Analyzing the Evolution of Data Structures in Trace-Based Memory Monitoring

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

Abstract

Modern software systems are becoming increasingly complex and are thus more prone to performance degradation due to memory leaks. Memory leaks occur if objects that are not needed anymore are still unintentionally kept alive. While there exists a variety of state-of-the-art memory monitoring tools, most of them only use memory snapshots, i.e., heap dumps, to analyze an application's live objects at a single point in time. This does not allow developers to identify data structures that grow over time. Trace-based monitoring tools tackle this problem by recording memory events, e.g., allocations or object moves performed by the garbage collector (GC), throughout an application's run time. In this paper, we present ongoing research on the use of memory traces for detecting the root causes of memory leaks introduced by growing data structures. This encompasses (1) a domain-specific language (DSL) to describe arbitrary data structures, (2) an algorithm to detect instances of previously defined data structures in reconstructed heaps, as well as (3) techniques to analyze the temporal evolution of these data structure instances to identify those possibly involved in memory leaks. All these concepts have been integrated into AntTracks, a trace-based memory monitoring tool, to prove their feasibility.

1 Introduction

Modern programming languages such as Java use automatic garbage collection. During garbage collection, objects that are not directly or indirectly reachable from static fields or thread-local variables (so-called *GC roots*) may be collected by the GC. We speak of a memory leak if no longer needed objects remain reachable from GC roots due to a programming error. For example, if a developer misses to remove no longer needed objects from their containing data structures, e.g., lists, these objects may not be collected by the GC. Beside excessive dynamic allocations [1], memory leaks are one of the major memory anomalies [4].

Since modern applications may involve hundreds of millions of objects at a single point in time, tool support to resolve memory problems is of paramount importance. Most state-of-the-art tools, such as VisualVM [8] or Eclipse Memory Analyzer (MAT) [7], perform heap analysis based on snapshots, i.e., heap dumps. While a single heap dump may allow developers to detect large data structures, it provides no information about the heap's evolution over time. Thus, some approaches [2, 8] take multiple snapshots and compare them. Nevertheless, such approaches do not allow temporal analyses on the *object-level*.

In contrast to snapshot-based approaches, trace-based approaches record additional information, e.g., object moves executed by the GC. This allows them to reconstruct the heap offline from the recorded trace for any point in time, as well as to track specific objects and their evolution throughout an application. Since it would not be feasible – due to memory restrictions and computational borders – to reconstruct and remember every single change to every single object, temporal analysis approaches need to focus on a certain subset of objects.

In this work, we present ongoing research on the use of memory traces. Our goal is to extract information about the root causes of memory leaks by focusing on the temporal development of *data structures*. This encompasses (1) a DSL that enables users to describe arbitrary data structures, (2) an algorithm to detect instances of previously defined data structures in reconstructed heaps, as well as (3) techniques to analyze the temporal evolution of these data structure instances to identify those possibly involved in memory leaks. To prove the feasibility of our approach, all concepts have been integrated into AntTracks. AntTracks is a trace-based memory monitoring tool based on the Hotspot Java VM, initially developed by Lengauer et al. [3] and extended by Weninger et al. [5, 6].

2 Approach

This section illustrates how data structures can be described in our DSL, how they are detected in reconstructed heaps, and how information about their temporal evolution is derived from AntTracks's memory traces.

2.1 Data Structure Definition

In object-oriented languages such as Java, data structures typically consist of a *head* object and multiple

other objects that reference each other according to a specific pattern. These patterns have to be known by a memory monitoring tool in order to enable it to perform data structure analyses. Therefore, we developed a DSL that allows us to describe arbitrary data structures. This allows us to ship descriptions of well-known data structures (e.g., data structures in Java's `java.util` package) directly with AntTracks. At the same time, tool users can extend this set of predefined data structure descriptions with descriptions of their own data structures.

Listing 1 shows an example of the DSL, describing the structure of `java.util.LinkedList`. Every type that is involved in the data structure needs a description, i.e., in our example `java.util.LinkedList` and `java.util.LinkedList$Node`. The former represents the *head* of the data structure (marked with the `DS` keyword), while the latter is an internal part of a data structure. Similar to Java syntax, the name of the type is followed by a set of curly braces. These contain a set of types (separated by semicolons) that may be referenced by the respective data structure part. For example, an instance of `java.util.LinkedList` may point to instances of `java.util.LinkedList$Node` (see line 2), which in turn may point to instances of `java.util.LinkedList$Node` instances (see line 5), and so on. Line 6 presents two special language features: (1) a star (i.e., `*`) can be used as a wildcard within the name of a pointed type and (2) enclosing a type in parentheses declares it as a *leaf*. The term `(*)` denotes a leaf of any type. Leaf information is used during data structure detection to determine the boundaries of a data structure. The DSL also supports namespaces which allow us to omit package declarations in type names.

2.2 Data Structure Detection

In order to detect data structures in a heap, the data structure definitions have to be parsed first. The parsed definitions are assigned to their corresponding types. Types for which no data structure definition was parsed are assigned a non-head dummy data structure definition that does not declare any pointed types. Array types are an exception to this rule and are handled in a special way. At this point, every type has a data structure definition assigned.

A reconstructed heap contains information about the objects live at a certain point in time (e.g., their types), as well as their references between each other. As a first step, the data structure detection algorithm filters and remembers all objects that are data structure heads, i.e., objects whose types have a *head* data structure definition assigned. Then, to determine which objects belong to a certain data structure instance, it recursively follows the head object's pointers. The recursive descent is stopped when a pointed object is encountered whose type is either (1) not part of the current object's data structure definition or (2) marked as a leaf in the current object's data structure definition. In the latter case, the object itself belongs to the data structure instance, but none of its referenced objects. Every visited object is marked to avoid multiple visits.

```
1 DS java.util.LinkedList {
2     java.util.LinkedList$Node;
3 }
4 java.util.LinkedList$Node {
5     java.util.LinkedList$Node;
6     (*);
7 }
```

Listing 1: Definition of `java.util.LinkedList` using our data structure DSL.

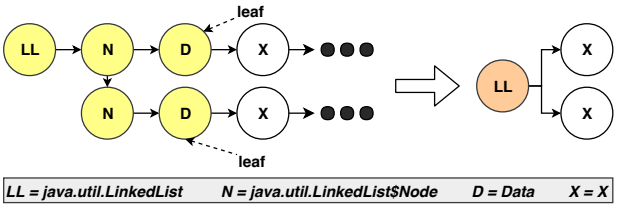

Figure 1: A `LinkedList` instance, consisting of the head (`LL`), two nodes (`N`) and two data objects (`D`).

For example, Figure 1 shows a `java.util.LinkedList` that has been detected using the description in Listing 1. Starting at the head `LL`, the first `N` instance is visited. The data structure description of `java.util.LinkedList$Node` then allows us to follow further nodes (line 5), or to visit any other object as a leaf without continuing the recursive descent (line 6). Thus, the first `D` instance and the second `N` instance are visited, continuing the descent from the `N` object. As a last step, the second `D` object is visited as a leaf.

2.3 Temporal Analysis

Trace-based approaches are better suited for temporal analysis than snapshot-based ones because they allow to derive temporal information on the *object-level*. Figure 2 illustrates this. Using only snapshots, without additional temporal information, it is not possible to decide whether two objects of type `X` are really the same or just share the same type.

AntTracks is able to derive this information by replaying the recorded GC move events. Thus, we know which objects survived between two points in time as well as their updated pointers. Using this knowledge, we can specifically search for data structure instances which (1) survived over a certain time window (since

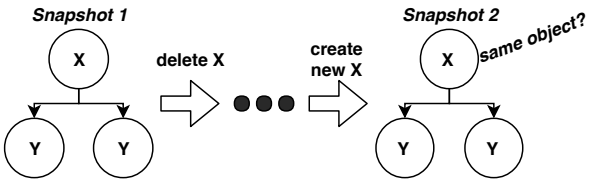

Figure 2: Analysis based on multiple snapshots lacks information *on the object-level*.

objects that died cannot be the root cause of a memory leak) and (2) reference / keep alive more objects than before.

Our workflow for temporal data structure evolution analysis consists of the following steps:

1. The user chooses two garbage collection points in time between which the temporal data structure evolution analysis should take place.
2. The heap is reconstructed for the first point in time and is stored. The addresses of all data structure heads in this heap, i.e., the *start addresses*, are stored as well.
3. At every garbage collection, we stop tracking data structure heads that died. For surviving heads, their new addresses (which can be reconstructed from GC move events) are stored alongside their start addresses.

Following this algorithm up to the end of the selected time window, we obtain (1) the reconstructed heap at the start, (2) the reconstructed heap at the end, and (3) a list of all data structures that survived, more specifically, their initial and final addresses.

For every data structure head, its *deep size* (i.e., how many objects can be reached from that object) as well as its *retained size* (i.e., how many objects are kept alive by that object) can be calculated for both points in time [5]. Based on that, the absolute and the relative change of these sizes can be calculated.

The metric that proved the most useful to identify problematic data structure instances in our preliminary evaluation is shown in Equation 1.

$$HGP(obj) = \frac{\Delta retained(obj)}{\Delta heapsize} \times 100 \qquad (1)$$

Given that the overall heap size increased, this formula calculates the ownership growth of each data structure relative to the heap growth, i.e., the *heap growth portion*. For example, assume that the overall heap size went from 1GB to 2GB and a list's retained size increased by 700MB. This would result in a HGP value of 70%, i.e., the ownership growth of this data structure contributes 70% to the total growth of the heap.

Sorting all data structures by this metric allows us to easily identify those that keep more objects alive than before. At the same time their growth is put into perspective to the absolute heap growth. In the case of a memory leak, objects that reveal a high HGP value are most likely involved in it.

3 Conclusion and Future Work

In this paper, we presented a new and easy-to-use DSL to describe arbitrary data structures and sketched an algorithm that detects instances of those data structures in reconstructed heaps. We discussed how temporal information regarding the growth of data structure instances can be derived from memory traces, including a metric that puts data structure growth in relation to the overall heap growth. This metric allows us to prioritize data structure instances according to how likely they are involved in a memory leak.

Being able to describe, detect and analyze the evolution of arbitrary data structures over time, even user-defined ones, yields many possibilities for future work. Due to the complexity of heap object graphs, it is not feasible to visualize and inspect them without abstraction, e.g., by aggregating nodes. Our work can be used to develop improved object graph visualization techniques that perform node aggregation based on data structure information. Data structure information may also be used to push automatic memory leak detection and resolution without human intervention.

4 Acknowledgement

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

References

[1] C. U. Smith and L. G. Williams. "Software performance antipatterns." In: *Workshop on Software and Performance*. 2000.

[2] M. Jump and K. S. McKinley. "Detecting memory leaks in managed languages with Cork". In: *Software: Practice and Experience* 40.1 (2010).

[3] P. Lengauer, V. Bitto, and H. Mössenböck. "Accurate and Efficient Object Tracing for Java Applications". In: *Proc. of the 6th ACM/SPEC Int'l. Conference on Performance Engineering*. 2015.

[4] M. Ghanavati et al. "Memory and Resource Leak Defects in Java Projects: An Empirical Study". In: *Proc. of the 40th Int'l Conf. on Software Engineering: Companion Proceeedings*. 2018.

[5] M. Weninger, E. Gander, and H. Mössenböck. "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*. 2018.

[6] M. Weninger and H. Mössenböck. "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*. 2018.

[7] Eclipse Foundation. *Eclipse Memory Analyzer (MAT) (last accessed August 13, 2018)*. `https://www.eclipse.org/mat/`.

[8] Oracle. *VisualVM: All-in-One Java Troubleshooting Tool (last accessed August 13, 2018)*. `https://visualvm.github.io/`.