberta, Canada, 2009; 253–262.
- Deelen P, van Ham F, Huizing C, van de Watering H. Visualization of dynamic program aspects. VISSOFT 2007: Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007; 39–46.
- Holten D, Cornelissen B, van Wijk JJ. Trace visualization using hierarchical edge bundles and massive sequence views. VISSOFT 2007: Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007; 47–54.
- Holten D. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions* on Visualization and Computer Graphics Sept-Oct 2006; 12(5):741–748.
- Johnson B, Shneiderman B. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. VIS 1991: Proceedings of the 2nd conference on Visualization, IEEE Computer Society Press, 1991; 284–291.
- Weidendorfer J, Kowarschik M, Trinitis C. A tool suite for simulation based analysis of memory access behavior. ICCS 2004: Proceedings of the 4th International Conference on Computational Science, LNCS, vol. 3038, Springer, 2004; 440–447.
- Greevy O, Lanza M, Wysseier C. Visualizing live software systems in 3D. Proceedings of SoftVis 2006 (ACM Symposium on Software Visualization), 2006, doi:10.1145/1148493.1148501. URL http://scg.unibe.ch/ archive/papers/Gree06aTraceCrawlerSoftVis2006.pdf.
- Sevitsky G, Pauw WD, Konuru R. An information exploration tool for performance analysis of java programs. TOOLS Europe '01: Proceedings of the 38th International Conference on Technology of Object-Oriented Languages and Systems, Components for Mobile Computing, 2001; 85–101, doi:10.1109/TOOLS.2001.911758.
- De Pauw W, Helm R, Kimelman D, Vlissides J. Visualizing the behavior of object-oriented systems. Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93), 1993; 326–337, doi:10.1145/165854.165919.
- Pauw WD, Vlissides JM. Visualizing object-oriented programs with jinsight. Workshop ion on Object-Oriented Technology, ECOOP '98, Springer-Verlag: London, UK, 1998; 541–542. URL http://portal.acm.org/ citation.cfm?id=646778.704665.
- 32. Reiss SP. JOVE: Java as it happens. Proceedings of SoftVis 2005(ACM Symposium on Software Visualization), 2005; 115–124.
- 33. Reiss SP, Renieris M. Jove: java as it happens. SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization, ACM: New York, NY, USA, 2005; 115–124, doi:10.1145/1056018.1056034.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using speauth.cls

- 34. Zhang X, Gupta R. Matching execution histories of program versions. *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM: New York, NY, USA, 2005; 197–206, doi:10.1145/1081706.1081738.
- Nagarajan V, Gupta R, Zhang X, Madou M, De Sutter B. Matching control flow of program versions. Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'07), 2007; 84 –93, doi:10.1109/ICSM. 2007.4362621.
- 36. Zhuang X, Kim S, Serrano Mi, Choi JD. Perfdiff: a framework for performance difference analysis in a virtual machine environment. CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, ACM: New York, NY, USA, 2008; 4–13, doi:10.1145/1356058.1356060.
- Mostafa N, Krintz C. Tracking performance across software revisions. PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, ACM: New York, NY, USA, 2009; 162–171, doi:10.1145/1596655.1596682.