Visual Patterns with Profiling Blueprint

Alexandre Bergel[1], Vanessa Peña[1], Chris Thorgrimsson[2], Chung Ho Huang[2]

[1]PLEIAD Lab, Department of Computer Science (DCC), University of Chile
Object Profile, Chile

[2]Lam Research, USA

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.6 [**Software Engineering**]: Programming Environments—*integrated environments, interactive environments*; D.2.8 [**Metrics**]: Performance measures

Keywords

Performance, Visualization, Pharo

1. PROFILING BLUEPRINT

Profiling blueprint is a visual representation of software execution [1]. As most profiling reports, profiling blueprint offers a post-mortem report of an execution. Such a profile blueprint aims at rapidly identifying software components that poorly perform or are considered to be a bottleneck.

An example of such a blueprint is given in Figure 1. The figure depicts a call graph obtained from the execution of a benchmark. The application we are using here as the running example is Roassal, an agile visualization engine[1].

Methods. Each box visually represents a method. The height of a box is proportional to the time spent in the method. Tall methods are therefore methods in which the virtual machine spent a significant amount of time executing them. Methods marked with `A`, `B`, `C`, `D` and `F` are time consuming.

The width of a method indicates the number of times the method is executed. Method `A`, `B`, `C`, `D` are very thin, meaning that these methods are executed a few times. Large methods such as `E`, `F` and `G` are executed many times. The exact values of the time spent and the amount of executions are given by a tool tip text when the mouse cursor is hovered over a method.

The method height is linear to the total execution time of the method. It therefore includes the time spent in the

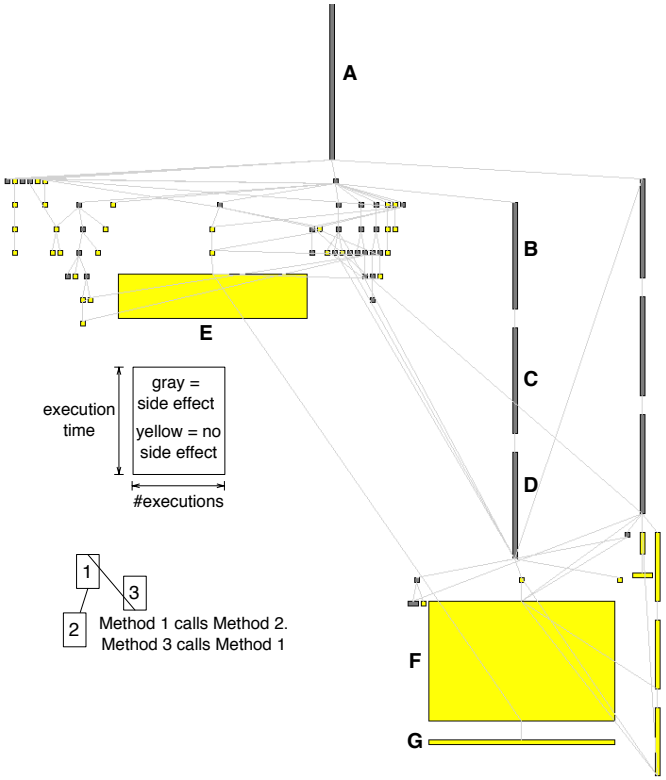

Figure 1: Example of a Blueprint Execution

[1]`http://objectprofile.com/roassal-home.html`

Dyla '13, Montpellier, France

called methods. The width uses a logarithmic scale instead of a linear one to cope with large variations.

Method invocations. The blueprint visualizes the execution control flow along the method call graph. Edges indicate invocations between two methods: a calling method is located above the methods it calls as long as the control flow is a tree. It frequently happens that the control flow forms a graph, in that case the location of edge's extremities in a method indicates the calling method and called method, as shown in Figure 1. Method `F` calls `E`. The bottom of a calling method box is always linked to the top of a called method box. Additionally, the layout-construction algorithm tries to place the calling methods above the called ones

Mutation. An important aspect of our blueprint is to indicate the presence of state mutation, resulting from side-effect. The color of the method indicates whether or not the object receiver or an argument has been modified by the method.

A method that mutates at least one variable in the object receiver or in the method arguments is shaded in gray. A method that does not modify the object is shaded in yellow.

For example, the source code of the method `B` is

```
ROElement>>addAll: els
  els do: [:el | self add: el ]
```

The method `addAll:` does not directly performs a mutation, however, it invokes `add:`, marked with a `C` in the figure, which itself adds an element into a collection held by the object that receives the method call `addAll:`.

The methods `E` and `F` are visually represented as large and tall yellow boxes, meaning that these two methods take a significant amount of CPU time, are executed many times, and do not modify the receiver and arguments. These methods are candidates for being cached using a memorization.

Implementation. The profiling blueprint is implemented for the Pharo[2] and VisualWorks[3] programming languages, two Smalltalk dialects. The runtime information is obtained using the Spy profiling framework [2]. Spy allows one to easily define a code execution profilers.

Visualization of the profile is carried out using Roassal[4], an agile visualization engine. Roassal offers facilities to easily define and represent polymetric views [3] such as the profiling blueprint.

2. EXECUTION PATTERNS

2.1 Time distribution

One strength of a visual representation is to let a human observer identify visual patterns [4]. We have discriminated two types of patterns related to the time distribution and the mutation.

Figure 2 gives six visual patterns related to time distribution that often occurs in profiling blueprint.

Pattern A - sequential calls. The method `a` directly calls a number of methods, 3 example methods are given, `b`, `c` and `d`. The height of `a` is the sum of the height of `b`, `c` and `d`. One can naturally deduce that the three calls to `b`, `c`, `d` are likely to be sequential. The methods `b`, `c`, `d` have the same width as `a`. This means that these calls are not contained in a loop.

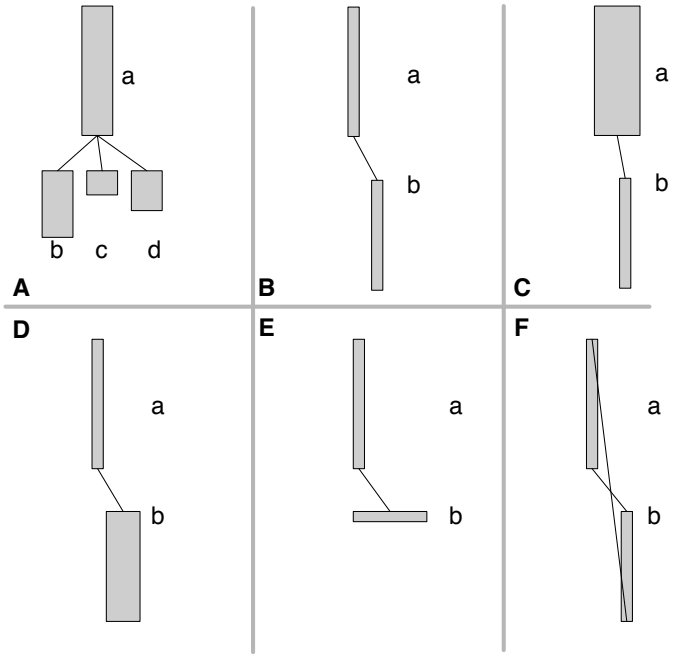

Figure 2: Visual patterns

The profiling blueprint orders the calls according to the sequence occurring at execution. The method `b` is called before `c`, itself called before `d`. However, this is given as a mere indication that may not reflect the actual execution, for example if the order is not always constant. The source code of `a` is accessible via a tool tip to assert this.

Pattern B - indirection. The method `a` directly calls the method `b` and the visual aspect of `b` is the same as `a`. Method `a` is simply calling `b` without doing any significant additional computation.

Pattern C - sporadic execution. The method `a` is invoked many times, shown from the width of its box. The method `b` is invoked less than `a`. The call to `b` is realized in infrequent situations.

Pattern D - multiple execution. The method `b` is executed several times, and the height of `b` is the same as `a`. The call of `b` is likely to be contained within a loop in `a`. This pattern is the opposite of Pattern C.

This pattern is illustrated by the methods `D` and `F` in Figure 1.

Pattern E - immediate value. The method `b` is executed many more times than `a`, which is indicated from the difference of the width. However, it does not consume much execution time since it is horizontally thin. The method `b` is likely to be computationally cheap like returning an immediate value (*e.g.,* being constant or a variable accessor).

Pattern F - recursion. Recursive calls are indicated with

[2] `http://pharo-project.org`
[3] `http://www.cincomsmalltalk.com/main/products/visualworks/`
[4] `http://objectprofile.com/#/pages/products/roassal/overview.html`

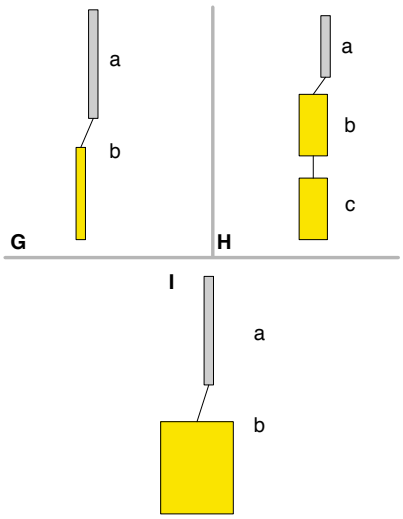

Figure 3: Visual patterns

crossing edges. The method `a` calls `b`, which itself calls `a`.

2.2 Object mutation

Three different patterns may involve methods that do not modify the object receiver or the arguments (Figure 3).

Pattern G - non altering delegation. The method `a` calls method `b`. Both are executed just once, thus visually represented as thin and long boxes. The method `b` does not cause a side effect on the object receiver. No benefit will be gained by caching `b` if it is called just once.

Pattern H - non altering delegating chain. The method `a` calls the method `b` several times, which itself calls `c`. The method `b` and `c` do not do a side effect on their object receiver and are executed the same amount of times since they have the same width.

This chain shows that the method `c` is time consuming since `a`, `b` and `c` have the same height. Introducing a cache in the Method `b` will leads to a performance improvement.

Pattern I - non altering and time consuming delegation. The method `b` is called several times by `a`; `b` is time consuming and does not modify the object receiver. Caching `b` will produce a performance gain.

3. DEMONSTRATION

The Kai profiler will be demonstrated to the Dyla workshop. Kai offers a range of interactive actions that will be presented. We hope the demonstration will trigger some discussion with respect to the usability of Kai and its possible extensions.

The objectives of the demonstrations are

- to present the Kai profiler
- to compare Kai with other code execution profilers
- to discuss the relevance of Kai in terms of performance problems

Acknowledgment. This work has been partially funded by Program U-INICIA 11/06 VID 2011, grant U -INICIA 11/06, University of Chile, and FONDECYT project 1120094.

4. REFERENCES

[1] A. Bergel, F. Bañados, R. Robbes, W. Binder, Execution profiling blueprints, Software: Practice and Experience 42 (9) (2012) 1165–1192. doi:10.1002/spe.1120.

[2] A. Bergel, F. B. nados, R. Robbes, D. Röthlisberger, Spy: A flexible code profiling framework, Journal of Computer Languages, Systems and Structures 38 (1). doi:10.1016/j.cl.2011.10.002.

[3] M. Lanza, S. Ducasse, Polymetric views—a lightweight visual approach to reverse engineering, Transactions on Software Engineering (TSE) 29 (9) (2003) 782–795. doi:10.1109/TSE.2003.1232284.

[4] C. Ware, Information Visualisation, Elsevier, Sansome Street, San Fransico, 2004.