Visualizing the Allocation and Death of Objects

Raoul L. Veroy, Nathan P. Ricci, Samuel Z. Guyer
Tufts University
Medford, MA
{rveroy, nricci01, sguyer}@cs.tufts.edu

***Abstract*—We present memory allocation and death plots, a visualization technique for showing both which method an object is allocated in a Java program, and in which method that object eventually dies. This relates the place in a program's execution where memory is first used to the place it is no longer used, thus helping the programmer to better understand the memory behavior of a program.**

I. Introduction

"How does my program use memory?" is a question programmers often have to answer. There are many tools available to answer at least part of this question; there are a plethora of ways to understand where objects are allocated, or what objects are resident in the heap at a particular point in time. However, there are few tools that can show the programmer where objects die.

To answer this last question, we present a system which determines the context in which objects die using a memory tracing tool, and presents this information to the programmer using a hive plot visualization [1]. We term such a visualization a *memory allocation and death plot*. The memory allocation and death plot of a program lets a programmer identify both the hot sites of object allocation and death, and quickly relate them.

II. Our Design

A. Data

We use data from *Elephant Tracks*, a dynamic program analysis tool for Java which produces detailed traces of garbage collection-related events [2]. The traces produced by Elephant Tracks have a record of all of the object allocations and deaths that occurred during execution of a program. An object is considered dead when it is no longer reachable from the roots (local and global variables) which is often different from when the object is actually collected. Elephant Tracks is able to place object allocation and death with sufficient precision to determine in what calling context (i.e., in what thread, and what methods were on the stack of that thread) those events occurred. The context information was not previously available, and thus its presentation will require novel visualization strategies which we will discuss in the next section.

Understanding some of the interesting cases of memory behavior requires a brief digression into garbage collection. Most modern JVM garbage collectors are *generational garbage collectors*. Generational collectors exploit the observation that most objects die young. New objects are allocated in a small (on order of megabytes), frequently collected, space called the *nursery*. Objects that survive a nursery collection are promoted to a larger, less frequently collected space.

For a generational collector, one problematic case is the object that survives the nursery, but then dies soon after. Such objects will remain resident in memory until a full heap collection occurs, even though they are dead. Objects in the *eclipse* benchmark from the DaCapo suite [3] frequently exhibit this behavior. Therefore we traced *eclipse* with Elephant Tracks to see if we could determine anything about these nursery escapees through visualization.

The Elephant Tracks trace includes a large amount of information; over 65 000 000 objects are allocated during the execution of the program, and the trace itself is roughly 30 GiB without compression. In order to cope with this data, we took several steps to reduce it. First, since we are primarily interested in objects that live long enough to escape the nursery, we filtered out short lived objects; for our purposes short lived objects are those that survived less than 8 MiB of allocation.

Second, we reduce the size of our contexts. Although we have the complete calling context for each object's death and allocation, the number of such contexts for *eclipse* is approximately 40 000. To reduce their number, we trim them to a single method (the method in which the event occurred). Furthermore, we group these methods based on their class. Thus, the *allocation context* of an object refers to the class that contains the method where the object was allocated, and the *death context* refers to the class containing the method where it died. This reduces the number from 40 000 contexts to approximately 2800 classes.

After taking these steps, our raw data consists of pairs of allocation and death contexts, one for each object allocated during our run of the *eclipse* benchmark.

B. Visual Encoding

The relationship of the object's allocation context to the death context can be represented as a graph. We propose to use hive plots, a layout algorithm for network diagrams developed by Krzywinski [1]. Nodes represent allocation and death contexts. Edges start with allocation contexts as the source and are connected to the death context where the object dies. Nodes are arranged radially on linear axes according to user defined rules. The rules determine assignment of nodes to axes, position and orientation of axes, and edge rendering.

By applying the same rules to the trace data, our visualization representation becomes repeatable and comparable.

We place axes similar to a 2D Cartesian layout. We shall call the axes *north*, *south*, *east* and *west*. The node corresponding to a context is assigned to an axis depending on whether the context is allocation only, death only or both. Allocation only nodes are assigned to the *north* axis. Death only nodes are assigned to the *west* axis. Nodes which serve as both allocation and death contexts are then placed on the *south* axis. The *south* axis is then cloned as the *east* axis. This enables the visualization to better handle edges that loop back to the *south* axis as well as self-edges.

As a result of our axes assignment rules, both vertical axes contain allocation sites and both horizontal axes contain death sites. This also means that we are able to represent direction as the edge endpoint incident to the vertical axes will always be the source and the edge endpoint incident to the horizontal axes will always be the target.

Given our axes assignment rules, edges can now be rendered as simple curves without losing direction information. Self edges only exist from the south to east axis quadrant. We color these edges green in our implementation to be able to distinguish self edges from regular edges which are colored gray. Classes (recall that each context is a single Java class) that belong to the same package are then grouped together and given the same color.

Even with the data reduction already described, the resulting memory plot for *eclipse* is still too cluttered. Therefore, we further decompose the hive plot into a group of hive plots. We group the edges into 4 sets based on the in degree of the death context nodes (0 to 200, 201 to 400, 401 to 600, everything greater than 600), and assign each grouping to a plot within the panel. Figures 2, 3, 4 show the hive plots for degrees 1 to 600. Figure 1 shows only the edges adjacent to death context nodes with in degree greater than 600.

We rendered the hive plots using the HiveR package [4] for the R programming language [5]. The data was processed using a combination of programs written in C and Python.

C. Results

Using our design, we are able to identify which classes are hot spots for object death. In Figure 1, we are able to identify six different classes where a lot of objects die, the largest of which is the *CharOperation* class. The majority of edges here are not self-edges which means that objects were allocated in a different class. Prompted by the visualization, we can explore the data and find that, within the CharOperation class, the plurality of objects die within the *splitOn* method. This method takes a character array, and breaks it into tokens based on a delimiting character passed as a parameter. The fact that significant numbers of objects die here indicates that the last act preformed on many of these character arrays is to be divided into pieces. Whether it would be possible to use this information to improve the memory performance of *eclipse* is unclear, and something we would like to explore in future work.

In contrast to Figure 1, Figure 2 has a higher number of self-edges. There is an interesting difference in structure in Figure 2 as compared to the other object flow diagrams, as evidenced by the distribution of edges. Our design limits us though in how we can relate this information to the program structure. We propose alternative designs in the following section that will further help the programmer in understanding memory flow and death.

III. Design Alternatives

Our node placement rules used class and package information to arrange the nodes along the axes. Although this rule allows us to somewhat minimize the visual clutter, the position along the axes does not provide us any interesting insight regarding the death behavior of objects. We propose to use some form of program time as the parameter to determine node position along the axis. Some chronological arrangement of the nodes is more likely to be a better use of the axial dimension.

In our design, the color of a node is associated with the package of the class where the context can be found. While it allows us to differentiate nodes according to package membership, this may not be the best use of this visual dimension. One possible alternative is to use color to represent node degree.

Our current implementation is non-interactive as we initially sought to develop and evaluate ideas for the visualization. Interactivity is necessary if the visualization is to be useful to programmers. An interactive implementation of the visualization would at least include the ability to identify contexts by interacting with the visual representation and the ability to modify the design parameters.

The large number of edges in the resulting visualization makes the flow trends virtually impossible to discern in for non-trivial traces. Applying edge-bundling is one possible way to reduce the visual clutter [6].

IV. Related Work

Although to our knowledge there are no existing visualization tools which relate the allocation and death contexts of an objects, there are numerous tools for visualizing the heaps of programs to provide other information.

De Pauw et al. present a tool focused on finding references which cause memory leaks [7]. Although our visualization is aimed at understanding object lifetimes (not memory leaks), memory leaks are inherently related to object lifetimes, in that they are one or more objects that remain reachable well past the time of their last use. De Pauw's technique aggregated objects with similar reference patterns, as well as interactive navigation and contraction/expansion of aggregates. Their technique, however, does not identify the contexts in which objects become unreachable, instead relying on the programmer to specify points where they expect certain objects to die, and calling their attention to the references causing violation of these programmer expectations.

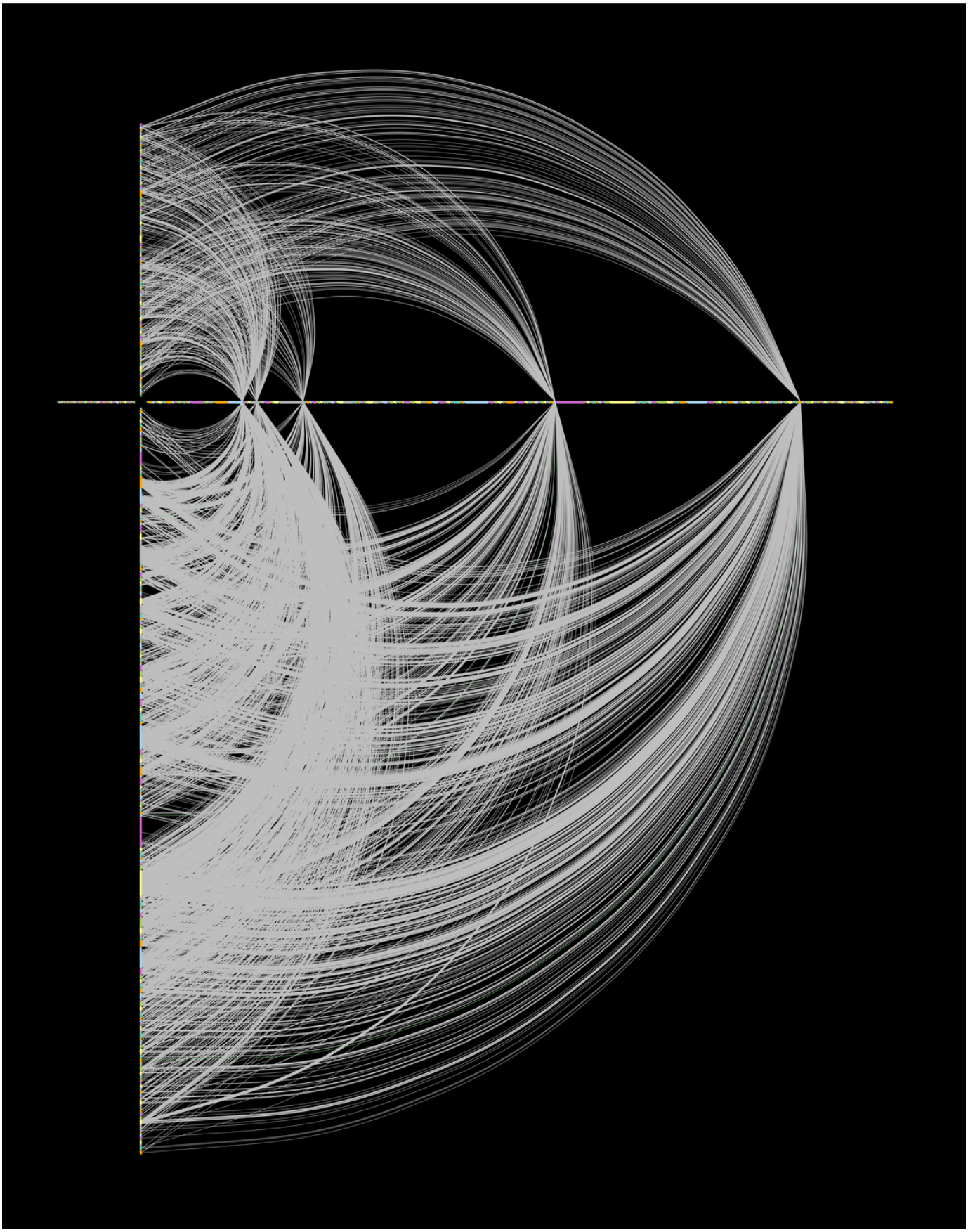

Fig. 1. In degree $d \mid d > 600$

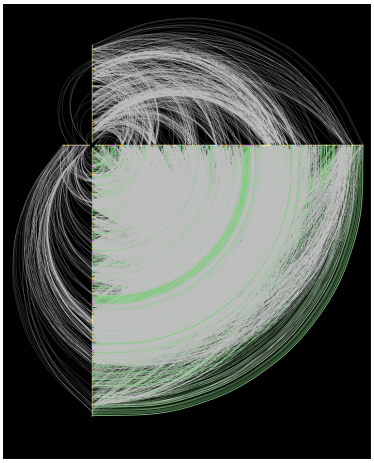

Fig. 2. In degree $d \mid 1 \leq d \leq 200$

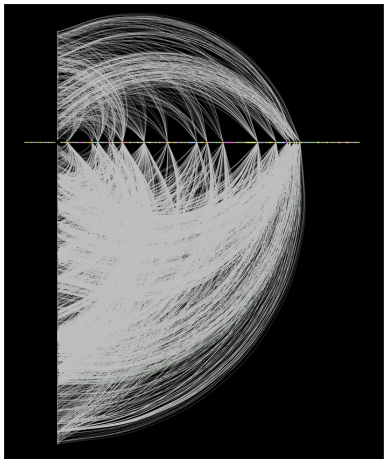

Fig. 3. In degree $d \mid 201 \leq d \leq 400$

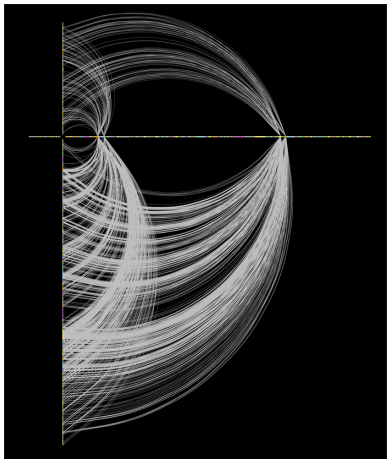

Fig. 4. In degree $d \mid 401 \leq d \leq 600$

Reiss [8] presents a visualization of the memory usage of Java programs. The visualization employs a summary of the Java heap based on object ownership, and is intended to help programmers diagnose errors involving excessive or incorrect usage of memory. It does not, however, provide information about when or where the involved objects eventually cease to be used.

In a similar vein, Mitchel et al. [9]'s work attempts to expose the runtime costs of design decisions. In order to do so, they created a visualization of the Java heap which aggregates nodes based on type and object ownership, and produces a graph of the resulting aggregation with nodes displaying the total size of the objects that underlies them. Taking advantage of ownership (and some edge pruning heuristics) allows the programmer to see the sizes of not just individual objects, or even the amount of memory used by a certain type, but the amount of memory used by whole data structures. By itself, however, it does not offer any guidance as to when the runtime might be done with that memory.

Printezis and Jones [10] GCSpy framework provides a way for visualizing the memory behavior of a program. It can connect to multiple VMs and work with data from traces. However, it is primarily focused on providing information about the memory management system. Memory management is of course, closely related to object death, since uncollected dead objects will continue to occupy memory. While one could use GC Spy to visualize, for example, which objects were dead after a garbage collection, it would not be simple to determine in what context they had died.

V. Conclusions

We have presented a technique, memory allocation and death plots, for visualizing the flow of objects from their allocation context to the context in which they eventually die. We are able to see intriguing patterns of the memory usage of a program using this technique. Nevertheless, many significant questions about how to best visualize and make use of this information remain. Can we further reduce visual clutter? Can we integrate more information about the memory properties of a program into our visualization? Are there other potential use cases? How can we make use of interactivity? These are questions we hope to explore further in the future.

References

[1] M. Krzywinski, I. Birol, S. J. Jones, and M. A. Marra, "Hive plots-rational approach to visualizing networks," *Briefings in Bioinformatics*, vol. 13, no. 5, pp. 627–644, 2012.

[2] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss, "Elephant tracks: portable production of complete and precise gc traces," in *Proceedings of the 2013 international symposium on International symposium on memory management*, ser. ISMM '13. New York, NY, USA: ACM, 2013, pp. 109–118.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: java benchmarking development and analysis," *SIGPLAN Not.*, vol. 41, no. 10, pp. 169–190, Oct. 2006.

[4] B. A. Hanson, *HiveR: 2D and 3D Hive Plots for R*, 2013, r package version 0.2-10.

[5] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013.

[6] D. Holten and J. J. Van Wijk, "Force-directed edge bundling for graph visualization," *Computer Graphics Forum*, vol. 28, no. 3, pp. 983–990, 2009.

[7] W. D. Pauw and G. Sevitsky, "Visualizing reference patterns for solving memory leaks in java," in *in Proceedings of the ECOOP 99 European Conference on Object-oriented Programming*. Springer-Verlag, 1999, pp. 116–134.

[8] S. P. Reiss, "Visualizing the java heap to detect memory problems," in *In VISSOFT 09: Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2009, pp. 73–80.

[9] N. Mitchell, E. Schonberg, and G. Sevitsky, "Making sense of large heaps," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 77–97.

[10] T. Printezis and R. Jones, "Gcspy: an adaptable heap visualisation framework," in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '02. New York, NY, USA: ACM, 2002, pp. 343–358.