AntTracks TrendViz: Configurable Heap Memory Visualization Over Time

Work-In-Progress Paper

Markus Weninger
Institute for System Software, CD Laboratory MEVSS
Johannes Kepler University
Linz, Austria

Lukas Makor
Johannes Kepler University
Linz, Austria

Elias Gander
CD Laboratory MEVSS
Johannes Kepler University
Linz, Austria

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University
Linz, Austria

ABSTRACT

The complexity of modern applications makes it hard to fix memory leaks and other heap-related problems without tool support. Yet, most state-of-the-art tools share problems that still need to be tackled: (1) They group heap objects only based on their types, ignoring other properties such as allocation sites or data structure compositions. (2) Analyses strongly focus on a single point in time and do not show heap evolution over time. (3) Results are displayed in tables, even though more advanced visualization techniques may ease and improve the analysis.

In this paper, we present a novel visualization approach that addresses these shortcomings. Heap objects can be arbitrarily classified, enabling users to group objects based on their needs. Instead of inspecting the size of those object groups at a single point in time, our approach tracks the growth of each object group over time. This growth is then visualized using time-series charts, making it easy to identify suspicious object groups. A drill-down feature enables users to investigate these object groups in more detail.

Our approach has been integrated into AntTracks, a trace-based memory monitoring tool, to demonstrate its feasibility.

KEYWORDS

Memory Monitoring, Heap Growth Analysis over Time, Visualization, Memory Leak Detection

ACM Reference Format:
Markus Weninger, Lukas Makor, Elias Gander, and Hanspeter Mössenböck. 2019. AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *Tenth ACM/SPEC International Conference on Performance Engineering Companion (ICPE '19 Companion), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3302541.3313100

ICPE '19 Companion, April 7–11, 2019, Mumbai, India
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Tenth ACM/SPEC International Conference on Performance Engineering Companion (ICPE '19 Companion), April 7–11, 2019, Mumbai, India*, https://doi.org/10.1145/3302541.3313100.

1 INTRODUCTION

Modern programming languages such as Java use automatic garbage collection. Heap objects that are no longer reachable from static fields or thread-local variables (so-called *GC roots*) are automatically reclaimed by a garbage collector (GC). Nevertheless, memory problems can still occur even in garbage-collected languages. One of the major types of memory problems are memory leaks [4], i.e., objects may remain reachable from GC roots even though they are no longer needed. For example, if a developer forgets to remove objects from their containing data structures, these objects cannot be reclaimed by the GC and will accumulate over time.

State-of-the-art tools, such as VisualVM [14] or Eclipse Memory Analyzer (MAT) [12], perform memory analysis based on a heap snapshot, i.e., a heap dump. They group the live heap objects by their types and display the number of objects and the number of bytes per type in a table, i.e., in a *type histogram.* In addition to that, they support comparing two heap snapshots, showing the increase or decrease of live objects / live bytes per type in a table.

While this information may be sufficient to detect basic memory problems, such analysis approaches have also various shortcomings. First, fixing a memory leak might require more information about the objects besides their types, for example, their allocation sites or the data structures in which they are contained. Second, comparing two snapshots does not reveal general trends in an application's memory behavior. An increase in instances of a certain type between two given points in time does not necessarily indicate a continuous memory growth. To detect trends, the heap has to be compared at multiple points in time, a feature that is not supported by the two tools mentioned. Finally, tools should make it as easy as possible to extract the needed information. Great potential to make data more accessible lies in the use of *data visualization* [5].

In the following, we present a work-in-progress approach on how to visualize continuous memory consumption trends over time. Users can group the heap objects by arbitrary criteria such as their types or their allocation sites and visually inspect the heap evolution per object group. Trends within certain object groups can hint at memory leaks and other memory anomalies. We integrated our approach into AntTracks, a trace-based memory monitoring tool based on the Hotspot Java VM, initially developed by Lengauer et al. [9] and extended by Weninger et al. [18–21].

Our contribution encompasses:

- a technique to derive growth information of heap object groups from memory traces.
- a highly configurable visualization approach for trend analysis. It displays the growth of object groups over time (based on various size metrics) using time-series charts.
- a drill-down feature to re-apply the same visualization approach on a specific subgroup of suspicious objects.
- a working implementation in AntTracks.

2 BACKGROUND

AntTracks consists of two parts: The AntTracks VM, a virtual machine based on the Java Hotspot VM [13], and the AntTracks Analyzer, a memory analysis tool. Since the concepts presented in this paper have been integrated into AntTracks, it is essential to understand how AntTracks works.

Trace Recording and Reconstruction. The AntTracks VM writes information about memory events such as object allocations and object movements executed by the GC into a trace file. It keeps the event size to a minimum and avoids the recording of redundant data [8, 9]. Later, the AntTracks Analyzer can incrementally process such a trace file. It is able to display the overall memory development over time and enables users to reconstruct and inspect the heap state at every garbage collection point [2]. For every heap object, a number of properties can be reconstructed, including its address, its type, its allocation site, the heap objects it references, and the heap objects it is referenced by.

Heap Object Classification. The AntTracks Analyzer uses *object classifiers* in combination with *multi-level grouping* [20, 21] to enable user-driven heap analysis. An object classifier groups heap objects into multiple object groups according to certain criteria such as their types, their allocation sites, or their allocating threads. For example, the *Type* classifier groups objects by their types, e.g., `java.util.LinkedList`. In multi-level grouping, objects are grouped according to the classification results of multiple classifiers. This results in a hierarchical classification tree. In general, every node in such a tree represents an object group and the amount of objects / bytes classified in the respective sub-tree.

For example, the classification tree in Figure 1 has been created by applying the *Type* classifier, followed by the *Allocation Site* classifier. Overall, the classification tree represents 120, 000 objects, 5, 000 of them are of type `Object[]`, and 1, 000 of these arrays have been allocated at `Stack:init()`.

3 APPROACH

This section covers the basic concepts of our new visualization approach. We show how to reconstruct object group growth information from memory traces, together with a highly configurable time-series-based visualization, and how this visualization can be used to drill-down into specific object groups to gain further insights on their growth behavior.

3.1 Gathering Object Group Information

When investigating an application with memory problems, certain parts of its execution trace will stand out. For example, if the memory consumption grew extraordinarily strong between two points in time and does not shrink again afterwards, it indicates that objects have been allocated within this timespan that cannot be reclaimed by the garbage collector later, indicating some type of memory leak. In our approach, users can select such a time window for subsequent analysis.

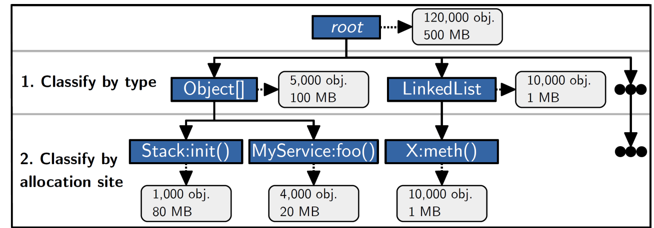

Figure 1: A *classification tree* that first groups all objects by their types and then by their allocation sites.

After selecting the time window of interest, the user has to decide according to which criteria the heap objects should be grouped into object groups. The growth of each object group will later be visualized to aid users in detecting object group growth trends.

Subsequently, the memory trace is incrementally parsed to reconstruct the heap objects within the selected time window. During parsing, the live objects are classified at every garbage collection point using the selected list of classifiers. The resulting classification tree is then stored alongside a timestamp that identifies the respective garbage collection. Every time a new entry is added to this timeline of classification trees, a new time-series data set is generated that is used to update the heap object group memory growth visualization.

3.1.1 *Improvements.*

Instead of classifying the live objects at every garbage collection point within the selected time window, only a subset of the garbage collections can be used for classification to improve performance. The user can select to only classify at every n-th garbage collection, or after at least x seconds have passed since the last classification in the traced application. The benefit is that the classification, which is the most performance-intensive task, can be performed less often. In most cases, existing memory trends will still be apparent.

Future work includes the automatic selection of interesting time windows, thus freeing the user from this task.

3.2 Data Set Generation

To enable growth detection through visualization, the sequence of classification trees first has to be converted into a visualizable data set. The individual classification trees represent the application's heap state at different points in time. Thus, a time-series-based visualization is the most natural choice. Since the time-series plot is the most frequently used form of graphic design [15], it is well-known and easy to understand. In general, time-series data takes the following form: $D = \{(t_1, y_1), (t_2, y_2), ..., (t_n, y_n)\}$ [17], i.e., it consists of data pairs where a given point in time t_i has a certain value y_i assigned to it.

To achieve this format, the nodes on the first level of every classification tree, i.e., the object groups formed by the first classifier, are extracted. Next, a time series is created for every distinct node key. For example, if the objects were first classified and grouped

using the *Type* classifier, every type would become a series in the data set. Each series contains one entry (t, y) per classification tree. t is the timestamp assigned to the respective classification tree, and y is the object group's size, which can be extracted from its tree node within the classification tree.

There are multiple size metrics [18] that users can choose from, either in number of objects or number of bytes:

- *Shallow size*: The number of objects / bytes of an object group, without taking into account any referenced objects.
- *Deep size*: The number of objects / bytes of an object group, including all objects *reachable* from them.
- *Retained size*: The number of objects / bytes of an object group, including all *owned* objects. In other words, it includes all objects that could be freed by the garbage collector if the given object group would be freed.

3.3 Visualization

Depending on the used classifier, the data set can end up containing a large number of series. For example, if the objects have been classified by their types, a series is created for every type, which can easily be several thousands. Yet, most of these series are not of interest when searching for possible memory problems. Thus, our approach supports various techniques to select those series that are of most interest to the user.

To decide which series should be shown, the series are sorted according to a given strategy, and only the top N series are selected. The following sorting strategies are currently supported:

- *Start* and *End* sorting: The series are sorted by their values at the start or the end of the time window, respectively. This way, the memory evolution of those series, i.e., object groups, that take up the most heap space at the start / end of the time window can be inspected.
- *Average* sorting: The series are sorted by their average y-value. This setting can be used to inspect the growth behavior of those object groups that took up the most heap space throughout the selected time window.
- *Absolute growth* and *Relative growth* sorting: The series are sorted by their absolute or relative increase between the start and the end of the time window, respectively. This enables users to inspect the growth behavior of the object groups that grew the most over the selected time window.

The user can also select if an *Other* series should be shown that combines all object groups that are not visualized as separate series. By default, the *Other* series is shown and the *Absolute growth* sorting with an N-value of 5 is selected. An example of the visualization is shown in Section 4.

3.4 Drill-down

As explained in Section 3.2, the initial visualization extracts the first level of the classification trees and visualizes their object groups' growth behavior over time. If multiple classifiers have been applied to build the classification trees, a suspicious object group (e.g., a group with a strong growth within the selected time window) can be selected for *drill-down* in the visualization. The drill-down feature re-applies the same visualization technique to the children of the selected object group and displays it in a new chart.

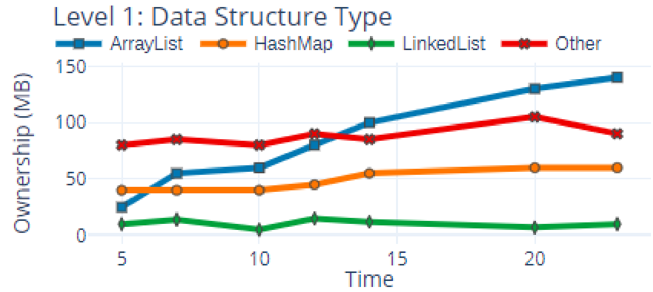

Figure 2: Visualizing the *retained size* of data structure types over time highlights `ArrayList` as suspect.

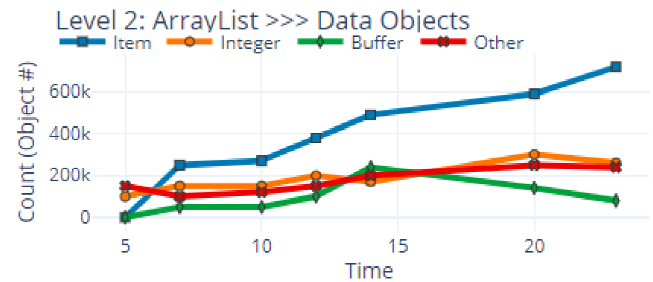

Figure 3: Drill-down showing the development of the *object count* of `ArrayList`'s data objects.

For example, assume that the *Type* classifier has been used as the first classifier, followed by the *Allocation Site* classifier, creating classification trees similar to the one shown in Figure 1. If the user detects suspicious growth for objects of type `Object[]`, this group can be selected for drill-down. A new data set will be created, based on the various allocation sites at which objects of type `Object[]` have been created, according to the steps explained in Section 3.2. This allocation site data set will then be visualized below the existing chart, using the steps described in Section 3.3.

The various settings, such as the sorting strategy, can be adjusted individually per chart. To make browsing the charts more convenient, interaction features such as zooming are synchronized over all charts.

4 EXAMPLE

In this example, we show a typical way of how to use our visualization approach by demonstrating how to identify and inspect data structure types with growing ownership in AntTracks.

First, we select a time window over which the application's memory grew considerably. In this example, the basic idea is to (1) first inspect the *retained size growth*, i.e., ownership growth, of data structure types (such as `HashMap`s), (2) then selecting the type with the highest growth for drill-down, (3) followed by a visualization of the data object growth within this data structures to find out which data objects accumulate the most.

Our example uses one filter and two classifiers to group the heap objects for visualization. We are using the *Data Structure* filter, which only includes data structure head objects (for example lists, maps, etc.) during classification, ignoring other objects. These data structure head objects are then classified by their types. Visualizing the retained size growth, i.e., the ownership growth, of these types results in a chart similar to the one in Figure 2, which shows that

objects of type `ArrayList` have the strongest retained size growth. Thus, this type is selected for drill-down.

As a second classifier we are using the *Data Object* classifier which enables us to analyze the data objects stored in a data structure. Due to paper length restrictions, we group those data objects only by their types. Typically, they would also be grouped by their allocation sites. In Figure 3, the drill-down on the `ArrayList` object group was configured to show the growth of the number of data objects in `ArrayLists` per data object type.

We are now able to easily pinpoint `ArrayList` data structures that contain `Item` objects as the major suspects for a possible memory leak. This information could now be used to investigate the memory problem on the source code level.

5 RELATED WORK AND FUTURE WORK

State-of-the-art tools include, among others, VisualVM [14] and Eclipse Memory Analyzer (MAT) [12], which have been discussed in Section 1.

In their work on the taxonomy and classification of memory analysis approaches in Java, Šor and Srirama [23] highlight the visualization approaches by De Pauw and Sevitsky [3, 10] and by Reiss [11]. The former extracts reference patterns (repetitive reference sequences in a heap object graph) and visualizes them. In addition to that, such reference patterns can also be extracted for those objects that are created between two heap snapshots (e.g., potentially leaking objects), which can then be visually explored. The latter work visualizes the object ownership in a tree-like visualization using shapes, coloring, hatching, hue and saturation. Another approach that compares heap snapshots has been developed by Jump and McKinley [6, 7]. Their tool, Cork, compares the heap object graph structure of two heap snapshots to detect the growth of certain reference patterns between classes.

Future work includes a more thorough evaluation and presentation of our new visualization approach based on real-world scenarios. In addition to that, the approach can still be extended by numerous features. For example, the analysis time windows could be chosen automatically by the tool. Besides the current visualization using line charts, other visualization techniques could also be evaluated based on the same underlying data, such as *small multiples* [16] or as *software cities* [22]. Other typical visualization techniques that can still be further explored involve the representation of aggregated heap objects as graphs [1].

6 CONCLUSION

In this work, we presented a new approach to visualize the growth of heap object groups over time. Trends detected in this visualization can hint at memory problems such as memory leaks involving certain object groups. To construct the underlying data for the visualization, the live heap objects are split into groups based on user-selected criteria (e.g., by their types) at multiple points in time. The evolution of each group over time is then visualized in a time-series chart. The visualization is highly user-configurable based on the user's needs, allowing users to select features such as series sorting, series selection or size metrics. A drill-down feature enables users to select an object group of interest, e.g., a strongly growing group, to classify the objects within this group by another criterion, and to re-apply the same visualization technique to this group.

ACKNOWLEDGMENTS

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] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proc. of the 5th Int'l Symp. on Software Visualization (SOFTVIS '10)*.

[2] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proc. of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*.

[3] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proc. of the European Conf. on Object-Oriented Programming (ECOOP '99)*.

[4] Mohammadreza Ghanavati, Diego Costa, Artur Andrzejak, and Janos Seboek. 2018. Memory and Resource Leak Defects in Java Projects: An Empirical Study. In *Proc. of the 40th Int'l Conf. on Software Engineering: Comp. Proc. (ICSE '18)*.

[5] Jeffrey Heer, Michael Bostock, and Vadim Ogievetsky. 2010. A Tour Through the Visualization Zoo. *Commun. ACM* 53, 6 (June 2010).

[6] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '07)*.

[7] Maria Jump and Kathryn S. McKinley. 2009. Detecting Memory Leaks in Managed Languages with Cork. *Software: Practice and Experience* 40, 1 (2009).

[8] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proc. of the 13th Int'l. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*.

[9] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE '15)*.

[10] Wim De Pauw and Gary Sevitsky. 2000. Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency: Practice and Experience* 12, 14 (2000).

[11] S. P. Reiss. 2009. Visualizing The Java Heap to Detect Memory Problems. In *5th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '09)*.

[12] Eclipse Foundation. 2018. Eclipse Memory Analyzer (MAT). https://www.eclipse.org/mat/

[13] Oracle. 2018. The HotSpot Group. http://openjdk.java.net/groups/hotspot/

[14] Oracle. 2018. VisualVM. https://visualvm.github.io/

[15] Edward R. Tufte. 2007. *The Visual Display of Quantitative Information (2nd edition)*. Graphics Press, Cheshire, CT, USA.

[16] Stef van den Elzen and Jarke J. van Wijk. 2013. Small Multiples, Large Singles: A New Approach for Visual Data Exploration. *Comput. Graph. Forum* 32 (2013).

[17] Marc Weber, Marc Alexa, and Wolfgang Müller. 2001. Visualizing time-series on spirals. In *Infovis*, Vol. 1.

[18] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *Proc. of the 15th Int'l Conf. on Managed Languages & Runtimes (ManLang '18)*.

[19] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proc. of the 10th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '19)*.

[20] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proc. of the 8th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '17)*.

[21] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proc. of the 9th ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '18)*.

[22] Richard Wettel and Michele Lanza. 2007. Visualizing Software Systems as Cities. In *4th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '07)*.

[23] Vladimir Šor and Satish Narayana Srirama. 2014. Memory leak detection in Java: Taxonomy and classification of approaches. *Journal of Systems and Software* 96 (2014).