Visualizing Dynamic Metrics with Profiling Blueprints

Alexandre Bergel[1], Romain Robbes[1], and Walter Binder[2]

[1] Pleiad Lab, DCC, University of Chile, Santiago, Chile
http://bergel.eu http://www.dcc.uchile.cl/~rrobbes

[2] University of Lugano, Switzerland
http://www.inf.usi.ch/faculty/binder

In Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE'10), LNCS Springer Verlag

Abstract. 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 visual and detailed runtime information to identify and eliminate computation redundancy. We provide two 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. These two 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.

1 Introduction

Even though computing resources are abundant, execution optimization through code profiling remains an important software development activity. A CPU time profiler is a crucial tool to identify bottlenecks – program elements that take a large part of the execution time. Today, it is inconceivable to ship a programming environment without a code profiler included or provided by a third party.

However, when we retrospectively look at the history of code profiler tools, we see that tool usability and profiling overhead reduction have steadily improved, but that the set of offered abstractions has remained constant. For instance, gprof, which appeared in 1982, offers a number of textual reports focussed on "how much time was spent executing directly in each function" and call graphs[1]. JProfiler essentially produces the same output, using a graphical rendering instead of a textual one[2]. Most of the research conducted in the field of code profiling focus on reducing the overhead triggered by the code instrumentation and

[1] http://sourceware.org/binutils/docs/gprof/Output.html#Output
[2] http://www.ej-technologies.com/products/jprofiler/screenshots.html

observation. On the other hand, the abstractions used to profile object-oriented applications are very close to the ones for procedural applications.

The contribution of this paper is to apply some visualizations that have been previously used in static software analysis to display dynamic metric for profiling purposes. We propose a visual mechanism 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 propose 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. The work presented in this paper aims at complementing existing profilers with new visualizations that help specific optimization tasks.

The results presented in this paper were realizing using Pharo[3], an open-source Smalltalk-dialect programming language. Nothing in the visualizations we propose prevents one from using them in a different setting.

We apply our techniques to the visualization framework Mondrian[4] [MGL06], our running example. We first describe our blueprints (Section 2). Subsequently, we identify and implement opportunities for optimization in Mondrian (Section 3). We then review related work (Section 4) and conclude (Section 5).

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 boolean property (*e.g.,* a gray 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 often-used methods faster and (ii) call slow methods less often. The metrics we adopted in this paper help finding methods

[3] http://www.pharo-project.org/home
[4] http://www.moosetechnology.org/tools/mondrian

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

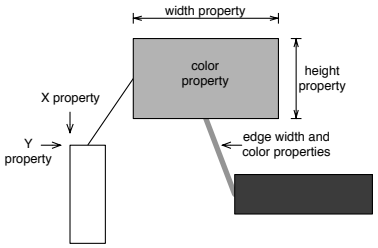

Fig. 1. Principle of polymetric view.

The blueprints we propose are graphically rendered as *polymetric views* [LD03]. 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.
- *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 [GL04] 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 [DLB04] (*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 1 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.

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)
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 1. Specification of the structural distribution blueprint

Example. Troughout this paper, we use the graph visualization framework Mondrian as a case study. The blueprints described in this paper 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:

```
ProfilingPackageSpy
    viewProfiling: [
        | view |
        view := MOViewRenderer new.
```

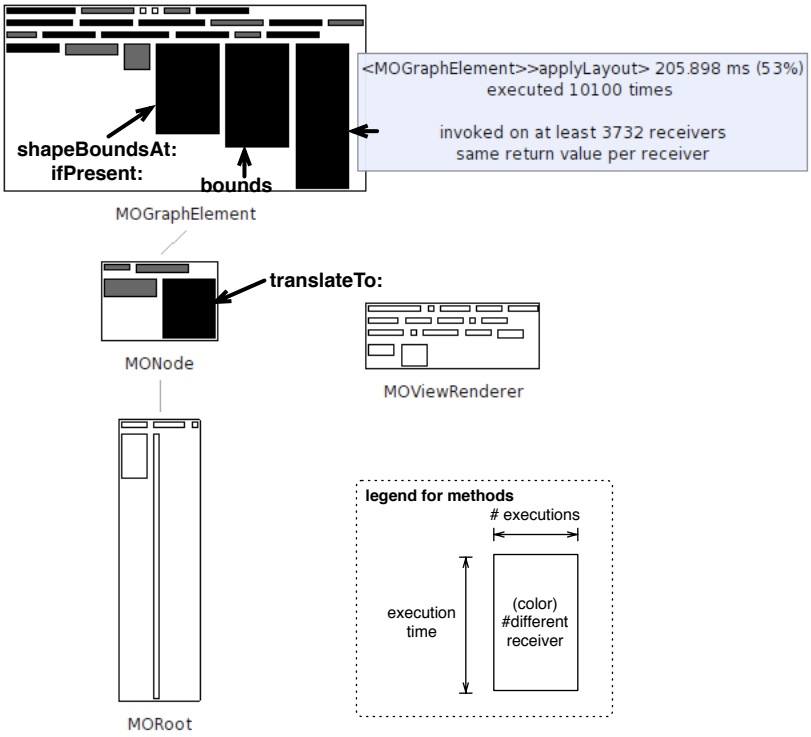

Fig. 2. Example of a structural blueprint

```
        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 CPU time. 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>> shapeBoundsAt: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 solely 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 2 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.

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.

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.

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 execution (logarithmic scale)
Example	Figure 3

Table 2. Specification of the behavioral distribution blueprint

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 this 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

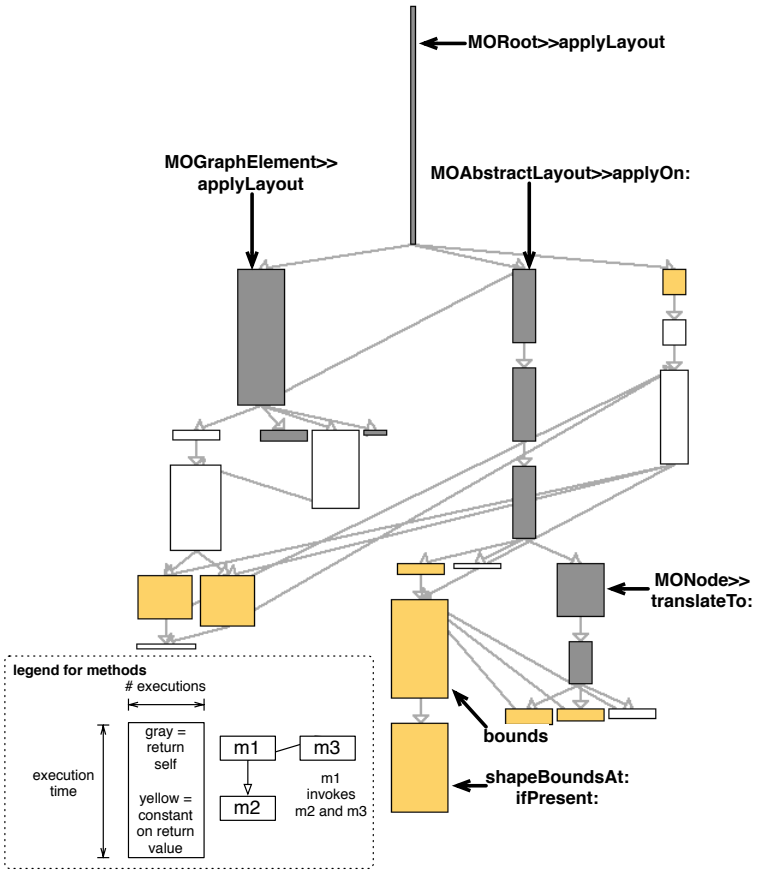

Fig. 3. Example of a behavioral blueprint

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 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 since it is painted in yellow. This method is involved in a rich invocation graph (presented in Figure 3). In general, 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.

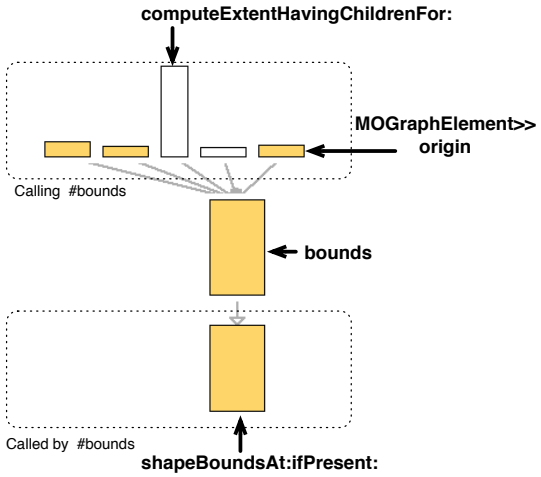

Fig. 4. Detailed view of MOGraphElement>> bounds

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[5], 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 | ^b], which has the effect of exiting the method, is therefore always executed in the considered example.

We first thought that the last tree 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 some scenarios although they are 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 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.

[5] http://en.wikipedia.org/wiki/Memoization

```
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 left blueprint has been produced before upgrading the method MOGraphElement>> bounds. Figure 2 is a part of it. The right one has been produced after upgrading bounds as described above. Many places are impacted. We annotated the figure with the most significant changes:

- 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

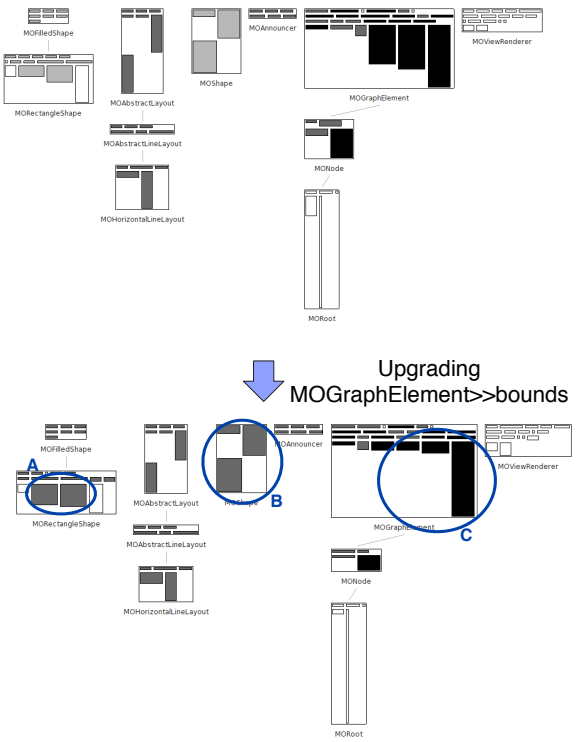

Fig. 5. Upgrading bounds has a global structural impact

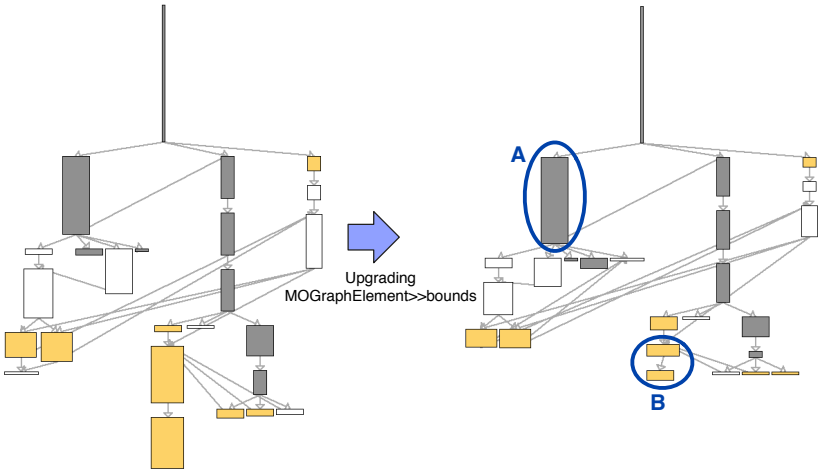

Fig. 6. Upgrading bounds has a global behavioral impact

events (e.g., window refreshes). Mondrian is no exception. The blueprints presented earlier focused on profiling the application logic.

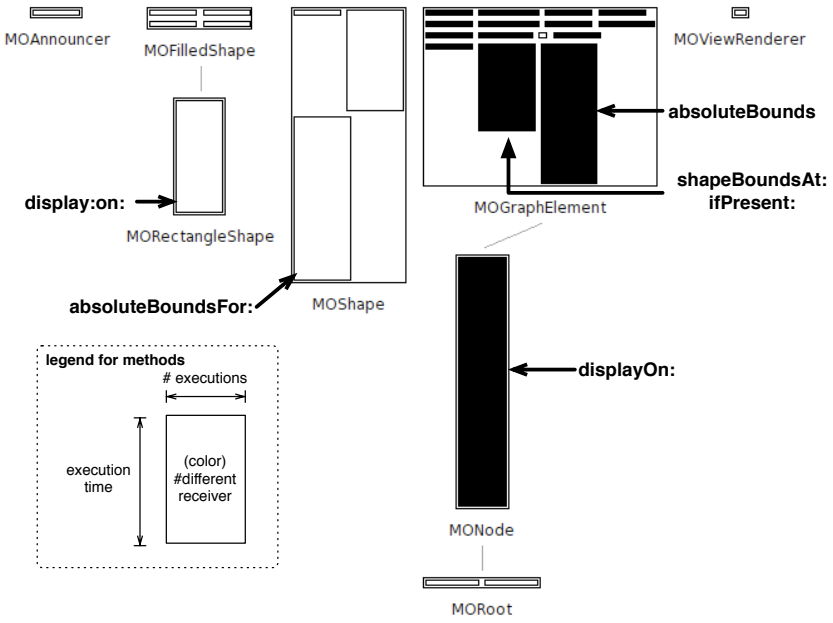

Fig. 7. Profiling of the UI thread in Mondrian

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 lead 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:

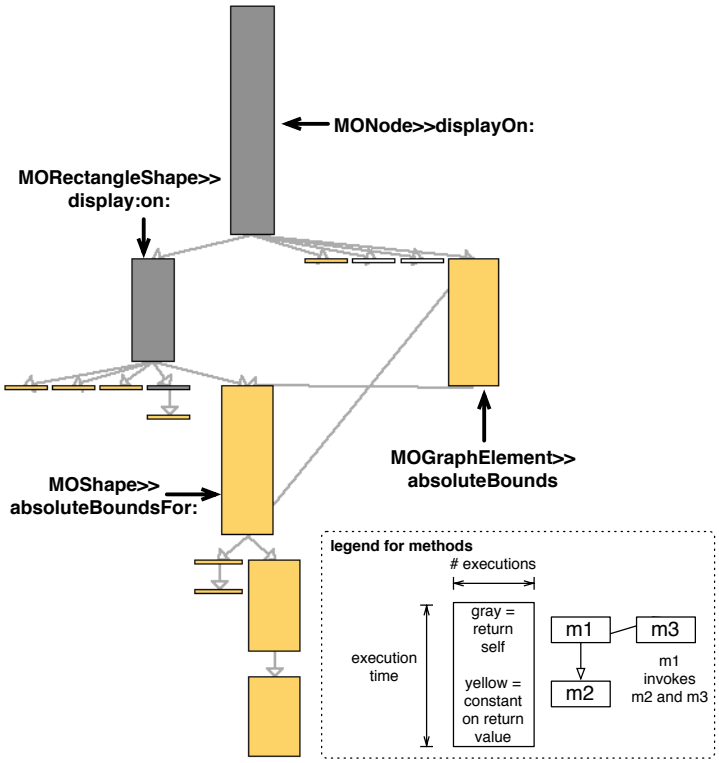

Fig. 8. Profiling of the UI thread in Mondrian

```
MOGraphElement>> absoluteBounds
  "Answer the bounds in absolute terms (relative to the entire Canvas, not just the parent)."
  absoluteBoundsCache ifNotNil: [ ^absoluteBoundsCache ].
  ^absoluteBoundsCache := self shape absoluteBoundsFor: self
```

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 time became so small, that it 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[6]. The improvement of ab-

[6] The source code is available at: http://www.squeaksource.com/Mondrian.html

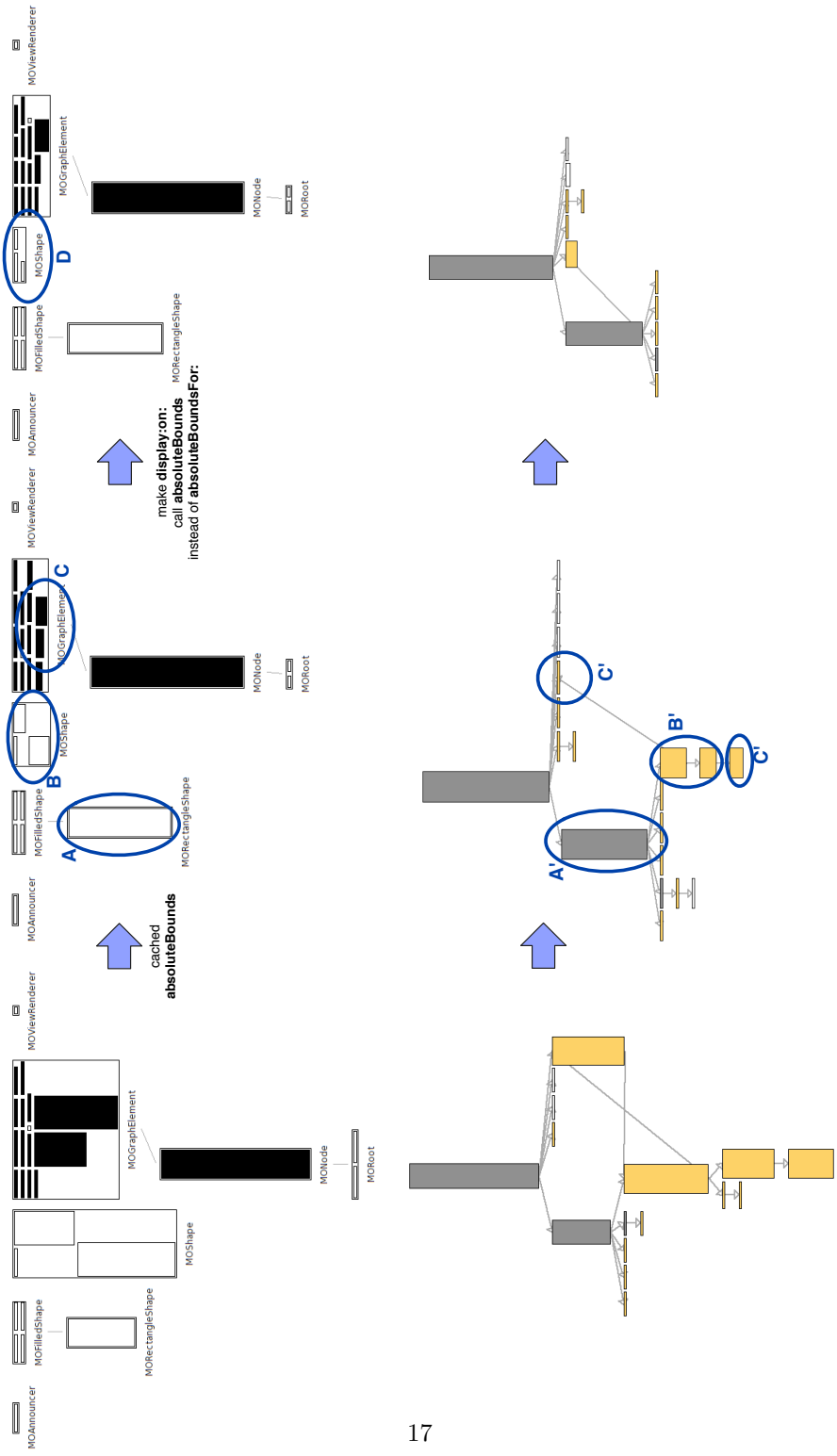

Fig. 9. Profiling of the UI thread in Mondrian

soluteBounds and display:on: may be found in the version 352 of Mondrian. The complete experiment lead 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 easily or not be realized heavily depends on a wide range of parameters (*e.g.,* algorithm, architecture, data structure).

4 Related Work

Profiling capabilities have been integrated in IDEs such as the NetBeans Profiler[7] and Eclipse's Tracing and Profiling Project (TPTP)[8]. The NetBeans Profiler uses JFluid [Dmi04], which offers a Calling Context Tree (CCT) [ABL97] 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) [MBAV09] is a CCT visualization that eases the exploration of large trees. Like the Sunburst visualization [Sta00], 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 [RHV+09], which enriches Eclipse's static source views with several dynamic metrics. Our blueprints have a different focus, since global information is shown instead of providing a line-of-code granularity.

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.* [DvHHvdW07] 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.* [HCvW07]. It uses the concept of hierarchical edge bundles [Hol06], 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 understand a particular phase in the execution of a program. Blueprint profiling

[7] http://profiler.netbeans.org/
[8] http://www.eclipse.org/tptp/performance/

offers a global map of the complete execution without focusing on sequentiality in time. On the other, they offer hints about the behavior of individual methods that help to solve a class of optimization problem, namely introducing caches.

Tree-maps [JS91] 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 [WKT04] 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 enables one to compare several different metrics, whereas a tree-map is dedicated to showing a single metric (besides color) in a compact space.

5 Conclusion

In this paper we presented two 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. For space reason, we couldn't describe all the optimization. The last version of Mondrian contains an improved version of the applyLayout method, thus mitigating the bottleneck caused by this method. This improvement was recently publicly announced[9].

A number of conclusions may be drawn from the experiment described in this paper. 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.

As future work, we plan to focus on architectural views by adopting coarser grain than methods and classes.

We used our blueprint visualizations on a number of case studies not described in this paper: Glamour and Moose, and O2[10]. Our visualizations and profiler are available in Pharo[11] under the MIT license.

References

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the conference on Programming language design and implementation*, pages 85–96. ACM Press, 1997.

[9] http://www.iam.unibe.ch/pipermail/moose-dev/2010-January/003781.html
[10] http://www.moosetechnology.org/tools/, http://www.squeaksource.com/O2.html
[11] http://www.squeaksource.com/Spy.html

[DLB04] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.

[Dmi04] Mikhail Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150. ACM Press, 2004.

[DvHHvdW07] P. Deelen, F. van Ham, Cornells Huizing, and H. van de Watering. Visualization of dynamic program aspects. In *VISSOFT 2007: 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 39–46, June 2007.

[GL04] Tudor Gîrba and Michele Lanza. Visualizing and characterizing the evolution of class hierarchies. In *WOOR 2004 (5th ECOOP Workshop on Object-Oriented Reengineering)*, 2004.

[HCvW07] D. Holten, B. Cornelissen, and J.J. van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. In *VISSOFT 2007: 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 47–54, June 2007.

[Hol06] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, Sept.-Oct. 2006.

[JS91] Brian Johnson and Ben Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 284–291. IEEE Computer Society Press, 1991.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[MBAV09] Philippe Moret, Walter Binder, Danilo Ansaloni, and Alex Villazón. Visualizing Calling Context Profiles with Ring Charts. In *VISSOFT 2009: 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 33–36.

[MGL06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006.

[RHV+09] David Röthlisberger, Marcel Härry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Augmenting Static Source Views in IDEs with Dynamic Metrics. In *ICSM '09: Proceedings of the 2009 IEEE International Conference on Software Maintenance*, pages 253–262.

[Sta00] John Stasko. An evaluation of space-filling information visualizations for depicting hierarchical structures. *Int. J. Hum.-Comput. Stud.*, 53(5):663–694, 2000.

[WKT04] Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. A tool suite for simulation based analysis of memory access behavior. In *ICCS 2004: 4th International Conference on Computational Science*, volume 3038 of *LNCS*, pages 440–447.