Efficiently Identifying Object Production Sites

Alejandro Infante, Alexandre Bergel

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

***Abstract*—Most programming environments are shipped with accurate memory profilers. Although efficient in their analyses, memory profilers traditionally output textual listing reports, thus reducing the memory profile exploration as a set of textual pattern-matching operations.**

***Memory blueprint* visually reports the memory consumption of a program execution. A number of simple visual cues are provided to identify direct and indirect object production sites, key ingredients to efficiently address memory issues. Scalability is addressed by restricting the scope of interest both in the call graph and the considered classes. Memory blueprint has been implemented in the Pharo programming language, and is available under the MIT license.**

I. Introduction

Debugging memory issues is known to be tedious and challenging. Memory profilers are dedicated tools to help practitioners address memory anomalies, including memory leaks or suspicious memory fragmentation. The primary objective of a memory profiler is to detail the memory allocated during a code execution. A memory profiler essentially reports its analysis by listing the number of instances per class (*e.g.,* Yourkit[1], JProfiler[2]). Such a listing is typically presented as a tree-widget in which each method context node is annotated with the resources it consumes. Using a textual medium to convey a complex data set, such as the memory usage profile, involves textual operations (such as searching and comparing) which are known to be suboptimal when compared to a dedicated visualization tool[3]. Visualizing the execution and tracing the memory consumption has been the topic of numerous research efforts [2], [3], [4]. Although efficient at indicating the global memory state, many of the proposed techniques for heap and memory visualization [5], [6], [7], [8] have been designed for program behavior comprehension and understanding. As far as we are aware, none of the proposed visualization techniques are made to identify and characterize *object production sites*, *i.e.,* methods and static initialization parts that create objects.

Memory blueprint is a visual representation of a memory profile. Our memory blueprint indicates direct and indirect object production source code sites. Such information is meant to be used by practitioners to identify memory bloat and optimization opportunities.

Our blueprint augments the method call graph obtained from a traditional code profiler with visual cues to indicate and characterize object production sites. Memory blueprint is a polymetric view [9] that (i) reports direct and indirect object production sites along a call-graph, (ii) characterizes the production site using dynamic metrics using visual cues, (iii) supports incremental execution, and (iv) offers interactive options to drill-down from the visual report to the actual source code.

This paper describes memory blueprint, and presents some practical scenarios for which visualization plays an important role in understanding a memory profile.

The paper is structured as follows: Section II describes memory blueprint. Section III highlights some of our experience when applying memory blueprint in an industrial setting. Section IV gives a brief overview of the related work. Section V concludes and presents our future work.

II. Memory Blueprint

A. In a Nutshell

Properly identifying and characterizing object production sites is critical to understanding the memory footprint of an application. Consider the following contrived, but representative, example[4]:

```
class Element { int x, y; }

class ElementFactory {
  Element create() { /* D */
    return new Element();
  }
}

class Canvas {
  List<Element> elements = new ArrayList<Element>(); /* B*/

  void add(Element e) { elements.add(e); }

  void fillWithElements(int nbElement) { /* C */
    ElementFactory factory = new ElementFactory();
    for(int i = 0; i < nbElement; i++) this.add(factory.create());
  }

  public static void main(String[] argv) { /* A */
    new Canvas().fillWithElements(100);
  }
}
```

The code given above defines three classes and five methods.

[1] http://www.yourkit.com

[2] https://www.ej-technologies.com/products/jprofiler/overview.html

[3] Wettel *et al.* [1] have demonstrated an improvement of correctness and completion time of some classical software engineering tasks when using the CodeCity visualization tool compared with Eclipse and Excel.

[4] We give a Java code excerpt as an illustration. Memory blueprint is written for the Pharo programming language (http://pharo.org).

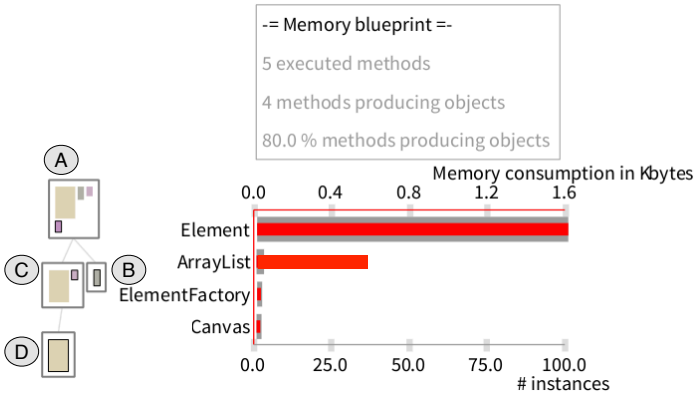

Fig. 1: Memory blueprint example

TABLE I: Memory blueprint (call graph) specification

visual dimension	description
outer box	factory method
outer box border	light gray = indirect object producer; black = direct object producer
outer box layout	cycle-resistant tree layout
inner colored box	instantiated class by the encapsulating method
inner box color	a unique color to distinguish classes
inner box border	dark = direct production site
inner box width	number of instances created by the encapsulating method (log scale)
inner box height	number of bytes allocated by the instances created by the encapsulating method (log scale)
inner box layout	grid layout; classes are ordered according to their consumption
edge	an upper method call methods located below

There are four object production sites, marked in **bold** in the code given above:

- `create()` instantiates the class `Element`
- `fillWithElements(int)` instantiates `ElementFactory`
- `main(...)` instantiates `Canvas`
- the constructor of `Canvas` instantiates `ArrayList`.

Profiling the code example and reporting the result using memory blueprint results in the visualization reported in Figure 1. The central horizontal chart indicates that there are 100 instances of the class `Element`, consuming 1,600 bytes (each instance of `Element` weights 16 bytes, two 4-byte variables and an 8 byte-long object header). The left hand-side of Figure 1 indicates the call-graph of the methods and constructor producing objects. The `main(...)` method, indicated with `A` on the figure, invokes the Canvas's constructor (`B`) and `fillWithElements(int)` (`C`), itself invoking `create()` (`D`).

Each inner box refers to a class instantiated during the execution. The gray border around an inner box indicates that the class is directly instantiated by the encapsulating method, otherwise it is indirect. The `A` method directly instantiates the class `Canvas`, indicated by the gray-bordered pink box. The pink box is relatively *small*, meaning that *very few* instances from `Canvas` has been created, and this instance does not consume *much* memory. We have deliberately used the informal terms (*e.g., small, very few*) to designate a relative visual comparison. A tooltip window indicating the exact represented number is accessible by moving the mouse above the visual element.

Similarly, one can notice that the constructor `B` instantiates the class `ArrayList` (class and methods names are obtained by placing the mouse cursor above a box). Method `A` and `C` indirectly instantiates `Element` by calling the `D` method.

B. Memory Blueprint Detailed

Blueprint description. Our blueprint is composed of two complementary reports: (i) a partial method call graph that indicates memory consumption along the execution path, located on the left-hand side, and (ii) a quantitative report represented as a double-bar chart, located on the right-hand side. The partial method call graph is a polymeric view [9], specified in Table I.

Figure 2 illustrates memory blueprint with an execution of an application using a complex graphical user-interface. The partial method call graph is made of 38 methods producing objects. Only methods that are directly or indirectly producing objects are shown in the call graph: the complete call-graph has 234 methods, for which only 38 methods directly or indirectly produce objects.

A double-bar chart is located on the right hand side of Figure 2. This chart offers quantitative information for each class instantiated. Classes are ordered along their number of instances created by the encapsulating method. The red bar indicates the memory consumption totaled by all the class instances (number of instances multiplied by the size in byte of each instance). The gray bar indicates the number of instances created during the program execution.

Identifying object production sites. The method indicated with "start" in Figure 2 is the entry point of the execution. This method instantiates 20 different classes, for which only 4 classes are directly instantiated by the start method (*i.e.,* the source code of the start method contains three class instantiations). The remaining 16 classes are indirectly instantiated, *i.e.,* methods directly or indirectly called by `start` instantiate these classes.

Visual cues. Memory blueprint uses several *visual cues* to indicate relevant parts of the visualization. We designed the call graph to be intuitive: large methods are methods that consume a large portion of memory, small methods consume a relatively small portion of memory. A method is located below its calling methods, whenever possible.

The color border indicates a production site: (i) a method directly producing some objects is highlighted in black, and (ii) a class contained in a method has a black border if the encapsulating method directly instantiates that class.

Each class has a particular color. Class colors are a visual aide to indicate occurrence of classes along the method call-graph. Class details may be obtained by locating the mouse above it, as a tooltip. We employ an algorithm that tries to apply a distinct color to each of the most memory consuming classes.

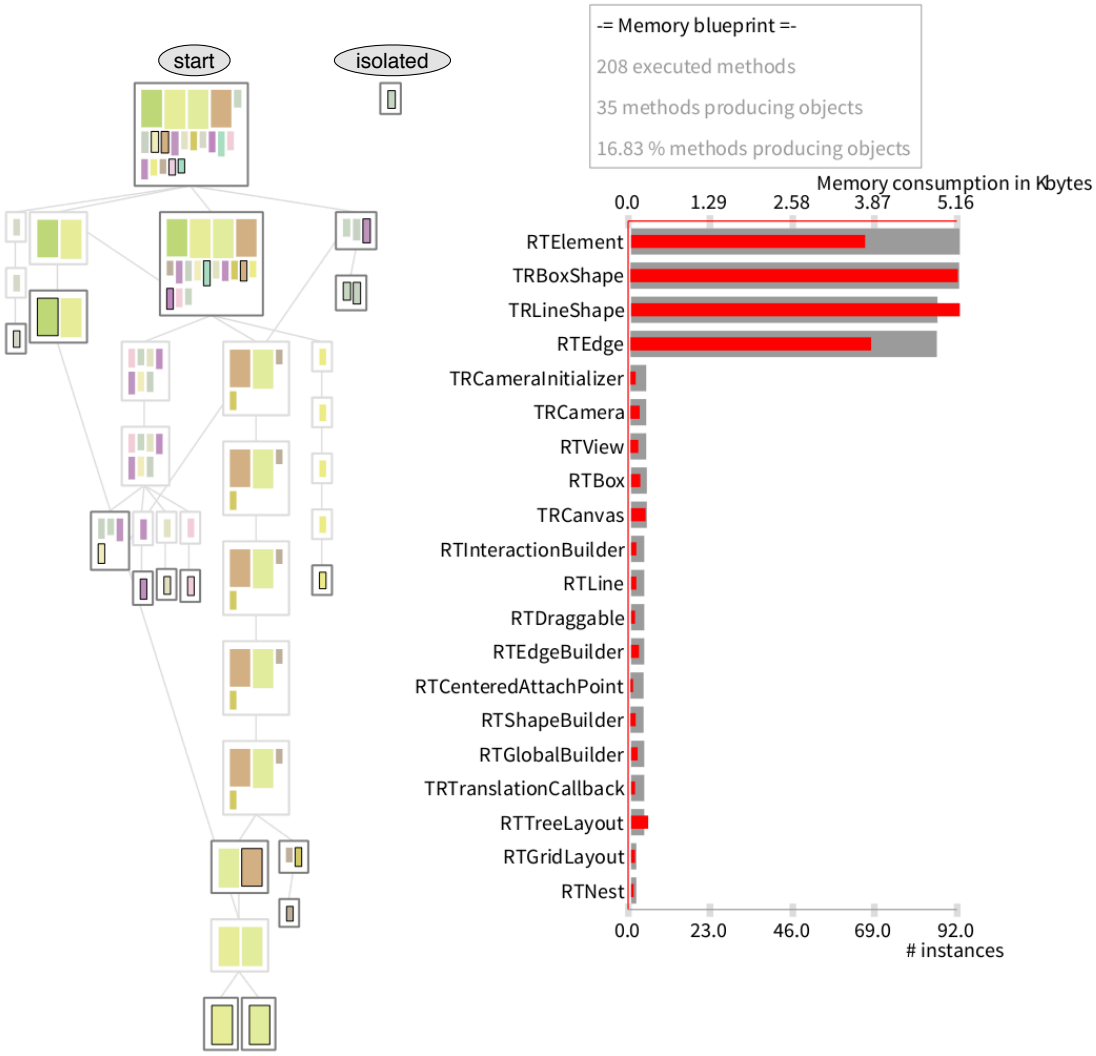

Fig. 2: Memory blueprint profile

We employ the color scheme proposed by ColorBrewer[5].

Cycle in the call graph. A method call sequence may form a cycle. The memory blueprint uses a tree layout to order represented methods. In presence of a cycle, edges extremities indicates the direction of the method invocation.

Consider the situation described in Figure 3 with three methods: `A` calls `B`; `B` calls `C`; `C` calls `A`. These three methods are cycling. An edge, representing a method call, begins from the bottom of the calling method and ends at the top of the invoked method. This way to represent cycles is similar to the cycle representation in Class Blueprint [11].

Profile exploration. Memory blueprint supports several options to interact with practitioners. Each element of the memory blueprint has a contextual menu and tool tip. These interactions are intended to facilitate the navigation and the exploration of the code profile. In particular jumping from a method to the source code is one click away.

[5]http://colorbrewer2.org [10].

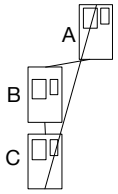

Fig. 3: Representation of cycles between method invocations

By locating the mouse cursor over class names contained in the double-bar chart, methods in the call-graph directly instantiating the pointed class are highlighted. This complements well the class coloring mechanism since class names are visible only by mouse tooltip. Methods may be dragged and dropped to accommodate the layout.

Profiling. The memory profile is obtained via a third-party application. In our case, we designated a memory profiler that associate classes to method producing objects using Spy, a

profiling framework [12] for the Pharo language.

III. Experience Gained

Memory blueprint is regularly employed in an industrial setting. ObjectProfile.com is a software company. Engineers of this company have used Memory Blueprint to track memory bloats. Such experience has been crucial for adequately tuning our visualization.

Call graph size. Call-graphs for non-trivial application may be large. Moret *et al.* [13] report that for the DaCapo benchmarks, the number of methods involved in a computation may go beyond 11,000 and the depth of a context-calling tree may be over 400. These figures are comparable with the situations we regularly meet in our development.

Memory blueprint only reports a subset of the complete method call graph: methods that are directly or indirectly producing objects are reported while methods not producing objects are not reported (*i.e.,* variable accessors). The rationale behind this decision we made is that a practitioner will necessarily focus on methods producing objects when addressing a memory issue.

A positive aspect of reporting only object production sites is to *significantly* reduce the call graph, since on average less than one third of the methods are object production sites. We considered 10 different and representative applications in the Pharo software ecosystems. We found that the ratio of executed methods being a production site ranges from 5% to 75%, with an average of 29%, a median of 23% and with a standard deviation of 17. Only such a small portion of the call graph is represented in memory blueprint. On a 27-inch screen, our blueprint reports graphs large of hundreds of methods without needing to scroll or zoom out. This is enough for most of the situations we have faced.

Local vs global. Traditional memory profilers report for a code execution the global memory consumption. Such reports are particularly efficient at conveying the general impression of how the memory is used. It also indicates classes that are likely to be problematic. However, quickly identifying culprit methods from such a global report is not trivial.

Memory blueprint *complements* traditional memory reports by indicating which methods directly and indirectly create objects. From our experience, memory blueprint is well adapted to track object production sites for dozens of classes for a call-graph less then 500 methods. Memory blueprint is apparently suitable for local memory introspection (such a claim will be carefully measured in our future work).

Linear and logarithm scales. The method-call graph reported on the left-hand side of a blueprint indicates for each method (i) the classes the method directly or indirectly instantiated, (ii) the number of produced instances, and (iii) the number of bytes used by these instances. These two metrics are visually reported using a logarithmic scale. Logarithm scales handle well disparities between represented values. The double-bar chart, reported on the right-hand side, uses a linear scale. This is useful for easily identifying large memory consumption.

The linear scale has the benefit of letting the practitioner informally relate classes from their number of instances. Such relation may pinpoint application invariant, letting the programmers deciding whether the invariant is correct or not. For example, consider the example given in Figure 2. The gray-bar indicates that the class `RTElement` and `TRBoxShape` have the same number of instances. Classes `TRLineShape` and `RTEdge` have also the same number of instances. These relations between classes may reveal an invariant of the profiled application: the classes `RTElement` / `TRBoxShape` and `TRLineShape` / `RTEdge` have the same amount of instances.

The size of the input often correlates with the number of objects involved in a computation. One of the case studies we studied consists in displaying an annotated world map. In total, 169 countries were represented using a SVG path. We have found that exactly 507 SVG path objects were created, which is three-times more than the number of countries. This revealed that 2 extra SVG path objects were created per country. This information was useful for improving the memory footprint of this application.

To accommodate the difference between different visual elements in a profile, the logarithm scale used in the call graph may be replaced with a softer (*e.g.,* square root) or a more aggressive disparity reduction scale (*e.g.,* adding a factor).

IV. Related Work

Analyzing and visualizing memory consumption has been the topic of several research works. This section summarizes the most prominent works in the field.

Condensed Run-time Information. Ducasse *et al.* [3] have employed polymetric views to visualize different aspects of an executing system. The *instance usage overview* shows which classes are instantiated and used an execution. The *communication interaction view* shows the communication between classes of a system. The *creation interaction view* shows instance creation between classes.

Waxlamp. Understanding how caches are used during a program execution is subtle and delicate. Waxlamp [14] presents a circular visual representation of cache behavior. The visualization represents salient events related to cache hits and misses. Time is circularly represented, producing a compact view.

Visualizing the Heap. Reiss [5] proposes a condensed visual representation of the heap. The visualization is structured along the class hierarchy. Size of the class indicates the number of instances the class has created during an execution. It also partially support differentiation between different executions.

Memory allocation and death plot. Veroy *et al.* [8] presents a scalable visualization that indicates in which method an object was allocated and in which method that object is likely to die. The proposed visualization uses a hive plot[6] [15] to indicate object profile sites. Contrary to memory blueprint, method call graphs are not presented.

V. Conclusion and Future Work

Identifying where and how in a source code objects are instantiated is difficult, despite the relevance to address memory-related issues. In addition, understanding the relation between

[6] http://www.hiveplot.net

factory methods helps practitioners understand who are the real culprits when facing an abnormal memory consumption. This paper presents memory blueprint, a visual representation of a memory consumption along the method call graph. Our blueprint is designed to easily identify and characterize object production sites.

Our memory blueprint has been employed in an industrial setting, which gives us confidence about the design decisions made in our blueprint. However, such claims have not been empirically validated. This is the topic of our future work.

We hope the work presented in this paper will contribute to the community to explore alternatives to textual and list-based memory profile reports.

ACKNOWLEDGMENTS

This work was partially supported by FONDECYT project 1120094 - Chile and by Program U-Apoya, University of Chile.

REFERENCES

[1] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 551–560. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985868

[2] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak, "Visualizing dynamic software system information through high-level models," in *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*. ACM, Oct. 1998, pp. 271–283.

[3] S. Ducasse, M. Lanza, and R. Bertuli, "High-level polymetric views of condensed run-time information," in *Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*. Los Alamitos CA: IEEE Computer Society Press, 2004, pp. 309–318.

[4] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[5] S. P. Reiss, "Visualizing Java in action," in *Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization)*, 2003, pp. 57–66.

[6] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer, "Heapviz: interactive heap visualization for program understanding and debugging," in *Proceedings of the 5th international symposium on Software visualization*, ser. SOFTVIS '10. New York, NY, USA: ACM, 2010, pp. 53–62. [Online]. Available: http://doi.acm.org/10.1145/1879211.1879222

[7] C. Myers and D. Duke, "A map of the heap: Revealing design abstractions in runtime structures," in *Proceedings of the 5th International Symposium on Software Visualization*, ser. SOFTVIS '10. New York, NY, USA: ACM, 2010, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/1879211.1879223

[8] R. L. Veroy, N. P. Ricci, and S. Z. Guyer, "Visualizing the allocation and death of objects." in *Proceedings of IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 2013, pp. 1–4.

[9] M. Lanza and S. Ducasse, "Polymetric views—a lightweight visual approach to reverse engineering," *Transactions on Software Engineering (TSE)*, vol. 29, no. 9, pp. 782–795, Sep. 2003.

[10] C. A. Brewer, G. W. Hatchard, and M. A. Harrower, "Colorbrewer in print: a catalog of color schemes for maps," *Cartography and geographic information science*, vol. 30, no. 1, pp. 5–32, 2003.

[11] S. Ducasse and M. Lanza, "The Class Blueprint: Visually supporting the understanding of classes," *Transactions on Software Engineering (TSE)*, vol. 31, no. 1, pp. 75–90, Jan. 2005.

[12] A. Bergel, F. Bañados, R. Robbes, and D. Röthlisberger, "Spy: A flexible code profiling framework," *Journal of Computer Languages, Systems and Structures*, vol. 38, no. 1, Dec. 2011.

[13] P. Moret, W. Binder, A. Villazón, D. Ansaloni, and A. Heydarnoori, "Visualizing and exploring profiles with calling context ring charts," *Softw. Pract. Exper.*, vol. 40, no. 9, pp. 825–847, Aug. 2010. [Online]. Available: http://dx.doi.org/10.1002/spe.v40:9

[14] A. N. M. I. Choudhury and P. Rosen, "Abstract visualization of runtime memory behavior," in *IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 2011, pp. 1–8.

[15] M. Krzywinski, I. Birol, S. Jones, and M. Marra, "Hive plots – rational approach to visualizing networks," *Briefings in Bioinformatics*, dec 2011.