Heap Evolution Analysis Using Tree Visualizations

Markus Weninger[⋆], Lukas Makor[⋆⊗], Hanspeter Mössenböck[⋆]
{firstname.lastname@jku.at}
[⋆] Institute for System Software, Johannes Kepler University, Linz, Austria
[⊗] Christian Doppler Laboratory MEVSS, Johannes Kepler University, Linz, Austria

Abstract

Memory anomalies such as memory leaks can dramatically impact application performance and can even lead to crashes. Thus, supporting developers in understanding the heap memory behavior of their systems is essential. Unfortunately, most memory analysis tools lack advanced visualizations that could facilitate developers in analyzing suspicious memory behavior.

To analyze heap memory, it is common to group the heap's objects, for example, by their types or by their allocation sites. Using multiple grouping criteria thus results in a tree-shaped representation of the heap content. Such a heap tree is then typically presented textually in a tree table.

In this paper, we present ongoing research on using well-known tree visualization techniques to visualize such heap trees as well as their evolution over time. Such visualizations may ease the detection of proliferating heap objects, facilitating memory leak analysis.

To demonstrate the feasibility and applicability of the presented approach, we implemented a web-based visualization tool and integrated it into AntTracks, our trace-based memory monitoring tool.

1 Introduction

Modern programming languages such as Java use garbage collection to automatically reclaim objects that are no longer reachable from static fields or thread-local variables. While this prevents a number of programming errors, certain problems such as memory leaks can still occur. For example, a developer may forget to remove objects from their containing long-living data structure. Consequently, these objects cannot be reclaimed by the garbage collector and thus accumulate over time [11].

Memory leaks cause more frequent garbage collections, which can have a significant negative performance impact. Worse, running out of memory even crashes the application. Thus, it is essential to provide tools to facilitate developers in detecting proliferating heap objects in their applications.

Even though *data visualization* [5] can help to convey information faster [6] and can aid in identifying patterns [7], most state-of-the-art memory monitoring tools do not take advantage of visualizations (except for time-series charts) and often present data in the form of tables and lists.

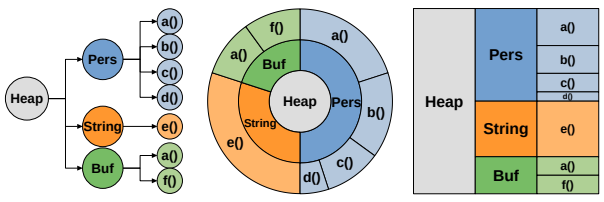
In this paper, we present work-in-progress to visualize memory evolution over time using tree visualizations. Our approach groups similar heap objects based on properties such as type, allocation site, or allocating thread into a *heap tree* (see Section 2). We then use well-known tree visualizations, i.e., the *sunburst plot* [1] and the *icicle plot* [9] to visualize the heap content at a single point in time (see Section 3). Generating heap trees at multiple points in time enables users to step through time to inspect the monitored application's heap evolution over time (see Section 4). This helps users to recognize suspiciously growing object groups that may hint at memory leaks.

2 Data Collection

To inspect the heap at single point in time, most tools use heap dumps. Yet, to visualize the heap's evolution over time, we need *continuous* information about its objects. Since regularly dumping the heap would incur too much run-time overhead (as the application is halted during the dump), we use the AntTracks VM [8], a modified Java virtual machine based on the Java Hotspot VM, to collect continuous memory traces (which introduces only around 5% run-time overhead [8]). We can reconstruct the heap state from such a trace at every garbage collection point. For every heap object, a number of properties can be reconstructed, including its address, type, allocation site, and the thread that allocated it. The heap objects can then be grouped by a user-defined combination of these properties which results in a *heap tree* [10].

3 Heap State Visualization

There is ample work on how to visualize tree-shaped data. Based on user studies that evaluated the usefulness [2] and aesthetics [3] of tree visualizations, we decided to use the sunburst plot as well as the icicle plot to visualize heap trees.



(a) Tree. (b) Sunburst. (c) Icicle.

Figure 1: Three visualizations showing the same data.

3.1 Tree Visualizations

In a heap tree (Figure 1a), each tree node represents a group of heap objects. A good heap tree visualization should show the parent/child relationships as well as the number of objects/bytes represented by a specific node; the latter can be expressed as node size, but is often missing in simple tree visualizations. Thus, the *sunburst plot* (Figure 1b) as well as the the *icicle plot* (Figure 1c) use variable-sized graphical elements to visualize nodes. The *sunburst plot* uses ring segments, where the angular size of the ring segments encodes a value. The *icicle plot* uses rectangles to encode a value using the rectangle's height. Instead of using explicit links to depict the tree hierarchy, in the sunburst plot the tree hierarchy is moving outwards, starting at a *root circle* in the middle. Similarly, in the icicle plot the tree hierarchy is moving from left to right.

To compare these tree visualizations, Figure 1a through Figure 1c visualize the same data. Imagine that the underlying heap tree was generated by grouping all heap objects by their types, and all objects of the same type by their allocation sites. The gray root node (`Heap`) represents the whole heap. The nodes on the first level represent different types. For example, we can see that the heap consists of objects of the types `Pers` (blue), `String` (orange), and `Buf` (green). In the sunburst and icicle plot we can further see that 50% of the heap space is taken up by objects of type `Pers`. On the second level, allocation sites are shown. There we can see that objects of type `Pers` have been allocated at four different allocation sites, most of them at site `a()`.

3.2 Handling Huge Trees

Heap trees can be too *wide* or too *deep* to be visualized as a whole. For example, real-world applications use objects of hundreds of different types, thus grouping the heap objects by type would result in a tree with hundreds of siblings, i.e., a *wide* tree. On the other hand, using multiple grouping criteria may lead to a tree with lots of levels, i.e., a *deep* tree. Thus, we apply *tree pruning* to narrow trees and provide a *drill-down* feature to hide deep tree levels by default.

Tree Pruning To reduce a heap tree's width, we only keep nodes that represent large object groups (i.e., those objects that most likely accumulated due to a memory leak), while smaller object groups are merged into artificial *"Other"* nodes. More specifically, we sort the child nodes of every node by their size (i.e., by their object count or byte count) and (1) keep the largest child nodes until they represent 90% of the objects on the current tree branch, yet we (2) keep a maximum of 9 child nodes. The remaining nodes are merged into an *"Other"* node.

Drill Down We only show two tree levels with the possibility to *drill down* into a certain tree branch. Clicking on a non-leaf node selects it as the new root of the visualization. Figure 2 depicts an icicle that groups all heap objects by *allocating thread*, then by *type* and finally by *allocation site*. The left-hand side shows the icicle without drill-down (the allocation sites are not visible since only two levels are shown). The right-hand side shows the icicle after drilling down into the node *Thread 2*. To step out again, the user can click on the *Thread 2* root node.

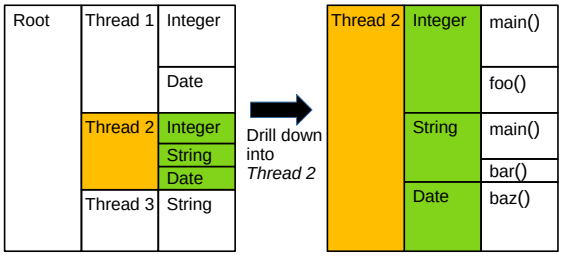


Figure 2: A drill-down feature enables users to explore deeper tree levels by selecting new tree roots.

4 Heap Evolution Visualization

It is not only possible to visualize the heap state at a single point in time, but also to visualize its evolution over time. For this, we use *time traveling* [4], a technique we already successfully applied in our *Memory Cities* visualization technique [12]. In time travelling, users can step back and forth through time, either using buttons or a time slider. After each step, the visualization updates itself to reflect the current heap state. Knowing which kinds of objects accumulate over time can greatly reduce the amount of source code that has to be inspected to fix a possible leak.

4.1 Stable Layout

A problem when switching from one point in time to another is that the order of the tree nodes could change. For example, if sibling nodes are ordered by size, and if their sizes change, the order of the nodes changes as well, which makes it hard to keep track of the evolution of different tree nodes.

In a *stable layout*, every node is assigned a sort position (based on a certain criterion) across all points in time *once* after all trees have been computed. This means that every node stays at the same relative position, e.g., at the second position, even if it grows or shrinks over time. In our heap evolution visualization, we currently sort all nodes based on their *end size*, i.e., based on their size in the last tree. For example, in Figure 3, the blue heap object group has the largest end size and is thus positioned first in both sunbursts.

4.2 Example

We implemented the presented approach as a `d3.js` web application[1]. In Figure 3, we show a composition of tool screenshots taken while inspecting *Dynatrace easyTravel*, a state-of-the-art demo application

[1]Prototype with example data hosted on `http://ssw.jku.at/General/Staff/Weninger/AntTracks/SSP20/WebTreeViz/`

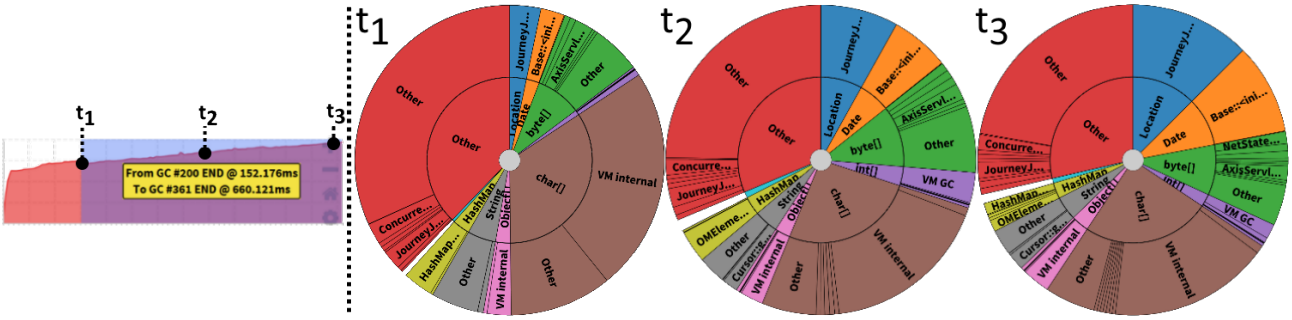


Figure 3: Heap evolution time travel through easyTravel shown at three different points in time t_1, t_2, and t_3.

that simulates a broken travel agency website. As apparent in the time-series chart, easyTravel's backend exhibits continuous memory growth. t_1 through t_3 exemplarily show our time travel sunburst visualization at three different points in time. The first tree level (inner ring) indicates the type, the second level depicts allocation sites. We can clearly see that the blue (type `Location`) and orange (type `Date`) segments grow. Both of these types are allocated only at a single allocation site each (as each type only has a single circle segment on the second level). Knowing which types accumulate the most objects (`Location` and `Date`) and where these objects are allocated (at a specific method in the class `JourneyJpaProvider`) makes it easy to locate the problematic code location.

5 Conclusion and Future Work

In this paper, we presented our approach to apply *tree visualizations* to facilitate heap memory analysis. We discussed how a heap state, more specifically its heap objects, can be grouped into a *heap tree* and how such a tree can be visualized using existing tree visualization techniques. We also visualize the heap evolution over time, where growing graphical elements hint at proliferating heap objects. These objects could be the result of a possible memory leak, which can then be inspected in more detail on the source code level based on information provided by the tree visualization.

Since this work is still in progress, various possibilities exist for future work. For example, our tool currently displays a single tree visualization, depicting the heap's composition at a given point in time, and this visualization is updated when moving through time (a technique called *time traveling*). In the future, we plan to implement a *timeline view*. In this view, based on a number of points in time selected by the user, multiple tree visualizations are generated and displayed next to each other, similar to an interactive version of Figure 3. This should make it even easier for the user to detect growth trends.

6 Acknowledgement

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

References

[1] J. Stasko and E. Zhang. "Focus+Context Display and Navigation Techniques for Enhancing Radial, Space-filling Hierarchy Visualizations". In: *VISSOFT*. 2000, pp. 57–65.

[2] S. T. Barlow and P. Neville. "A Comparison of 2-D Visualizations of Hierarchies". In: *INFOVIS*. 2001, pp. 131–138.

[3] N. Cawthon and A. V. Moere. "The Effect of Aesthetic on the Usability of Data Visualization". In: *IV*. 2007, pp. 637–648.

[4] R. Wettel and M. Lanza. "Visual Exploration of Large-Scale System Evolution". In: *WCRE*. 2008, pp. 219–228.

[5] J. Heer, M. Bostock, and V. Ogievetsky. "A Tour through the Visualization Zoo". In: *ACM Queue* 8.5 (2010), p. 20.

[6] M. O. Ward, G. G. Grinstein, and D. A. Keim. *Interactive Data Visualization - Foundations, Techniques, and Applications*. A K Peters, 2010.

[7] S. Murray. *Interactive Data Visualization for the Web*. O'Reilly Media, 2013.

[8] P. Lengauer, V. Bitto, and H. Mössenböck. "Accurate and Efficient Object Tracing for Java Applications". In: *ICPE*. 2015, pp. 51–62.

[9] I. Bacher, B. M. Namee, and J. D. Kelleher. "Using Icicle Trees to Encode the Hierarchical Structure of Source Code". In: *EuroVis*. 2016, pp. 97–101.

[10] M. Weninger and H. Mössenböck. "User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring". In: *ICPE*. 2018, pp. 115–126.

[11] M. Weninger, E. Gander, and H. Mössenböck. "Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection". In: *ICPE*. 2019, pp. 273–284.

[12] M. Weninger, L. Makor, and H. Mössenböck. "Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor". In: *VISSOFT*. 2020.