User-centered Offline Analysis of Memory Monitoring Data

[Work in Progress]

Markus Weninger[1] Philipp Lengauer[2] Hanspeter Mössenböck[2]

[1]Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz, Austria
markus.weninger@jku.at

[2]Institute for System Software
Johannes Kepler University Linz, Austria
philipp.lengauer@jku.at

ABSTRACT

State-of-the-art memory monitoring tools collect lots of raw data. Yet, the analysis of this data is often not well supported. Existing tools restrict the user in the way how to analyze the underlying data, how to process it, and how to visualize it. This results in the dilemma that the raw data often contains more information than what can be exploited. We present a novel user-centric approach, allowing custom offline analysis and visualization of memory monitoring data by employing user-defined classification on heap objects. Putting the user in the center of the analysis process and providing flexible query and classification interfaces can change the way how we monitor memory usage. Our goal is to turn special-purpose memory monitoring tools into more general and customizable tools.

Keywords

Java, Memory Monitoring, Analysis Tool, User-centered, Classification, Grouping

1. INTRODUCTION

Modern programming languages, such as Java or C#, relieve the programmer from the error-prone task of freeing memory manually by relying on automatic garbage collection. Nevertheless, poor data structure implementation and usage may result in performance problems and memory leaks. Performance problems often root in known software performance anti-patterns (Smith and Williams [9]), e.g., *excessive dynamic memory allocations* (Peiris and Hill [7]). Memory leaks occur if an object references objects which are not needed anymore. This results in allocated memory that would be released otherwise. Since it is hard to manually detect and prevent such defects, advanced analysis tools are required.

A multitude of memory monitoring and analysis tools exist, all serving a specific purpose. They range from heap health metrics derived from the heap structure, as done in Mitchell and Sevitsky [6], object death time estimation, as done in Ricci et al. [8], to graph mining tools to detect reoccurring subgraphs, as done in Maxwell [5]. While serving their specific purpose well, they ignore the fact that much more information could be derived from the underlying data, and that this information is lost.

The data structures employed by these approaches are designed to provide fast access and processing only for the given purpose, but miss features to derive further information which lies outside their focus. Minimalistic user interfaces, which are often not even graphical, further increase the gap between available data and retrievable information. To reduce this information loss, tools have to support two fundamental concepts: (1) a general-purpose data structure with a flexible yet efficient data access layer and (2) a user interface to this data access layer, both graphically and via an programming interface.

In this paper, we present a novel approach on how to achieve flexible, user-centered information retrieval in memory monitoring and analysis tools using *user-defined object classifiers*. An object classifier is used to classify heap objects, i.e., objects that existed in the monitored application's heap, based on certain properties. New custom classifiers can easily be defined by the user as small dynamically-loaded code segments.

Instead of providing a non-extendable, predefined set of classifiers (e.g., the object's *type* or its *allocation site*), as done in conventional tools, *user-defined object classifiers* allow versatile object classification based on the user's needs, which puts the user in the center of the data analysis process. This enables retrieving information without the mentioned restrictions in a user-centered way.

Our heap processing framework encompasses various features based on object classifiers, such as filtering or grouping the objects on the heap. It is also possible to group heap objects based on multiple classifiers, e.g., to first group them by *type* and then by *allocation site*. This results in a classification tree that allows for step-wise, fine-grained data analysis. We call this *multi-level grouping*, a novel approach that is not available in other tools.

User-defined object classifiers do not have to be defined at load-time. They can be added at run-time utilizing Java Service Provider Interfaces (SPI) and in-memory on-the-fly compilation.

As a proof of concept, we implemented *user-defined object classifiers* in AntTracks (Lengauer et al. [4]), a memory monitoring tool based on a modified JVM (Java Virtual Machine) that was derived from the Java Hotspot™ VM.

ICPE'17, April 22-26, 2017, L'Aquila, Italy

DOI: http://dx.doi.org/10.1145/3030207.3030236

2. BACKGROUND

This section encompasses a short overview on AntTracks's workflow and heap reconstruction architecture.

2.1 Trace Recording And Reconstruction

As described in Lengauer et al. [4, 3], AntTracks' JVM records memory events throughout an application's execution and write them into trace files. These events include object allocations, object movements, and pointer updates during garbage collection. The heap's state can be reconstructed for the beginning and the end of every garbage collection cycle by incrementally processing the events in the trace.

2.2 Data Structure

The work in Bitto et al. [1] shows that a naïve approach, representing each heap object by a Java object in the analysis tool, results in unacceptable memory overhead. Therefore, the data structure shown in Figure 1 has been developed. It separates the heap into multiple spaces (e.g., the ParallelOldGC's heap consists of one eden space, two survivor spaces, and one old space). Each of these spaces encompasses various fields such as the starting address, the size, or the kind of the space (i.e., *eden*, *survivor* or *old*). Additionally, each space contains an address-to-LAB map. A LAB (local allocation buffer) represents a set of objects that have been processed together during garbage collection (e.g., objects that have been allocated by the same thread within the same thread-local allocation buffer (TLAB)). The LAB's object array contains pointers to the global cache of object representations, called `ObjectInfo`. `ObjectInfo`s are cached structures that contain information about objects, namely the event which created the object (e.g., an allocation by the interpreter), the object's allocation site, its type and its size. For arrays, also the array-length is stored. Furthermore, for each entry in the LAB's object array, a corresponding pointer entry exists.

3. APPROACH

To allow user-centric analysis within AntTracks, the underlying data structures and the user interface have been refactored and extended. The heap data structure supports heap traversal, filtering, mapping, and (multi-)grouping operations. Most of these operations rely on *object classifiers*, which are guidelines on how to classify heap objects based on their properties. In addition to a set of ready-to-use pre-defined classifiers, new classifiers and filters (a special kind of classifiers) can be defined by the user, either before or at run-time. Based on all the available classifiers and filters, the user can select a set of classifiers and filters which should be used to process, filter and group the heap. The resulting classification tree is then shown to the user in an expandable tree view control and an experimental visualization view, which allow step-wise and user-controlled analysis and data exploration.

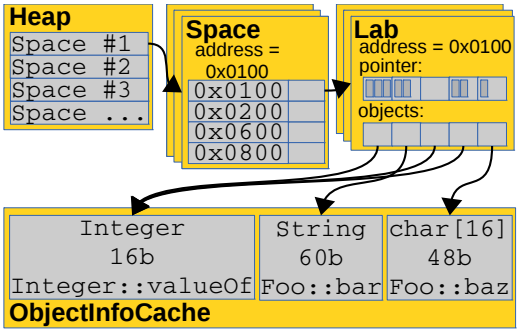

Figure 1: Reconstructed heap structure, splitted into spaces and LABs, in conjunction with a cache for `ObjectInfo` objects

The following sections will give details on the *heap analysis framework*, the different kinds of *object classifiers* and the resulting *classification tree*.

3.1 Heap Analysis Framework

As explained in the previous section, heap objects are not stored as distinct Java objects, but each LAB contains a map of addresses that refer to `ObjectInfo`s. Furthermore, the containing space and the object's pointers are available, therefore the information known about every heap object can be expressed as the set

$$Object = \{Space, Address, PointsTo, ObjectInfo\}$$

Constructing temporary objects for this set of information, which would be needed to support Java 8's streaming functionality, would introduce too much overhead (cf. Bitto et al. [1]), in terms of memory (i.e., object headers) as well as increased GC frequency. Therefore, a performant, user-extendable framework has been developed for heap analysis. It encompasses the following functions: (1) **Filter** allows restricting further functions to only process heap objects which match the defined `ObjectFilter`. (2) **ForEach** allows method execution on every filtered heap object. (3) **Mapping** allows mapping a filtered heap object's properties to an object of type `T`, which allows further processing as a Java `Stream<T>`. (4) **Grouping** applies (multiple) object classifiers to group all filtered objects based on their classification keys into a classification tree.

3.2 Object Classifiers

Object classifiers are objects that provide a `classify` function, which takes the heap object's properties as parameter and returns one ore more classification keys. If a classifier cannot classify an object, it is expected to return null.

To support user-defined classifiers, we provide a `ObjectClassifier` *service provider interface (SPI)* which can be seen in Figure 2. Classifiers implementing this interface can either be included as precompiled jar files or can be compiled on-the-fly from source code within AntTracks.

```
interface ObjectClassifier<T> {
  T classify(Space space, long address,
  long[] pointsTo, ObjectInfo obj);
}
```

Figure 2: ObjectClassifier interface

To provide a name, a description and a type for the classifier, classes which override this interface are also expected to be annotated with the following Java annotation:
`@Classifier(name = "<name>", desc = "<description>", type = ClassifierType.<type>`

Classifiers can differ in their multiplicity of classification: (1) One-to-one classifier (`ClassifierType.ONE`), (2) One-to-many classifier (`ClassifierType.MANY`) and (3) One-to-hierarchy classifier (`ClassifierType.HIERARCHY`).

3.2.1 One-to-one Classifier

A *one-to-one classifier* classifies a heap object by exactly one key. An example for such a classifier is the predefined *type* classifier within AntTracks, which identifies a heap object based on its type's name. Figure 3 shows how this classifier would look like if it were implemented by the user.

```
public String classify(Space space, long address,
  long[] pointsTo, ObjectInfo obj) {
  return obj.type.name;
}
```

Figure 3: A one-to-one classifier that classifies objects based on their *type*

`ObjectFilter`s are a special kind of *one-to-one classifiers*, which classify each heap object by a `boolean` value. This defines whether a heap object should be further processed.

3.2.2 One-to-many Classifier

A *one-to-many classifier* classifies a heap object by multiple keys. An example for such a classifier is the predefined *feature* classifier within AntTracks. By using a feature-to-code mapping tool, a mapping between object allocations and 0...∗ features may be created (e.g., allocations at code location x belong to feature #1, #4 and #7). The *feature* classifier processes this mapping for every heap object and returns an unordered list of features to which its allocation site belongs. Another example for a *one-to-many classifier* is the one that can be seen in Figure 4, which classifies objects based on the *authors of the package* in which their type was declared. Note that an unordered `String` *array* is returned, instead of a single object as in the case of a one-to-one classifier.

```
public String[] classify(Space space, long address,
  String[] pointsTo, ObjectInfo obj) {
    Type type = obj.type;
    if(type.package.startsWith("foo"))
      return new String[] {"mw", "pl"};
    if(type.package.startsWith("bar"))
      return new String[] {"pl", "hm"};
    return new String[] {"mw", "pl" , "hm"};
  }
```

Figure 4: One-to-many classifier that classifies objects based on their *package's authors*

3.2.3 One-to-hierarchy Classifier

A *one-to-hierachy classifier* is typically used if a heap object is classified by multiple keys in a hierarchical fashion (i.e., keys with a parent-child relation). Such a classifier returns an ordered array, where the object at index 0 is the root, and for all $n > 0$ the object at index $n-1$ is the parent of the object at index n. An example for a *one-to-hierarchy classifier* is the predefined *allocation site* classifier within AntTracks, which classifies an object by its allocation site and the allocation's call sites. The root object (at index 0) is the code location where the object was allocated, the object at index 1 the code location from where the allocating method was called, and so on. For each object, AntTracks collects a partial call stack, containing the first few call frames. An example implementation for the *allocation site* object classifier can be seen in Figure 5. Note that an ordered `String` *array* is returned.

```
public String[] classify(Space space, long address,
  long[] pointsTo, ObjectInfo obj) {
  AllocationSite as = obj.allocationSite;
  return as.toString() +
         Arrays.stream(as.callSites)
         .map(cs -> cs.toString())
         .toArray(String[]::new);
}
```

Figure 5: One-to-hierarchy classifier that classifies objects based on their *allocation sites*

3.3 Classification Tree

Grouping based on multiple classifiers results in a classification tree. The tree's root has a child node for ever distinct classification key returned by the first classifier, while each of this nodes has a child node for every distinct classification key returned by the second classifier, and so on. Therefore, for each $n > 0$, each node on level n represents a group of objects that have been classified with the same keys by the classifiers on all levels $< n$. An exception are *one-to-hierarchy classifier*s, which may add multiple levels at once within the tree.

For example, assuming a list of n objects, i.e., $O(1)$, $O(2)$, ..., $O(n)$, the user chooses to multi-group based on three classifiers, i.e., age, author and allocation site (AS) including call sites. The grouping algorithm classifies one object after another based on the selected classifiers and merges the result into the classification tree. Processing the following list of objects yields the classification tree in Figure 6.

Object (Input): Classification keys (Result)
O(1): Age(1), Author("mw", "pl"), AS(add:4, A:480)
O(2): Age(1), Author("mw", "pl"), AS(add:4, B:110, D:88)
O(3): Age(3), Author("mw"), AS(main:44, C:520, A:100)
O(4): Age(3), Author("mw"), AS(clone:16, D:172)
O(5): Age(3), Author("mw"), AS(main:44, C:520, A:100)
O(6): Age(1), Author("mw", "pl"), AS(add:4, A:480)

Rectangles (yellow) represent tree nodes and smoothed rectangles (blue) represent the data a node is holding. This tree can then be visualized hierarchically, first grouping by age, then by author, and finally by allocation site, showing information such as the number of objects, the number of bytes and the average object size for each node.

3.4 Dynamic loading

Since classifiers are instances of Java classes, their classes have to be loaded before the classifiers can be applied. We

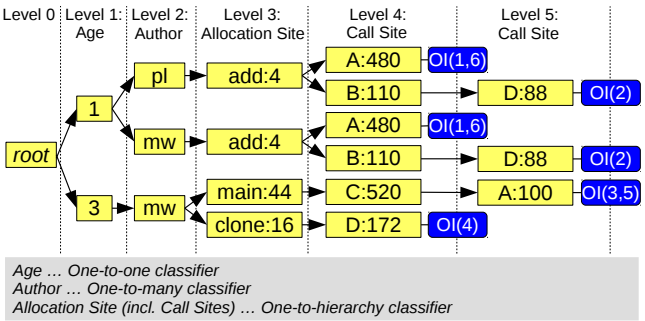

Figure 6: A sample *classification tree* based on an age classifier, author classifier and allocation site classifier

implemented two ways to dynamically load user-defined classifier classes at run-time: (1) Utilizing Java Service Provider Interfaces (SPI), and (2) in-memory on-the-fly compilation.

Java SPI defines a way how to provide services and how to look for them in the environment. First, a service interface has to be provided, i.e., the `ObjectClassifier` interface. Based on this interface, concrete services, i.e., user-defined classifiers, can be developed by the user and third-party developers. These services can then be detected and loaded at runtime using Java's `ServiceLoader` class.

For on-the-fly compilation, we take advantage of the `JavaCompiler` class, which allows compilation of code at runtime. To prevent the generation of class files (the default behavior), we implemented a custom `JavaFileManager` and a `ClassLoader` that manage the compilation of user-defined classifier in-memory at runtime.

4. USE CASES

Use cases for user-defined classifiers include all tasks where memory classification and analysis is needed. For memory leak detection, classifiers based on various metrics could be implemented, such as the number of objects reachable via a classified object or the GC size (i.e., the number of bytes that will be released if the object is reclaimed by the garbage collector). Moreover, user-defined classifiers may also be used for tasks outside the area of memory leak detection, such as change impact analysis. Classifiers are so versatile that one could think of a classifier that groups objects based on the source code repository change history (e.g., to detect if changes at certain code locations had an impact on the memory behavior).

5. RELATED WORK

The *Object Query Language (OQL)*, developed by the Object Data Management Group, is a SQL-like query language used to query objects from object-oriented databases. The downside of OQL is its complexity, which results in the problem that no vendor implements the whole standard. For example, the Eclipse Memory Analyzer (MAT) as well as VisualVM, two popular memory analysis tools, only allow queries in the form of `SELECT <select clause> FROM <from clause> WHERE <where clause>`, without further grouping. *Where clauses* can be represented in our approach using filters, while *select clauses* can be represented using an object classifier. Multi-grouping, as supported in our approach, is neither possible in MAT nor in VisualVM.

6. FUTURE WORK

AntTracks also supports heap diffing, e.g., analyzing object-level changes within the heap during a certain time span. While currently the same grouping and classification methods are available for heap states as well as heap diffing, heap-diffing-specific information may increase the potential applications of heap diffing classifiers. The same applies to feature classification, to ease the integration in memory monitoring tools based on features and product variants, as done by Lengauer et al. [2].

While pointers are currently included as `long[]` parameter in an `ObjectClassifier`'s `classify` method, this only allows for restricted pointer analysis. We plan to develop a more efficient data structure that allows for more sophisticated pointer-based classification.

Finally, all heap query features may be combined in a DSL for memory object querying.

Detailed evaluation is also part of future work. The performance can be compared to a *Java Stream-based* approach. To analyze AntTracks's applicability and effectiveness in memory leak detections compared to other tools, technical metrics such as *task completion time* or *number of found memory leaks* and subjective metrics such as *user satisfaction* can be collected during a user study, based on faulty benchmark implementations or industry applications.

7. CONCLUSIONS

In this paper we presented a user-centered memory analysis technique that builds up on *user-defined classifiers*, a novel way to classify heap objects.

This versatile classification method, in combination with heap filtering and multi-level grouping, results in a classification tree, representing matching groups of heap objects, that allows stepwise data analysis, from a coarse overview to in-detail information. This allows more flexible and user friendly analysis of an application's memory behavior.

Beside broadening the field of memory monitoring to allow classification of heap objects on arbitrary properties, even those that are untypical or not supported in state-of-the-art tools, user-centric memory analysis may change the way how memory monitoring tools are designed. It opens new possibilities on how to visualize the recorded information, allowing for new visual exploration and analysis techniques to be applied.

8. ACKNOWLEDGMENTS

This work was supported by the Christian Doppler Forschungsgesellschaft and by Dynatrace Austria.

9. REFERENCES

[1] V. Bitto, P. Lengauer, and H. Mössenböck. Efficient rebuilding of large java heaps from event traces. PPPJ '15.

[2] P. Lengauer, V. Bitto, F. Angerer, P. Grünbacher, and H. Mössenböck. Where has all my memory gone?: Determining memory characteristics of product variants using virtual-machine-level monitoring. VaMoS '14.

[3] P. Lengauer, V. Bitto, S. Fitzek, M. Weninger, and H. Mössenböck. Efficient memory traces with full pointer information. PPPJ '16.

[4] P. Lengauer, V. Bitto, and H. Mössenböck. Accurate and efficient object tracing for java applications. ICPE '15.

[5] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. KDD '10.

[6] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. OOPSLA '07.

[7] M. Peiris and J. H. Hill. Automatically detecting "excessive dynamic memory allocations" software performance anti-pattern. ICPE '16.

[8] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: Portable production of complete and precise gc traces. ISMM '13.

[9] C. U. Smith and L. G. Williams. Software performance antipatterns. WOSP '00.