STAG: Smart Tools and Applications in Graphics (2020)
S. Biasotti, R. Pintus and S. Berretti (Editors)

Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations

M. Weninger[1] and L. Makor[1,2] and H. Mössenböck[1]

[1]Johannes Kepler University Linz, Institute for System Software, Austria
[2]Johannes Kepler University Linz, Christian Doppler Laboratory MEVSS, Austria

Abstract

Memory leaks occur when no longer needed objects are unnecessarily kept alive. They can have a significant negative performance impact, leading to a crash in the worst case. Thus, tool support for heap evolution analysis, especially memory leak analysis, is essential. Unfortunately, most memory analysis tools lack advanced visualizations to facilitate this task.
In this paper, we present an approach to use well-known tree visualization techniques for memory growth visualization. Our approach groups heap objects into memory trees *based on a user-defined set of properties such as their types or their allocation sites at multiple points in time. We present two novel approaches to inspect how these trees evolve over time: In our* time-travel-based *visualization, a single space-filling tree visualization shows the monitored application's heap memory at a given point in time. Users can step back and forth in time, causing the visualization to update itself. In our* timeline-based *visualization, a time-series chart depicts the overall memory consumption over time. Above this chart, multiple memory tree visualizations are shown side-by-side for a number of user-selected points in time. Using these techniques to visually inspect the evolution of the heap over time should enable users to gain new insights and to detect (problematic) memory trends in their applications.*
To demonstrate the feasibility and applicability of the presented approach, we integrated it into AntTracks, a trace-based memory monitoring tool and applied it in two memory leak case studies.

CCS Concepts
•General and reference *$\rightarrow$ Performance;* ***•Software and its engineering*** *$\rightarrow$ Software performance; Software maintenance tools;* ***•Information systems*** *$\rightarrow$ Data analytics; Information extraction;* ***•Human-centered computing*** *$\rightarrow$* ***Interactive systems and tools; Visualization techniques; Visual analytics; Information visualization;***

1. Introduction

Many modern programming languages such as Java use a garbage collector (GC) to automatically reclaim heap objects that are no longer reachable from static variables or thread-local variables (i.e., GC roots). Even though automatic memory management prevents certain memory-related mistakes, various problems can still occur. Memory leaks are very common defects [GCS*20] and occur when objects remain reachable even though they are no longer needed. For example, a developer may forget to correctly clear a long-living data structure. Consequently, its objects cannot be reclaimed by the GC and thus accumulate over time [WGM18a, WGM19a].

Memory leaks can have a significant performance impact, leading to a crash in the worst case. Therefore, it is essential to provide tooling for memory analysis. Yet, existing tools have two major drawbacks: most of them (1) inspect the heap only at a single point in time and (2) do not use advanced visualizations. For example, existing tools such as VisualVM [Ora20] or Eclipse Memory Analyzer (MAT) [Ecl20] use heap dumps to inspect the heap at a single point in time. Yet, to detect and inspect trends in the memory behavior, which is needed to investigate memory leaks, the heap has to be compared at multiple points in time [WMGM19]. Those tools that do support memory evolution analysis often present the raw data in tables and do not employ visualization techniques. This is unfortunate, since domains such as software evolution and program comprehension have shown that using graphical means can help users in understanding and interpreting systems and their growth [CZvDR09, WLR11, FKH15, FFHW15, FKH17, BAB18].

In this paper, we tackle both of the mentioned problems by presenting an approach to visualize the heap memory evolution over time using tree visualizations. In order to create useful heap visualizations, the heap objects have to be brought into a suitable structure first. Many tools group heap objects based on a certain property (e.g., type or allocation site). Using multiple grouping criteria results in a hierarchical grouping structure, i.e., a memory tree [WLM17, WM18]. We record memory traces that allow us to create memory trees of a monitored application at multiple points in time, each representing the state of the heap at a certain point in time. In this work, we present a *time travel* visualization approach that enables users to step through the individual points

https://doi.org/10.2312/stag.20201241

in time, as well as a *timeline* visualization approach that visualizes the heap at multiple points in time in juxtaposition side-by-side. This way, we enable users to recognize trends in the memory behavior of the monitored application to identify accumulating object groups, thereby gaining new insights that help to locate and resolve memory leaks. The contributions of this paper are:

- a suggestion of tree visualizations suitable for memory visualization based on a requirements catalog (Section 3).
- an approach to visualize a single heap state using the previously selected tree visualizations (Section 4).
- two novel approaches to visualize the evolution of the heap over time: the *time-travel-based* approach (Section 5.3) and the *timeline-based* approach (Section 5.4).
- an implementation of the presented techniques in the memory monitoring tool AntTracks (Section 6).
- two memory leak analysis case studies that demonstrate the approach's feasibility and applicability (Section 7).

2. Background

This section explains how our approach collects memory data and how this data is transformed to be suitable for visualization. As the approach has been integrated into AntTracks, this section gives a short overview of the tool. AntTracks consists of two parts, the *AntTracks VM* [LBM15,LBF*16,LBM16], a virtual machine based on the Java Hotspot VM, and the *AntTracks Analyzer*, a trace-based memory analysis tool [WLM17, WM18]. We chose this tool since its source code is publicly available [Wen20] and the authors already had prior experience with its code base.

2.1. Trace Recording

While heap dumps are good enough to perform analyses at a single point in time, they fall short compared to continuous memory tracing approaches when performing analyses over time [WGM19a]. Thus, the AntTracks VM records memory events such as object allocations or objects moves during garbage collection and stores them in a trace file [LBM15]. This approach introduces a run-time overhead of about 5%, but provides more fine-grained memory information than heap dumps. To keep the size of the trace file low, the AntTracks VM tries to avoid storing redundant data [LBM16].

2.2. Heap State Reconstruction and Memory Trees

The AntTracks Analyzer uses the recorded trace file as input to reconstruct the memory data and provides various features to analyze this data. By incrementally parsing the recorded trace file, the tool is able to reconstruct a heap state for each garbage collection point [BLM15]. A heap state is a set of heap objects that were live in the monitored application at a certain point in time. Properties such as the address, type, allocation site, and allocating thread can be reconstructed for each heap object.

One of the core features of the AntTracks Analyzer is to perform object classification and multi-level grouping [WLM17, WM18]. This approach groups the heap objects into a hierarchical memory tree based on a user-defined set of criteria (called *classifiers*), e.g., based on their types, allocation sites, or allocating threads. Every node in such a tree represents a set of objects that share the same properties, also called an object group. An exemplary memory tree created using the *allocating thread* classifier and *type* classifier is depicted in Figure 1. The root node of the memory tree represents the whole heap, every node represents an object group. For example, the node *T1* represents objects 0, 1 and 2, which were allocated by thread T1. The *Integer* node below *T1* represents objects 0 and 1, which were allocated by thread T1 *and* are of type `Integer`.

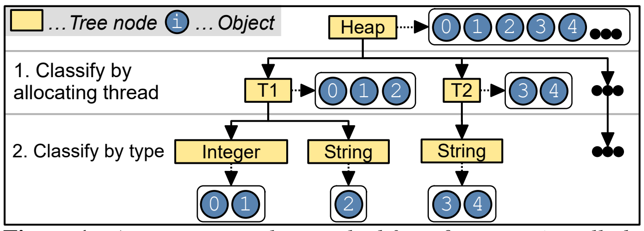


Figure 1: *A memory tree that resulted from first grouping all objects by their allocating threads and then by their types.*

3. Tree Visualizations

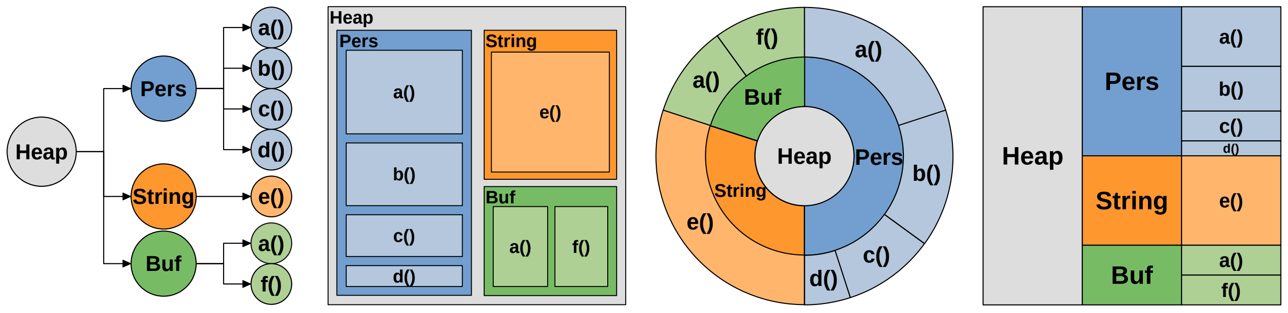
Memory traces enable reconstructing a vast amount of information about all heap objects that have been live at some point in time in the monitored application. Since presenting this data with raw numbers would most probably overwhelm the user, we chose tree visualizations as a means of data visualization [HBO10]. Data visualization can help to convey information faster [KK91, WGK10] and can facilitate the identification of patterns [War13, Mur13] which can lead to new insights [War13]. This section discusses different properties of tree visualization and presents a requirements catalog we used to select two tree visualizations that seem to be adequate for an interactive heap evolution visualization.

3.1. Tree Properties and Requirements Catalog

Tree visualizations are the most common type of visualization to depict hierarchies such as memory trees [SZ00]. Consequently, ample research has been performed on tree visualizations, which resulted in a vast number of tree visualization techniques. For example, `treevis.net` [Sch11] lists more than 300 publications on tree visualizations. However, not all tree visualization techniques are suitable for visualizing the heap memory evolution over time.

In general, different tree visualizations use different approaches to display a tree's *content information* as well as its *structural information*. *Content information* associates data of the underlying tree nodes to visual attributes of the nodes (such as node size, color, or transparency). *Structural information* concerns the tree's hierarchy and can either be expressed *explicitly* or *implicitly* [JS91] . Explicit visualizations, also called node-link visualizations, use explicit graphical elements such as lines between nodes to indicate relationships. Implicit visualizations, also called space-filling visualizations, indicate the relationships of the nodes via spatial arrangement, e.g., containment [SS06]. Various works provide further and more detailed taxonomies on tree visualizations [Sch11, STLD20].

To identify tree visualizations suitable for memory evolution visualization, we define four requirements that are vital to help users in gaining insights into the heap's evolution over time.



(a) *Node-link diagram.* **(b)** *Treemap.* **(c)** *Sunburst.* **(d)** *Horizontal icicle.*

Figure 2: *Four different tree visualization techniques depicting the same memory tree.*

1. A node's content information (e.g., its number of heap objects) should at least be represented by size, following the visualization metaphor *more is bigger* [Lak94, HHK*17].
2. Updating hundreds of nodes should be feasible in real-time since our approach revolves around an interactive visualization. It must not involve inconvenient latency since this complicates making observations and drawing generalizations [LH14]. Furthermore, it has been shown that users tend to use an interactive system less often if it exhibits interaction latency [Bru09].
3. Since we want to make the tool accessible to novice users, the visualization has to be easy to understand, even for inexperienced users. Explicit explanation should not be necessary. Thus, we do not search for novel and experimental visualization techniques, but rather want to use visualizations that have been empirically proven to be effective, efficient and easy to use.
4. The tree visualization has to support a stable layout of evolving data over time. As Hahn et al. [HTMD14] state *layout stability is considered essential for [...] visual analysis tasks such as comparing hierarchies and attributes of such hierarchies' nodes, and tracking changes to hierarchies over time*, which are common tasks in memory analysis. When visually exploring data, users create *cognitive maps* that are based on spatial relations and attributes of the presented data [Kit94]. Consequently, we look for visualizations that support a stable layout, i.e., a layout that requires few changes to the user's cognitive map as the spatial relations mostly stay intact when updating the visualization.

3.2. Exemplary Tree Visualizations

Figure 2 shows four different tree visualizations, all of which depict the same tree. This memory tree was generated by grouping all heap objects by their types, and all objects of the same type by their allocation sites. The gray `Heap` root node represents the whole heap, nodes on the first level represent different types. We can see that the heap consists of objects of the types `Pers` (blue), `String` (orange), and `Buf` (green). In Figure 2b through Figure 2d we can further see that 50% of the heap is taken up by objects of type `Pers`. On the second level we can see that the `Pers` objects have been allocated in four different methods, most of them in method `a()`.

3.3. Selection of Tree Visualization Techniques

Most *explicit* visualizations (such as the node-link diagram in Figure 2a) do not use variable-sized nodes, making it hard to distinguish nodes that represent few or many objects. As this contradicts our first requirements, no explicit visualizations were chosen for our approach.

We also excluded visualizations that require complex layout calculations, as they cannot be updated fast enough to not disturb the users during analysis (requirement three). Examples for excluded visualizations encompass *variational circular treemaps* [ZL15] or *GosperMaps* [AHL*13], since calculating their layout based on a few hundred nodes already takes seconds.

To fulfill the third requirement, we explored existing study results. Barlow and Neville [BN01] performed two experiments to compare the performance of icicle visualizations, tree ring visualizations (a visualization similar to the sunburst visualization) and treemap visualizations. They found that icicle and tree ring both worked quite well, while treemap worked significantly worse than the other tree visualizations. Cawthon and Moere [CM07] conducted an online survey to evaluate the aesthetics and task performance of eleven visualization techniques. With regard to aesthetics, sunburst was the clear winner, while in terms of correctness and response time, both sunburst and icicle were among the best techniques. Treemap again ranked among the worst techniques.

The fourth requirement, i.e., that the chosen visualizations have to support stable layouts, is discussed in more detail in Section 5.1.

As icicle and sunburst fulfill all our requirements and consistently ranked among the best tree visualization techniques in the discussed studies, both were chosen to be used for our heap evolution analysis. Consequently, we chose to not include treemaps due to their negative study results. Nevertheless, treemap algorithms are still useful. For example, they are successfully used to generate layouts for *software maps* and *software cities* [LSDT19] such as *CodeCity* [WL07, WLR11], *SynchroVis* [WWF*13], *ExplorViz* [FKH17], or *Memory Cities* [WMM19, WMM20].

3.4. Chosen Tree Visualizations

This section shortly explains the *sunburst* and *icicle* visualizations that were chosen to be part of our heap visualization approach.

Sunburst As shown in Figure 2c, sunburst is a radial space-filling visualization [SZ00]. In a sunburst, the root node of the hierarchy is depicted as a circle in the center of the visualization. This circle is surrounded by multiple levels of circular ring segments where each

https://doi.org/10.2312/stag.20201241

ring segment represents a tree node. The tree hierarchy is moving outwards, i.e., each tree level is further away from the center. The sunburst visualization is adjacency-based, meaning that the children of a node are positioned next to each other on the next level within the angular sweep of their parent. The angular size of the segment is based on an attribute of that node, in our case the number of objects or bytes of the respective heap object group.

Icicle Icicle is another space-filling visualization [HBO10]. Similar to sunburst, it is adjacency-based, meaning that the children of each node are positioned next to each other on the next level within the extent of their parent. In a horizontal icicle [BNK16] (as shown in Figure 2d), each rectangle has the same width and its height is based on some attribute of its respective tree node.

4. Heap State Visualization

A memory tree, i.e., the result of grouping heap objects based on common properties, is the basis for our heap state visualization. Figure 3 shows an example of the same memory tree being displayed in our tool, once as an icicle and once as a sunburst.

Problems that can arise are that memory 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. This section discusses these techniques in more detail.

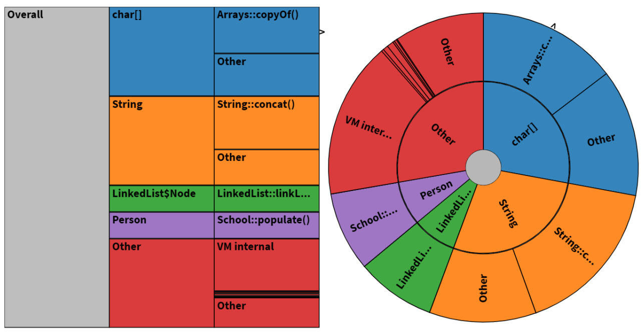


Figure 3: *Screenshots of a memory tree visualized in our tool, once as an icicle and once as a sunburst.*

4.1. Tree Pruning

The goal of tree pruning is to keep only the most significant nodes and to hide less important ones. In memory leak analysis, we are interested in nodes that represent large object groups, as their objects potentially accumulated due to a memory leak. Thus, unimportant smaller object groups can be merged. In our implementation, 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 artificial *"Other"* node.

4.2. Drill-Down

As the screen space is limited, it is neither feasible nor reasonable to display the full hierarchy of deep trees. Consequently, we decided to only display two levels below the root node by default, with the possibility to *drill-down* into deeper tree branches. By clicking on a non-leaf node, the selected node becomes the new root of the visualization. Figure 4 depicts an icicle that was created using the *allocating thread*, *type* and *allocation site* classifiers. On the left, the visualization is shown without drill-down. On the right, the drill-down has been performed on the node *Thread 2*. The orange and green rectangles highlight the node selected for drill-down as well as its children. Additionally, the allocation sites of the objects that were allocated by *Thread 2* are now shown in the drilled-down view. While in a drill-down, the user can click on the root node to step up one level again.

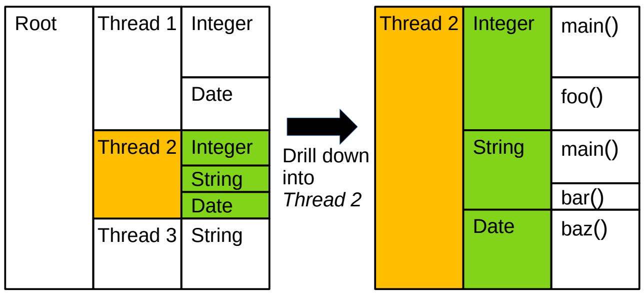


Figure 4: *Drilling down into a node in the initial icicle (left) results in the selected node becoming the root of the visualization (right).*

4.3. Local View and Global View

Showing only a limited number of tree levels enables us to display the shown nodes with reasonable size. Thus, also more room is available for text within the nodes (e.g., type names or method names). However, limiting the number of shown levels comes at the cost of losing the hierarchy overview, e.g., how many levels really exist. Users may also possibly lose track of their current drill-down position within the hierarchy after multiple drill-down steps, putting potential additional cognitive load on the user [TM04].

To tackle these problems, we use two synchronized visualizations next to each other. The first one, called the *local view*, only displays the currently selected node and its two sublevels (as discussed before). Additionally, a second view called the *global view* displays the full hierarchy, independent of the tree depth or the currently selected drill-down node. An example for this can be seen in Figure 5, where an icicle is shown that was generated using the *allocating thread*, *type* and *allocation site* classifiers. The *local view* on the left shows the *Thread 2* node (that has been selected via drill-down) as the root as well as its two direct sublevels. The *global view* on the right shows the full hierarchy, i.e., all tree levels, with the drill-down node highlighted. The other tree branches are set to semi-transparent. Having these two visualizations next to each other solves the problems of losing track in the overall hierarchy. To further make orientation easier, in addition to highlighting the currently selected node, when the user hovers a node in the local view we also highlight the respective node in the global view. This makes it easier to spot its position within the hierarchy. Local and global view (including an example) will be revisited in Section 5.3.

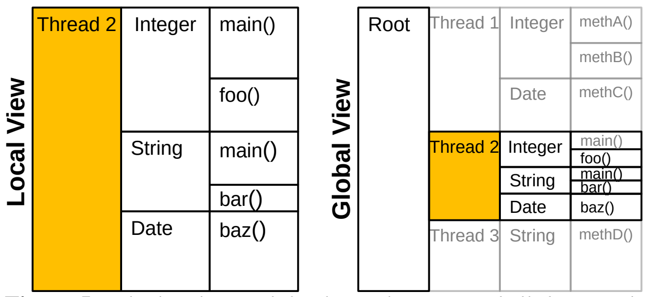


Figure 5: *The local view (left) shows the current drill-down node as root and two more levels, while the global view (right) shows the whole hierarchy with highlighted drill-down branch.*

5. Heap Evolution Visualization

Visualizing the heap evolution means to not only visualize the state of the heap at a single point in time, but to visualize its evolution across multiple points in time. To narrow down the search space, users can select a time window of interest for visualization [WGM19b]. Within the selected time window, a user-selected list of classifiers is used to group all live heap objects into memory trees at multiple points in time. This results in a sequence of memory trees, where each tree represents the heap state of the monitored application at a certain point in time. In this section, we discuss preprocessing steps to achieve a stable layout of the tree visualizations across multiple points in time, as well as a technique to better visualize absolute growth. Following, we present two novel approaches to inspect the evolution of these trees: the *time-travel-based* approach and the *timeline-based* approach.

5.1. Stable Layout

Visualizations that show the evolution of a system have to be carefully designed. A major risk is that a small change in the underlying data can result in vastly different layouts being generated. For example, between two points in time it can happen that the order of the tree nodes would change. Imagine two sibling nodes that are ordered by size: when their size, e.g., heap object count, changes, the order of their nodes changes as well. Such a behavior would make it unnecessarily hard to keep track of the evolution of different tree nodes. Thus, it is of utmost importance to ensure that our tree visualizations exhibit a stable layout across all points in time.

Static Position Animation One way to achieve a stable layout is the *static position animation* approach [LSP08, WL08]. In it, all visual elements stay in the same place throughout the whole evolution and just grow and shrink within a fixed area reserved for them. This area is calculated based on the maximum size that the element will reach at any point in time. A downside of this approach is that it wastes lots of space when an element is not at its maximum size (or worse, not shown at all). Also, it may work well for certain visualizations such as treemaps but not for most other tree visualizations. For example, applying this approach to an icicle would lead to empty spaces between the rectangles if they are not at their maximum size. As such a layout might rather distract than help users, we decided to implement a relaxed version of it that we call *relative position animation*.

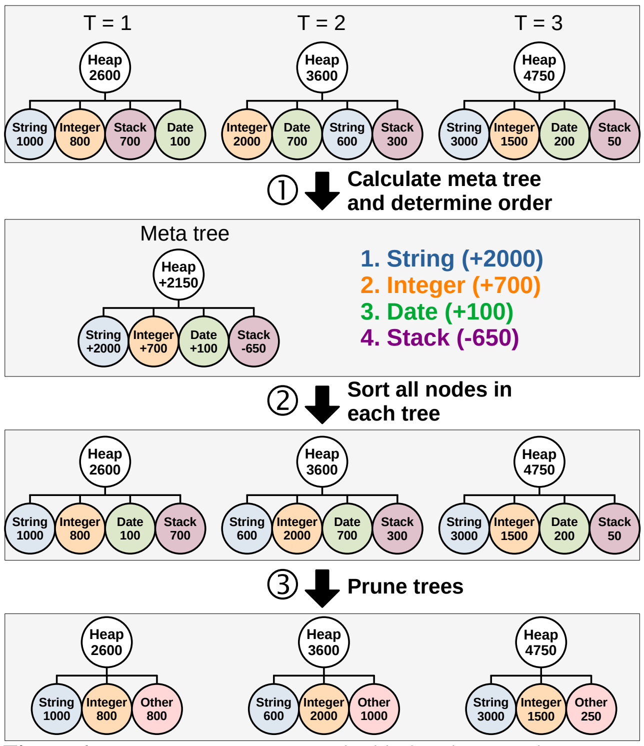


Figure 6: *Preprocessing steps applied before heap evolution visualization: (1) meta tree calculation, (2) sorting, and (3) pruning.*

Relative Position Animation Relative position animation means that the absolute position of a tree node might change when stepping through time, but the *order* of the nodes will stay the same. This is achieved by assigning a sort position (based on a certain criterion) to each node. This sort position is fixed across all points in time and is calculated for all nodes *once* after all memory trees have been computed. Weninger et al. [WMGM19] already used a similar concept in another memory evolution visualization. There, they used various sorting strategies, including *start size* sorting, *end size* sorting and *absolute growth* sorting, which we also support for our tree visualizations. When applying the start size or the end size sorting strategy, all nodes in all trees are sorted by their object count or byte count at the start or end of the inspected time window respectively. Yet, we found that the *absolute growth* sorting strategy is usually the most useful strategy for investigating memory leaks. When applying the absolute growth sorting strategy, the absolute growth of each node between the first and the last point in time is calculated and stored in a meta tree. The first step in Figure 6 depicts how such a meta tree is calculated based on the nodes' absolute growth and how the meta tree is used to create a fixed sort order. In the second step of Figure 6, all nodes in each tree are sorted based on this sort order.

Tree Pruning Revisited In Section 4.1, we described tree pruning for a single memory tree by keeping the largest tree nodes. When pruning trees for heap evolution visualization, this pruning is slightly adjusted. Now, those nodes that have been ranked first *based on the sorting strategy* are preserved (instead of selecting them based on the current size). This ensures as few node additions and removals as possible between multiple points in time, while

	T=1	T=2
Heap	100	200
— A	50	60
— B	30	60
— C	20	80

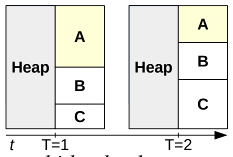


Figure 7: *Unscaled visualizations may hide absolute growth.*

	T=1	T=2
Heap	100	200
— A	50	60
— B	30	60
— C	20	80

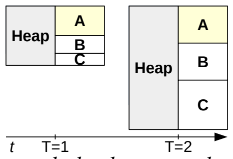


Figure 8: *Scaled visualizations reveal absolute growth.*

still benefiting from tree pruning. In the example in Figure 6, the sorted trees are pruned in the third step by preserving the first two nodes per tree, even though they might not be the biggest ones at that point in time (as is the case for *String* in the tree at $T = 2$).

5.2. Absolute Growth Visualization

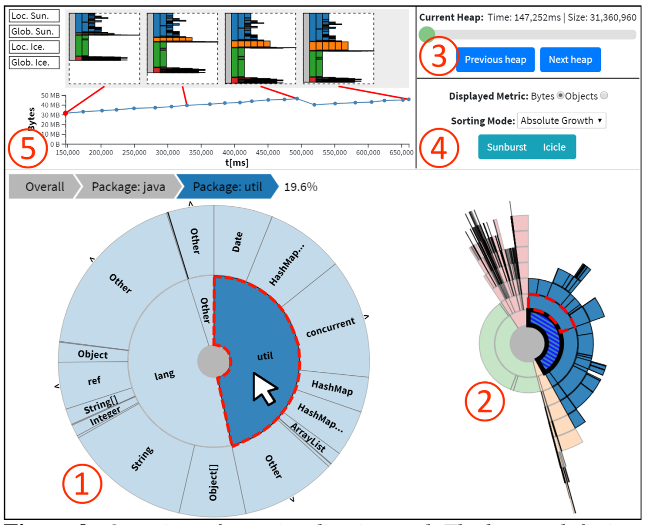
A problem of our chosen tree visualizations is that they use the same amount of screen space to visualize any heap, independent of the heap's size. Figure 7 illustrates this problem. At time $T = 1$, the heap has a size of 100MB, and at time $T = 2$ it has a size of 200MB, i.e., it doubled in size. Yet, the resulting visualizations do not reflect this. For example, by only looking at the tree visualizations, one may think that the number of `A` objects shrank between the two points in time, while the contrary is the case. Thus, we support to display *scaled* visualizations, as shown in Figure 8. For example, icicles are scaled along the y-axis based on the heap size at the respective point in time, where the largest heap within the analyzed time window uses the whole height. This should make it easier for users to comprehend the absolute growth of the heap over time.

Unscaled visualizations, i.e., visualizations that use the whole available space, are especially useful when text should be displayed. This is the reason why we use an unscaled visualization as the local view of our tool, which will be explained in more detail in Section 5.3. Scaled visualizations are more helpful when comparing the heap state of a system at multiple points in time, thus the timeline-based approach (which will be explained in more detail in Section 5.4) uses a scaled visualization by default.

5.3. Time-Travel-based Approach

In the context of visualization, Wettel and Lanza [WL08] define *time traveling* as stepping back and forth through time while the visualization updates itself to reflect the current state. Other memory evolution visualization approaches [WMM19, WMM20] also successfully used time travelling as their means of evolution visualization, which inspired us to also use this interaction technique for our memory tree evolution visualization.

Figure 9 shows a snapshot of our tool. On the bottom half the tool shows our time-travel-based visualization. It shows the heap at the currently selected point in time, once in (1) local mode and once in (2) global mode. Users are provided with (3) buttons to go to the next and the previous heap state, as well as a slider to



Current Heap: Time: 147,252ms | Size: 31,360,960

Displayed Metric: Bytes ● Objects

Sorting Mode: Absolute Growth

Figure 9: *Overview of our visualization tool. The bottom-left visualization shows (1) the drilled-down* local view *of the heap, the bottom-right visualization shows (2) the* global view *with highlighted drill-down node and highlighted hover node. On the top right, the (3) time controls and the (4) visualization options can be found. Beside them, the (5)* timeline-based *visualization is situated.*

move through time. When stepping through time, the visualization updates itself to show the memory tree at the given time. To make these updates more appealing, we leverage various animation features. For example, when the visualized data changes, the existing nodes do not snap to their new location but their positions and their sizes are gradually adjusted to match their new values, an approach called *tweening* [Wil09]. If a node was selected for drill-down, this selection is also preserved when stepping through time. At any time during analysis, the user (4) can switch between the different visualizations, i.e., sunburst and icicle (beside being able to change other settings). The visualizations are synchronized, i.e., the current drill-down node (if any) and all user-chosen settings such as metric (i.e., either object count or byte count) are preserved. Thus, users are able to continue at the exact same state of the analysis at which they were before they switched the visualization type.

5.4. Timeline-based approach

While the time-travel-based approach displays only one tree visualization (or more specifically, two, i.e., the local and the global view) at a time, the timeline-based visualization displays multiple tree visualizations, each representing the heap at a different point in time, in juxtaposition side-by-side. The name *timeline-based* stems from the fact that users can select which points in time to visualize by selecting them on a time-series chart, i.e., on a timeline. By comparing these tree visualizations with each other, the user should be able to detect changes over time in the heap composition.

In the upper left part of Figure 9 at (5), our timeline-based visualization can be seen. We display the overall heap consumption in a time-series line chart with clickable data points. Clicking them toggles the visualization of the tree visualization of the heap at the respective point in time. The tree at the currently selected point in time is always shown in the timeline. Since the visualizations in the timeline view are quite small, it is not possible to display reasonable sized text within the node elements. Nevertheless, comparing the visualizations with each other quickly provides insight into the general evolution of the heap. Thus, the timeline-based visualization (overview) and the local view of the time-travel-based visualization (detailed analysis) complement each other well.

Figure 10 shows another example of the timeline view with four icicle visualizations, each depicting the heap state at a different point in time. The trees have been generated using the *type* classifier, followed by the *allocation site* classifier. This example uses *scaled* icicle visualizations, as explained in Section 5.2. Thus, the growth of the grey rectangle reflects the heap's overall growth. As the first level represents objects of different types, we can see that at any point in time the heap mostly consists of four different types. Looking at the second level, we can see that the blue and the orange types have two different allocation sites each (where one of the two created far more objects than the other one), while the green and violet objects all have been allocated within a single method each.

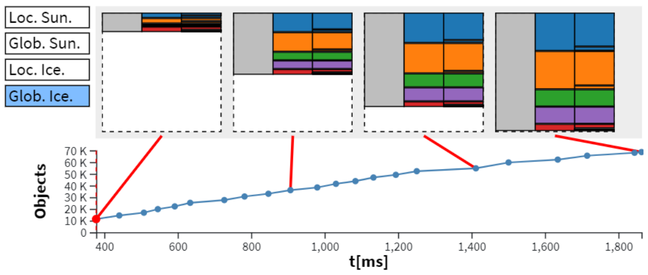


Figure 10: *The timeline view shows the evolution of the heap over time by showing its state at multiple points in time in juxtaposition.*

6. Implementation Details

We implemented our approach as a Javascript web application that heavily uses the *D3.js* library. This library provides many utility functions to create hierarchical visualizations [BOH11, CPRG16]. The implemented web application was integrated into AntTracks, which is a JavaFX-based application, using a JavaFX *WebView*. After loading a trace file, the user can select a time window and a list of classifiers which are used to classify the heap at multiple points within the selected time window. Subsequently, the resulting sequence of memory trees is converted to *JSON* and sent to the tree visualization web application via *WebSockets* [FM11, WPJR11]. Using this JSON interface, our visualization tool could also be used by other monitoring tools than AntTracks. A prototype of the tool can be found at `http://bit.ly/STAG-MemoryTreeVizTool`. This prototype also contains the data used in the following two case studies. A video explaining the tool can be found at `http://bit.ly/STAG-MemoryTreeVizVideo`.

7. Case Studies

To demonstrate the feasibility of our approach, we searched online for real-world applications that contain memory leaks to showcase how to investigate them. In the following, we present the analysis of a memory leak in the *Commons HttpClient* library, as well as the analysis of a memory leak in the *Dynatrace easyTravel* application.

7.1. Commons HttpClient

Finding applications or libraries that contain memory leaks requires lots of effort, since their source code and the needed build tools have to be publicly available. To find the memory leaking library discussed in this section, we browsed Apache's issue tracker [Apa20] for the keyword *leak*. This way, we found an old issue regarding a memory leak in the *Commons HttpClient* library, a library that can be used to send HTTP requests. As the library was completely unknown to us authors, it seemed like a good example to check if our tree visualizations are useful to detect accumulating objects even in an unknown application. We downloaded the affected version 3.0.1 [Apa06] and built a small driver application [WM20], which creates HTTP connections in multiple batches, where in each batch 10,000 connections are created and deleted shortly thereafter.

To analyze the memory behavior of our driver application, we recorded a memory trace and inspected it using AntTracks. One would expect to see spikes in the memory usage, as it should go up when the connections are created and should go down after their deletion. Yet, contrary to this expectation, Figure 11 shows that the memory consumption continuously rose. It seems as if only a part of the objects that were allocated during every batch are actually garbage-collected afterwards. To create the tree visualizations in Figure 12 through Figure 14, we grouped the heap objects by *type* and *allocation site* and sorted them by *absolute growth*.

We started our analysis by comparing the first two sunbursts in Figure 12. The first sunburst depicts the heap at the first memory consumption peak, while the second sunburst depicts the heap at the first dip. What immediately catches one's eye is that the percentage of the heap that is occupied by the red type (i.e., the memory tree's artificial *Other* node) shrank significantly between the peak and the dip, while the relative amount of memory occupied by objects of the other types grew. Looking at the next two sunbursts we can see that this trend continues. In the end, around 90% of the heap are occupied by objects of only six different types. The local view of the time-travel-based visualizations at this point in time is also shown in Figure 13. All of these types except `HostParams` (brown) are allocated at a single allocation site each (since all types only have a single node on the second level).

To better grasp how the absolute sizes of the nodes develop over time, the *scaling feature* of our icicle views is used in Figure 14. Comparing the first icicle to the last icicle immediately indicates that the heap grew several-fold. Furthermore, we again can see that nearly the complete growth accounts to six different types.

To find the reason for the memory leak, the allocation sites of the types that grew the most are inspected. We can see that nearly all `LinkedList` objects were allocated in the constructor of `MultiThreadedHttpConnection-Manager$HostConnectionPool`, which is the type that grew the third most (green). The main allocation site of that type is the method `MultiThreadedHttpConnectionManager$-ConnectionPool::getHostPool()`. This provided us with enough information to investigate that method in the source code. There we found that the `MultiThreadedHttpConnection-Manager$HostConnectionPool` objects are added to a map, yet they are not removed from that map when the connection is

`https://doi.org/10.2312/stag.20201241`

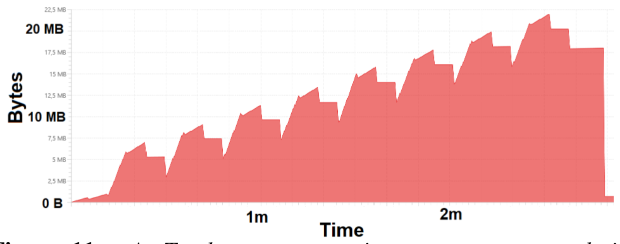


Figure 11: *AntTracks reports continuous memory growth in* `HttpClient`.

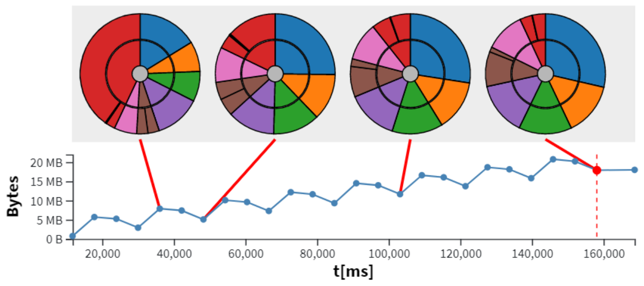


Figure 12: *Timeline view with unscaled sunbursts.*

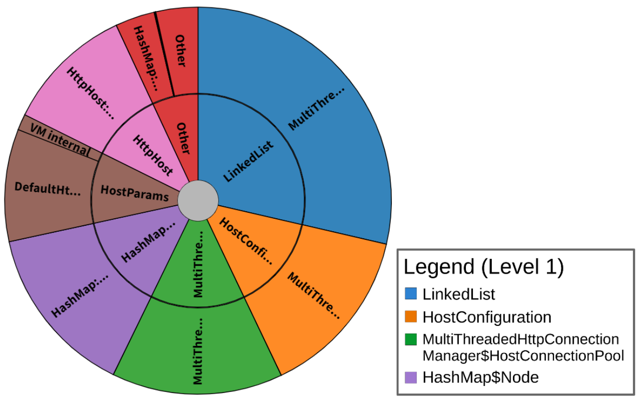


Figure 13: *Final sunburst at the end of the time window.*

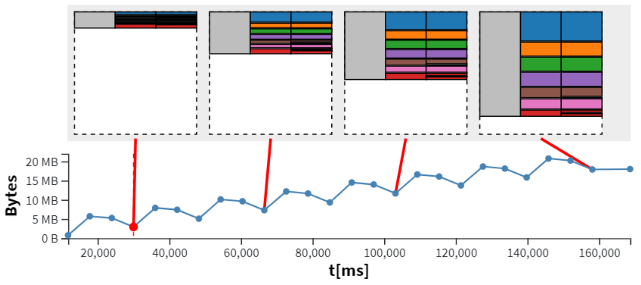


Figure 14: *Timeline view with scaled icicles.*

deleted, resulting in a memory leak. This causes the connection pool (and all other objects referenced by it) to accumulate over time. Fixing the code by correctly removing the objects from the map once the connections are closed gets rid of the memory leak and leads to the expected memory behavior, as shown in Figure 15.

7.2. easyTravel

The second investigated application is *Dynatrace easyTravel*. Dynatrace focuses on application performance monitoring (APM) and

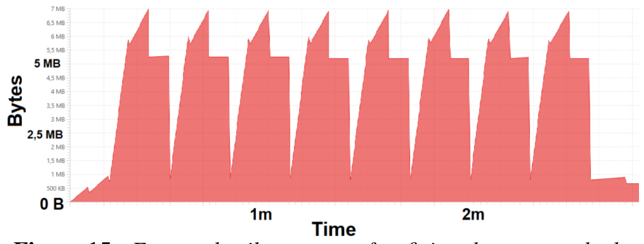


Figure 15: *Expected spike pattern after fixing the memory leak.*

distributes easyTravel as their state-of-the-art demo application. It is a multi-tier application for a travel agency, using a Java backend. An automatic load generator distributed together with easyTravel can simulate accesses to the service. When easyTravel is started, different problem patterns can be enabled and disabled, one of which is a hidden memory leak somewhere in the backend.

Our heap evolution visualization grouped the heap at multiple points in time using three classifiers: *containing data structure*, *type*, and *closest domain call site*. The first classifier groups objects based on the data structure(s) they are contained in. If an object is contained in a single data structure it is assigned the group "*<Data structure type> (allocated in <Data structure allocation site>)*", for example "*HashMap (allocated in MyClass::myMethod())*", otherwise it is either assigned the group "*Not contained in a data structure*" or "*Contained in multiple data structures*". This way, single data structures that keep alive a lot of objects can easily be detected. The third classifier, i.e., *closest domain call site*, differs from the normal allocation site classier as it returns the method call within easyTravel's code base that caused the allocation even if the allocation itself is hidden inside a third-party framework.

Figure 16 shows the local view of our time-travel-based sunburst visualization at three different points in time. The nodes within the trees have been sorted by *absolute growth*, i.e., independent of which sunburst we look at, we can automatically infer that the object group represented by the blue segment (i.e., objects stored in a `ConcurrentHashMap` data structure that has been allocated in method `findLocations()` of class `JourneyService`) grows the most over the selected time window. This also becomes apparent when comparing the sunburst at time t_1 to the sunburst at time t_3. While all other data structure segments (inner circle segments) shrink, the segment of the suspicious `ConcurrentHashMap` grows strongly. By hovering over the circle segment at both points in time, we can find out that the `ConcurrentHashMap` only makes up 11.8% of the heap at t_1, while it makes up 39.5% at t_3.

Figure 17 shows the sunburst at t_3, with a drill-down performed on the `ConcurrentHashMap`. We can see that around 45% of the objects stored in this data structure are each of type `Location` and `Date`. Due to the drill-down, we can now also see the third tree level, i.e., the *closest domain call sites*. Since `Location` as well as `Date` both only have one child node, we know that all allocations of these objects are caused by a single method each.

The collected information greatly helps to locate and fix the problem. The method in which the `Location` objects depicted in Figure 17 are added to the suspicious `ConcurrentHashMap` was easily found. Its name `locationCache` and its use in the code

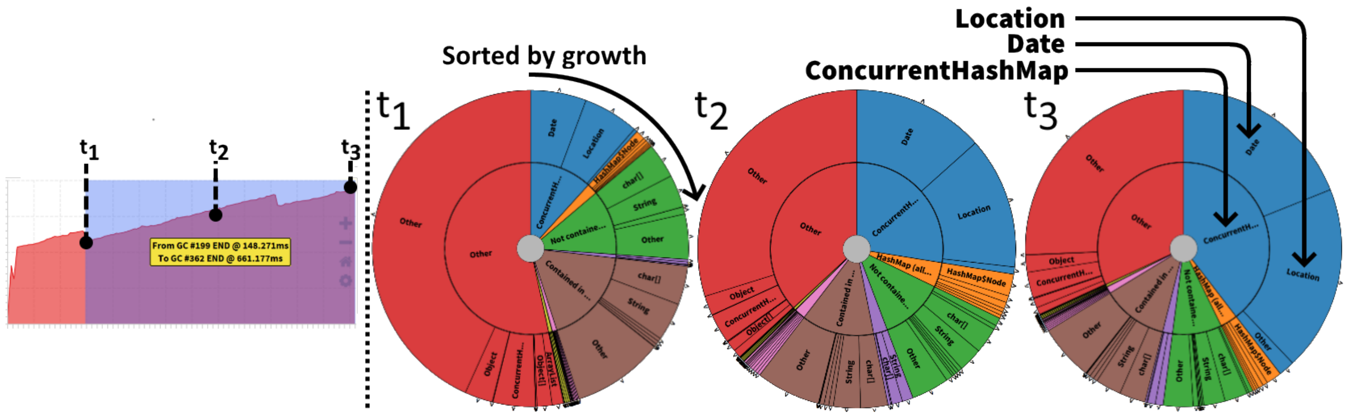


Figure 16: *Heap evolution time travel through easyTravel shown at three different points in time t_1, t_2, and t_3. This indicates a leak involving a data structure of type* `ConcurrentHashMap` *that has been allocated in method* `findLocations()` *of class* `JourneyService` *(inner blue circle segment). This data structure accumulates* `Location` *and* `Date` *objects over time (outer blue circle segments).*

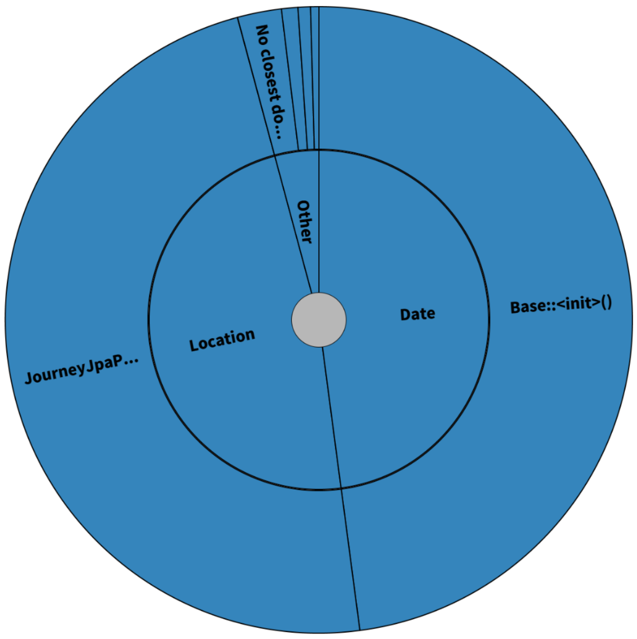


Figure 17: *Drill-down into the suspicious* `ConcurrentHashMap` *data structure at timestamp t_3.*

reveal that this map should serve as a cache for location searches. Once a search has been executed for a given `QueryKey`, a list of `Location` objects is stored. Subsequent searches for the same key should find the respective entry in the map. However, `QueryKey` neither implements `hashCode` nor `equals`. Thus, every request (even for an already existing key) resulted in a cache miss, which led to this typical memory leak. We were able to easily resolve this problem by implementing the two mentioned methods accordingly.

8. Related Work

As ample work regarding tree visualizations has already been presented throughout this work, this section focuses on visualizations in the domain of memory monitoring. Most memory visualizations revolve around object (reference) graph visualization. A pure object graph consists of nodes representing heap objects and edges that represent the references between them [PNC98]. Even though such a graph could be directly visualized as a node-link diagram [ZZ01], the size of modern applications (having millions of live objects) renders approaches that display every heap object as a separate node infeasible. Thus, most approaches create *ownership trees* using the concept of *object ownership* [PNC98, Mit06, WGM18b] based on the *dominator relation* [LT79]. Ownership trees can be used to detect objects that keep many other objects alive.

Reiss [Rei09, Rei10] visualizes the aggregated ownership graph in an icicle-like visualization using coloring, hatching, hue and saturation. The approach by Hill et al. [HNP00, HNP02] plots the evolution of ownership trees in a scalable tree visualization that shares visual similarities with flame graph [Gre16a, Gre16b]. Mitchell et al. [MSS09] apply further transformations on ownership trees to detect costly data structures, which are then displayed in a node-link diagram. Heapviz [AKG*10, KAG*13] is a tool that also displays data structures on different levels of detail, arranging collapsible nodes in a radial node-link diagram. The work by De Pauw and Sevitsky [DPS00] is one of the few object graph visualization approaches that does not utilize the dominator relation. Instead, they extract reference patterns, i.e., repetitive reference sequences in the heap object graph, and visualize occurrences of these patterns. The detection of these patterns can be restricted to those objects that have been created between two heap snapshots (i.e., potentially leaking objects), which then can be explored visually.

Our approach is orthogonal to these existing visualization approaches. Most object reference graph visualizations focus on the analysis of the keep-alive relation between objects (e.g., which objects keep alive objects of type *B*). Yet, this expects the user to already know which objects need to be inspected in more detail. This is where our approach comes into play. It gives the user visual information about which objects accumulate over time, as these are the objects that are most likely the result of a memory leak. For example, it may report that objects of type `B` that are allocated in method `MyClass:myMethod()` consistently increased in number over time. Thus, the main focus of this work lies in the *detection* of growing heap object groups. Nevertheless, the information provided by our visualizations may not only help in detecting growing object groups, but are also often already detailed enough to help in *fixing* a leak. As shown in Section 7, knowing which kinds of objects accumulate, where these accumulating objects are allocated and in which data structures they are stored (information that can be provided by our approach) is often enough to be able to locate and fix a problem in the source code. Thus, our approach can be used on its own, or in combination with existing graph-based analysis techniques if additional information is needed or wanted.

9. Current Limitations and Future Work

In this section, we discuss current limitations of our work and our tool and how we will address them in the future.

9.1. User Study

We believe that the presented techniques are useful to inspect memory evolution over time, especially for novice users that could otherwise easily be overwhelmed if the visualized data was presented in raw format or tables. We presented case studies to demonstrate how tree visualization can be used in the domain of memory monitoring and how users are able to understand and reason about the memory behavior of real world applications. Even though existing work suggests that tree visualizations are useful for a variety of analysis tasks [WTM06, Teo07, BPP17], a more thorough evaluation is planned. We want to conduct a user study to compare the performance of participants who use tree visualizations to inspect an application's memory behavior with the performance of participants who use other graphical and textual representations.

9.2. Information Highlighting and Guidance

Even though information visualization has the potential to ease the analysis of the underlying data, a person still requires a fair amount of background knowledge and experience to perform memory analysis effectively. Especially novice users often lack this experience and consequently struggle when using memory analysis tools [WGGS20]. In the future, we want to further increase the accessibility of our tree visualizations by making their use easier. The tool should automatically detect suspicious memory behavior, e.g., growing object groups, and then *guide* the user through the analysis by *explaining* the steps that have to be performed, alongside automatic *highlighting* of important information in the visualization.

9.3. Reference Visualization

As shown, our current visualizations are a great way to detect and inspect growth over time. Yet, once we know which objects accumulate over time (often objects of a few different types allocated at a few different allocation sites), most of the further analysis happens on the source code level in the IDE. Objects accumulate over time if they are directly or indirectly referenced by a GC root. For example, as presented in Section 7.1, objects of one type that were stored in a map caused objects of multiple different types to accumulate. Thus, the references between objects can be a vital information in the analysis of memory leaks. In the future, we plan to extend our tree visualization to support the depiction of references between object groups, for example by using tree visualization that support hierarchical edge bundling [Hol06, HCvW07, HdRFH12].

9.4. Data Structure Growth Visualization

Currently, our visualization tool can display the evolution of the whole heap over time by grouping the live heap objects based on a list of user-defined classifiers at multiple points in time. In the future, we want to explore how to use the same tree visualization techniques to display the results of existing analysis features that yet lack visualization support. Inspired by related work on data structure visualization [AKG*10, KAG*13], one of AntTracks's analysis features that we want to enrich using our visualizations is its *automatic data structure growth analysis* [WGM18a, WGM19a]. This feature automatically detects strongly growing data structures in a monitored application and reports them to the user for more detailed inspection using drill-down operations within a tree table. Since the visualization of evolutionary data as well as drilling down on the data are core features of our approach, we plan to integrate our tree visualizations with the existing data structure analysis.

10. Conclusions

In this paper, we presented our approach to apply *tree visualizations* to facilitate the analysis of *memory leaks*. We discussed how a heap state, more specifically its heap objects, can be grouped into a memory tree, and how such a tree can be visualized using existing tree visualization techniques. We defined a requirements catalog that we used to select the *sunburst* and the *icicle* visualization techniques as suitable to display heap memory. We then presented techniques how to reduce a memory tree's complexity by pruning it and how a drill-down functionality can be used in the selected visualizations to enable detailed analyses of the heap composition. Our approach is not only able to display a single heap state, but can also visualize the memory evolution over time in two different ways: a timeline-based approach that displays the visualized state of the heap at multiple point in time side-by-side, and a time-travel-based approach for detailed analyses. Growing elements in these visualizations hint at an accumulation of heap objects that could be the result of a possible memory leak.

We implemented our approach as a D3.js web application that supports various convenience features such as animations when moving between points in time. We presented case studies in which we performed memory analyses of different applications to show the approach's feasibility and usefulness. We hope that our tree visualizations can aid experienced users as well as users with a limited background in memory analysis in visually inspecting and analyzing the memory behavior of their applications.

Acknowledgment

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

[AHL*13] AUBER D., HUET C., LAMBERT A., RENOUST B., SALLABERRY A., SAULNIER A.: GosperMap: Using a Gosper Curve for Laying Out Hierarchical Data. *IEEE Trans. Vis. Comput. Graph. 19*, 11 (2013), 1820–1832. URL: https://doi.org/10.1109/TVCG.2013.91, doi:10.1109/TVCG.2013.91. 3

[AKG*10] AFTANDILIAN E., KELLEY S., GRAMAZIO C., RICCI N. P., SU S. L., GUYER S. Z.: Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proc. of the ACM Symposium on Software Visualization (SOFTVIS)* (2010), pp. 53–62. URL: https://doi.org/10.1145/1879211.1879222, doi:10.1145/1879211.1879222. 9, 10

[Apa06] APACHE SOFTWARE FOUNDATION: Commons HttpClient version 3.0.1, 2006. URL: https://mvnrepository.com/artifact/commons-httpclient/commons-httpclient/3.0.1. 7

[Apa20] APACHE SOFTWARE FOUNDATION: Issue tracker for HttpClient, 2020. URL: https://issues.apache.org/jira/projects/HTTPCLIENT/issues. 7

[BAB18] BLANCO A. F., ALCOCER J. P. S., BERGEL A.: Effective Visualization of Object Allocation Sites. In *Proc. of the IEEE Working Conference on Software Visualization (VISSOFT)* (2018), pp. 43–53. URL: https://doi.org/10.1109/VISSOFT.2018.00013, doi:10.1109/VISSOFT.2018.00013. 1

[BLM15] BITTO V., LENGAUER P., MÖSSENBÖCK H.: Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proc. of the Int'l. Conf. on Principles and Practices of Programming on The Java Platform (PPPJ)* (2015), pp. 76–89. URL: https://doi.org/10.1145/2807426.2807433, doi:10.1145/2807426.2807433. 2

[BN01] BARLOW S. T., NEVILLE P.: A Comparison of 2-D Visualizations of Hierarchies. In *Proc. of the IEEE Symposium on Information Visualization (INFOVIS)* (2001), pp. 131–138. URL: https://doi.org/10.1109/INFVIS.2001.963290, doi:10.1109/INFVIS.2001.963290. 3

[BNK16] BACHER I., NAMEE B. M., KELLEHER J. D.: Using Icicle Trees to Encode the Hierarchical Structure of Source Code. In *Proc. of the Eurographics Conf. on Visualization (EuroVis)* (2016), pp. 97–101. URL: https://doi.org/10.2312/eurovisshort.20161168, doi:10.2312/eurovisshort.20161168. 4

[BOH11] BOSTOCK M., OGIEVETSKY V., HEER J.: D^3 Data-Driven Documents. *IEEE Trans. Vis. Comput. Graph. 17*, 12 (2011), 2301–2309. URL: https://doi.org/10.1109/TVCG.2011.185, doi:10.1109/TVCG.2011.185. 7

[BPP17] BIUK-AGHAI R. P., PANG P. C., PANG B.: Map-like Visualisations vs. Treemaps: An Experimental Comparison. In *Proc. of the 10th Int'l. Symposium on Visual Information Communication and Interaction (VINCI)* (2017), ACM, pp. 113–120. URL: https://doi.org/10.1145/3105971.3105976, doi:10.1145/3105971.3105976. 10

[Bru09] BRUTLAG J.: Speed Matters for Google Web Search, 2009. URL: https://ai.googleblog.com/2009/06/speed-matters.html. 3

[CM07] CAWTHON N., MOERE A. V.: The Effect of Aesthetic on the Usability of Data Visualization. In *Proc. of the 11th Int'l. Conf. on Information Visualisation (IV)* (2007), pp. 637–648. URL: https://doi.org/10.1109/IV.2007.147, doi:10.1109/IV.2007.147. 3

[CPRG16] CALLEYA J., PAWLING R., RYAN C., GASPAR H. M.: Using Data Driven Documents (D3) to Explore a Whole Ship Model. In *Proc. of the 11th System of Systems Engineering Conf. (SoSE)* (2016), pp. 1–6. URL: https://doi.org/10.1109/SYSOSE.2016.7542947, doi:10.1109/SYSOSE.2016.7542947. 7

[CZvDR09] CORNELISSEN B., ZAIDMAN A., VAN DEURSEN A., ROMPAEY B. V.: Trace Visualization for Program Comprehension: A Controlled Experiment. In *Proc. of the 17th IEEE Int'l. Conf. on Program Comprehension (ICPC)* (2009), pp. 100–109. URL: https://doi.org/10.1109/ICPC.2009.5090033, doi:10.1109/ICPC.2009.5090033. 1

[DPS00] DE PAUW W., SEVITSKY G.: Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency - Practice and Experience 12*, 14 (2000), 1431–1454. URL: https://doi.org/10.1002/1096-9128(20001210)12:14<1431::AID-CPE542>3.0.CO;2-2, doi:10.1002/1096-9128(20001210)12:14<1431::AID-CPE542>3.0.CO;2-2. 9

[Ecl20] ECLIPSE FOUNDATION: Eclipse Memory Analyzer (MAT), 2020. URL: https://www.eclipse.org/mat/. 1

[FFHW15] FITTKAU F., FINKE S., HASSELBRING W., WALLER J.: Comparing Trace Visualizations for Program Comprehension Through Controlled Experiments. In *Proc. of the 23rd IEEE Int'l. Conf. on Program Comprehension (ICPC)* (2015), pp. 266–276. URL: https://doi.org/10.1109/ICPC.2015.37, doi:10.1109/ICPC.2015.37. 1

[FKH15] FITTKAU F., KRAUSE A., HASSELBRING W.: Hierarchical Software Landscape Visualization for System Comprehension: A Controlled Experiment. In *Proc. of the 3rd IEEE Working Conf. on Software Visualization (VISSOFT)* (2015), pp. 36–45. URL: https://doi.org/10.1109/VISSOFT.2015.7332413, doi:10.1109/VISSOFT.2015.7332413. 1

[FKH17] FITTKAU F., KRAUSE A., HASSELBRING W.: Software Landscape and Application Visualization for System Comprehension with ExplorViz. *Inf. Softw. Technol. 87* (2017), 259–277. URL: https://doi.org/10.1016/j.infsof.2016.07.004, doi:10.1016/j.infsof.2016.07.004. 1, 3

[FM11] FETTE I., MELNIKOV A.: The WebSocket Protocol. *RFC 6455* (2011), 1–71. URL: https://doi.org/10.17487/RFC6455, doi:10.17487/RFC6455. 7

[GCS*20] GHANAVATI M., COSTA D., SEBOEK J., LO D., ANDRZEJAK A.: Memory and Resource Leak Defects and their Repairs in Java Projects. *Empirical Software Engineering 25*, 1 (2020), 678–718. URL: https://doi.org/10.1007/s10664-019-09731-8, doi:10.1007/s10664-019-09731-8. 1

[Gre16a] GREGG B.: The Flame Graph. *ACM Queue 14*, 2 (2016), 10. URL: https://doi.org/10.1145/2927299.2927301, doi:10.1145/2927299.2927301. 9

[Gre16b] GREGG B.: The Flame Graph. *Commun. ACM 59*, 6 (2016), 48–57. URL: https://doi.org/10.1145/2909476, doi:10.1145/2909476. 9

[HBO10] HEER J., BOSTOCK M., OGIEVETSKY V.: A Tour through the Visualization Zoo. *ACM Queue 8*, 5 (2010), 20. URL: http://doi.acm.org/10.1145/1794514.1805128, doi:10.1145/1794514.1805128. 2, 4

[HCvW07] HOLTEN D., CORNELISSEN B., VAN WIJK J. J.: Trace Visualization Using Hierarchical Edge Bundles and Massive Sequence Views. In *Proc. of the 4th IEEE Int'l. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)* (2007), pp. 47–54. URL: https://doi.org/10.1109/VISSOF.2007.4290699, doi:10.1109/VISSOF.2007.4290699. 10

[HdRFH12] HOP W., DE RIDDER S., FRASINCAR F., HOGENBOOM F.: Using Hierarchical Edge Bundles to Visualize Complex Ontologies in GLOW. In *Proc. of the ACM Symposium on Applied Computing (SAC)* (2012), pp. 304–311. URL: https://doi.org/10.1145/2245276.2245338, doi:10.1145/2245276.2245338. 10

[HHK*17] HINIKER A., HONG S. R., KIM Y., CHEN N., WEST J. D., ARAGON C. R.: Toward the Operationalization of Visual Metaphor. *J. Assoc. Inf. Sci. Technol. 68*, 10 (2017), 2338–2349. URL: https://doi.org/10.1002/asi.23857, doi:10.1002/asi.23857. 3

[HNP00] HILL T., NOBLE J., POTTER J.: Scalable Visualisations with Ownership Trees. In *Proc. of the 37th Int'l. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS)* (2000), pp. 202–213. URL: https://doi.org/10.1109/TOOLS.2000.891370, doi:10.1109/TOOLS.2000.891370. 9

[HNP02] HILL T., NOBLE J., POTTER J.: Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Vis. Lang. Comput. 13*, 3 (2002), 319–339. URL: https://doi.org/10.1006/jvlc.2002.0238, doi:10.1006/jvlc.2002.0238. 9

[Hol06] HOLTEN D.: Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Trans. Vis. Comput. Graph. 12*, 5 (2006), 741–748. URL: https://doi.org/10.1109/TVCG.2006.147, doi:10.1109/TVCG.2006.147. 10

[HTMD14] HAHN S., TRÜMPER J., MORITZ D., DÖLLNER J.: Visualization of Varying Hierarchies by Stable Layout of Voronoi Treemaps. In *Proc. of the 5th Int'l. Conf. on Information Visualization Theory and Applications (IVAPP* (2014), pp. 50–58. URL: https://doi.org/10.5220/0004686200500058, doi:10.5220/0004686200500058. 3

[JS91] JOHNSON B., SHNEIDERMAN B.: Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures. In *Proc. of the IEEE Conf. on Visualization* (1991), pp. 284–291. URL: https://doi.org/10.1109/VISUAL.1991.175815, doi:10.1109/VISUAL.1991.175815. 2

[KAG*13] KELLEY S., AFTANDILIAN E., GRAMAZIO C., RICCI N. P., SU S. L., GUYER S. Z.: Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. *Information Visualization 12*, 2 (2013), 163–177. URL: https://doi.org/10.1177/1473871612438786, doi:10.1177/1473871612438786. 9, 10

[Kit94] KITCHIN R. M.: Cognitive maps: What are they and why study them? *Journal of Environmental Psychology 14*, 1 (1994), 1–19. URL: https://doi.org/10.1016/S0272-4944(05)80194-X, doi:10.1016/S0272-4944(05)80194-X. 3

[KK91] KAMADA T., KAWAI S.: A General Framework for Visualizing Abstract Objects and Relations. *ACM Trans. Graph. 10*, 1 (1991), 1–39. URL: https://doi.org/10.1145/99902.99903, doi:10.1145/99902.99903. 2

[Lak94] LAKOFF G.: *Master Metaphor List*. University of California, 1994. 3

[LBF*16] LENGAUER P., BITTO V., FITZEK S., WENINGER M., MÖSSENBÖCK H.: 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)* (2016), pp. 4:1–4:11. URL: https://doi.org/10.1145/2972206.2972220, doi:10.1145/2972206.2972220. 2

[LBM15] LENGAUER P., BITTO V., MÖSSENBÖCK H.: Accurate and Efficient Object Tracing for Java Applications. In *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)* (2015), pp. 51–62. URL: https://doi.org/10.1145/2668930.2688037, doi:10.1145/2668930.2688037. 2

[LBM16] LENGAUER P., BITTO V., MÖSSENBÖCK H.: Efficient and Viable Handling of Large Object Traces. In *Proc. of the 7th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)* (2016), pp. 249–260. URL: https://doi.org/10.1145/2851553.2851555, doi:10.1145/2851553.2851555. 2

[LH14] LIU Z., HEER J.: The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Trans. Vis. Comput. Graph. 20*, 12 (2014), 2122–2131. URL: https://doi.org/10.1109/TVCG.2014.2346452, doi:10.1109/TVCG.2014.2346452. 3

[LSDT19] LIMBERGER D., SCHEIBEL W., DÖLLNER J., TRAPP M.: Advanced Visual Metaphors and Techniques for Software Maps. In *Proc. of the 12th Int'l. Symposium on Visual Information Communication and Interaction (VINCI)* (2019), pp. 11:1–11:8. URL: https://doi.org/10.1145/3356422.3356444, doi:10.1145/3356422.3356444. 3

[LSP08] LANGELIER G., SAHRAOUI H. A., POULIN P.: Exploring the Evolution of Software Quality with Animated Visualization. In *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2008), pp. 13–20. URL: https://doi.org/10.1109/VLHCC.2008.4639052, doi:10.1109/VLHCC.2008.4639052. 5

[LT79] LENGAUER T., TARJAN R. E.: A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1 (1979), 121–141. URL: https://doi.org/10.1145/357062.357071, doi:10.1145/357062.357071. 9

[Mit06] MITCHELL N.: The Runtime Structure of Object Ownership. In *Proc. of the 20th European Conf. on Object-oriented Programming (ECOOP)* (2006), pp. 74–98. URL: https://doi.org/10.1007/11785477_5, doi:10.1007/11785477_5. 9

[MSS09] MITCHELL N., SCHONBERG E., SEVITSKY G.: Making Sense of Large Heaps. In *Proc. of the 23rd European Conf. on Object-Oriented Programming (ECOOP)* (2009), pp. 77–97. URL: https://doi.org/10.1007/978-3-642-03013-0_5, doi:10.1007/978-3-642-03013-0_5. 9

[Mur13] MURRAY S.: *Interactive Data Visualization for the Web*. O'Reilly Media, Inc., 2013. 2

[Ora20] ORACLE: VisualVM: All-in-One Java Troubleshooting Tool, 2020. URL: https://visualvm.github.io/. 1

[PNC98] POTTER J., NOBLE J., CLARKE D. G.: The Ins and Outs of Objects. In *Proc. of the Australian Software Engineering Conf. (ASWEC)* (1998), pp. 80–89. URL: https://doi.org/10.1109/ASWEC.1998.730915, doi:10.1109/ASWEC.1998.730915. 9

[Rei09] REISS S. P.: Visualizing the Java Heap to Detect Memory Problems. In *Proc. of the 5th IEEE Int'l. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)* (2009), pp. 73–80. URL: https://doi.org/10.1109/VISSOF.2009.5336418, doi:10.1109/VISSOF.2009.5336418. 9

[Rei10] REISS S. P.: Visualizing the Java Heap. In *Proc. of the 32nd ACM/IEEE Int'l. Conf. on Software Engineering* (2010), pp. 251–254. URL: https://doi.org/10.1145/1810295.1810344, doi:10.1145/1810295.1810344. 9

[Sch11] SCHULZ H.: Treevis.net: A Tree Visualization Reference. *IEEE Computer Graphics and Applications 31*, 6 (2011), 11–15. URL: https://doi.org/10.1109/MCG.2011.103, doi:10.1109/MCG.2011.103. 2

[SS06] SCHULZ H., SCHUMANN H.: Visualizing Graphs - A Generalized View. In *Proc. of the 10th Int'l. Conf. on Information Visualisation (IV)* (2006), IEEE Computer Society, pp. 166–173. URL: https://doi.org/10.1109/IV.2006.130, doi:10.1109/IV.2006.130. 2

[STLD20] SCHEIBEL W., TRAPP M., LIMBERGER D., DÖLLNER J.: A Taxonomy of Treemap Visualization Techniques. In *Proc. of the 15th Int'l. Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)* (2020), pp. 273–280. URL: https://doi.org/10.5220/0009153902730280, doi:10.5220/0009153902730280. 2

[SZ00] STASKO J. T., ZHANG E.: Focus+Context Display and Navigation Techniques for Enhancing Radial, Space-Filling Hierarchy Visualizations. In *Proc. of the IEEE Symposium on Information Visualization (INFOVIS)* (2000), pp. 57–65. URL: https://doi.org/10.1109/INFVIS.2000.885091, doi:10.1109/INFVIS.2000.885091. 2, 3

[Teo07] TEOH S. T.: A Study on Multiple Views for Tree Visualization. In *Proc. of SPIE - Visualization and Data Analysis* (2007), vol. 6495. URL: https://doi.org/10.1117/12.703076, doi:10.1117/12.703076. 10

[TM04] TORY M., MÖLLER T.: Human Factors in Visualization Research. *IEEE Trans. Vis. Comput. Graph. 10*, 1 (2004), 72–84. URL: https://doi.org/10.1109/TVCG.2004.1260759, doi:10.1109/TVCG.2004.1260759. 4

[War13] Ware C.: Chapter One - Foundations for an Applied Science of Data Visualization. In *Information Visualization (Third Edition)*, third edition ed., Interactive Technologies. Morgan Kaufmann, Boston, 2013, pp. 1 – 30. URL: http://www.sciencedirect.com/science/article/pii/B9780123814647000016, doi:https://doi.org/10.1016/B978-0-12-381464-7.00001-6. 2

[Wen20] Weninger M.: AntTracks, 2020. URL: http://mevss.jku.at/AntTracks. 2

[WGGS20] Weninger M., Grünbacher P., Gander E., Schörgenhumer A.: Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study. *Proc. ACM Hum.-Comput. Interact. 4*, EICS (June 2020). URL: https://doi.org/10.1145/3394977, doi:10.1145/3394977. 10

[WGK10] Ward M. O., Grinstein G. G., Keim D. A.: *Interactive Data Visualization - Foundations, Techniques, and Applications*. A K Peters, 2010. URL: http://www.akpeters.com/product.asp?ProdCode=4735. 2

[WGM18a] Weninger M., Gander E., Mössenböck H.: Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring. In *Proc. of the 9th Symp. on Software Performance (SSP)* (2018), pp. 64–66. URL: http://pi.informatik.uni-siegen.de/stt/39_3/01_Fachgruppenberichte/SSP18/WeningerGanderMoessenboeck18.pdf. 1, 10

[WGM18b] Weninger M., Gander E., Mössenböck H.: 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)* (2018), pp. 14:1–14:13. URL: https://doi.org/10.1145/3237009.3237023, doi:10.1145/3237009.3237023. 9

[WGM19a] Weninger M., Gander E., Mössenböck H.: Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proc. of the 2019 ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)* (2019), pp. 273–284. URL: https://doi.org/10.1145/3297663.3310297, doi:10.1145/3297663.3310297. 1, 2, 10

[WGM19b] Weninger M., Gander E., Mössenböck H.: Detection of Suspicious Time Windows In Memory Monitoring. In *Proc. of the 16th ACM SIGPLAN Int'l. Conf. on Managed Programming Languages and Runtimes (MPLR)* (2019), pp. 95–104. URL: https://doi.org/10.1145/3357390.3361025, doi:10.1145/3357390.3361025. 5

[Wil09] Williams R.: *The Animator's Survival Kit–Revised Edition: A Manual of Methods, Principles and Formulas for Classical, Computer, Games, Stop Motion and Internet Animators*. Faber & Faber, Inc., 2009. 6

[WL07] Wettel R., Lanza M.: Visualizing Software Systems as Cities. In *Proc. of the 4th IEEE Int'l. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)* (2007), IEEE Computer Society, pp. 92–99. URL: https://doi.org/10.1109/VISSOF.2007.4290706, doi:10.1109/VISSOF.2007.4290706. 3

[WL08] Wettel R., Lanza M.: Visual Exploration of Large-Scale System Evolution. In *Proc. of the 15th Working Conf. on Reverse Engineering (WCRE)* (2008), pp. 219–228. URL: https://doi.org/10.1109/WCRE.2008.55, doi:10.1109/WCRE.2008.55. 5, 6

[WLM17] Weninger M., Lengauer P., Mössenböck H.: User-centered Offline Analysis of Memory Monitoring Data. In *Proc. of the 8th ACM/SPEC on Int'l. Conf. on Performance Engineering (ICPE)* (2017), pp. 357–360. URL: https://doi.org/10.1145/3030207.3030236, doi:10.1145/3030207.3030236. 1, 2

[WLR11] Wettel R., Lanza M., Robbes R.: Software Systems as Cities: A Controlled Experiment. In *Proc. of the 33rd Int'l. Conf. on Software Engineering (ICSE)* (2011), ACM, pp. 551–560. URL: https://doi.org/10.1145/1985793.1985868, doi:10.1145/1985793.1985868. 1, 3

[WM18] Weninger M., Mössenböck H.: User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proc. of the 2018 ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)* (2018), pp. 115–126. URL: https://doi.org/10.1145/3184407.3184412, doi:10.1145/3184407.3184412. 1, 2

[WM20] Weninger M., Makor L.: HttpClient Leak Driver, 2020. URL: https://github.com/NeonMika/httpclient-leak-driver/. 7

[WMGM19] Weninger M., Makor L., Gander E., Mössenböck H.: AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *Comp. of the 2019 ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)* (2019), pp. 29–32. URL: https://doi.org/10.1145/3302541.3313100, doi:10.1145/3302541.3313100. 1, 5

[WMM19] Weninger M., Makor L., Mössenböck H.: Memory Leak Visualization using Evolving Software Cities. In *Proc. of the 10th Symp. on Software Performance (SSP)* (2019), pp. 44–46. URL: http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Weninger.pdf. 3, 6

[WMM20] Weninger M., Makor L., Mössenböck H.: Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *Proc. of the 8th IEEE Working Conference on Software Visualization (VISSOFT)* (2020). 3, 6

[WPJR11] Wessels A., Purvis M., Jackson J., Rahman S. S.: Remote Data Visualization through WebSockets. In *Proc. of the 8th Int'l. Conf. on Information Technology: New Generations (ITNG)* (2011), pp. 1050–1051. URL: https://doi.org/10.1109/ITNG.2011.182, doi:10.1109/ITNG.2011.182. 7

[WTM06] Wang Y., Teoh S. T., Ma K.: Evaluating the Effectiveness of Tree Visualization Systems for Knowledge Discovery. In *Proc. of the Joint Eurographics - IEEE VGTC Symposium on Visualization (EuroVis)* (2006), pp. 67–74. URL: https://doi.org/10.2312/VisSym/EuroVis06/067-074, doi:10.2312/VisSym/EuroVis06/067-074. 10

[WWF*13] Waller J., Wulf C., Fittkau F., Dohring P., Hasselbring W.: Synchrovis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency. In *Proc. of the 1st IEEE Working Conf. on Software Visualization (VISSOFT)* (2013), pp. 1–4. URL: https://doi.org/10.1109/VISSOFT.2013.6650520, doi:10.1109/VISSOFT.2013.6650520. 3

[ZL15] Zhao H., Lu L.: Variational Circular Treemaps for Interactive Visualization of Hierarchical Data. In *Proc. of the IEEE Pacific Visualization Symposium (PacificVis)* (2015), pp. 81–85. URL: https://doi.org/10.1109/PACIFICVIS.2015.7156360, doi:10.1109/PACIFICVIS.2015.7156360. 3

[ZZ01] Zimmermann T., Zeller A.: Visualizing memory graphs. In *Software Visualization* (2001), pp. 191–204. URL: https://doi.org/10.1007/3-540-45875-1_15, doi:10.1007/3-540-45875-1_15. 9