Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring

Markus Weninger
Institute for System Software,
CD Labor MEVSS,
Johannes Kepler University
Linz, Austria
markus.weninger@jku.at

Elias Gander
CD Labor MEVSS,
Johannes Kepler University
Linz, Austria
elias.gander@jku.at

Hanspeter Mössenböck
Institute for System Software,
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Complex software systems often suffer from performance problems caused by memory anomalies such as memory leaks. While the proliferation of objects is rather easy to detect using state-of-the-art memory monitoring tools, extracting a leak's root cause, i.e., identifying the objects that keep the accumulating objects alive, is still poorly supported. Most state-of-the-art tools rely on the dominator tree of the object graph and thus only support single-object ownership analysis. Multi-object ownership analysis, e.g., when the leaking objects are contained in multiple collections, is not possible by merely relying on the dominator tree. We present an efficient approach to continuously collect GC root information (e.g., static fields or thread-local variables) in a trace-based memory monitoring tool, as well as algorithms that use this information to calculate the *transitive closure* (i.e., all reachable objects) and the *GC closure* (i.e., objects that are kept alive) for arbitrary heap object groups. These closures allow to derive various metrics for heap object groups that can be used to guide the user during memory leak analysis. We implemented our approach in AntTracks, an offline memory monitoring tool, and demonstrate its usefulness by comparing it with other widely used tools for memory leak detection such as the Eclipse Memory Analyzer. Our evaluation shows that collecting GC root information tracing introduces about 1% overhead, in terms of run time as well as trace file size.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; *Software defect analysis*; • **Information systems** → *Clustering and classification*; • **Theory of computation** → Data structures design and analysis;

KEYWORDS

Memory Monitoring, Memory Leak, Pointer Analysis, Garbage Collection, Graph Closure,

ManLang'18, September 12–14, 2018, Linz, Austria

ACM Reference Format:
Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3237009.3237023

1 INTRODUCTION

Modern programming languages such as Java or C# rely on garbage collection to relieve the programmer from freeing allocated memory manually. The garbage collector (GC) tries to free heap space by removing dead objects that are not reachable from root objects anymore. Examples for root objects are objects referenced by GC roots such as static fields or thread-local variables. Since these objects cannot be reclaimed by the GC, they also keep alive all other objects that are directly or indirectly referenced by them. This reveals one of the drawbacks of garbage collection. Since programmers are no longer required to free memory manually, they tend to use object allocations more carelessly. However, even in garbage-collected languages, careless handling of memory can lead to anomalies such as memory leaks. A memory leak occurs when objects that are not needed anymore are still unintentionally reachable and can therefore not be garbage collected.

Detecting the presence of a memory leak is often relatively easy. A simple chart displaying an application's growing memory usage may be enough to detect such a leak. Yet, it is difficult to track down a memory leak's root cause, i.e., to identify which objects are leaking and which objects are responsible for that by keeping the leaking objects alive. Most state-of-the-art memory monitoring tools analyze the heap based on the object graph in conjunction with its dominator tree [25]. The object graph is a representation of the heap state, where each object is represented as a node, and the references between objects are represented as edges. The dominator tree of an object graph describes a *keeps-alive relationship* between the objects. If an object A dominates an object B and A is collectible by the GC, also B is collectible. While the dominator tree may help to find memory leaks where a single object is responsible, e.g., when all leaking objects are contained in a single large list, it fails to provide insight in situations where multiple objects are keeping objects alive (e.g., in multiple collections) [6].

AntTracks [4, 21–23] is a memory monitoring tool for Java based on the Java Hotspot™ VM [30]. During an application's execution, it records various events such as object allocations, object moves during garbage collection, or pointer information. Based on such

trace files, the heap can be reconstructed for any garbage collection point in time. For inspecting the live objects at such points, Weninger et al. [42, 43] presented the concept of object classifiers and multi-level grouping, which enable the user to classify and group heap objects on multiple levels based on arbitrary grouping criteria.

In this paper, we extend their work by presenting a novel approach to collect and use information about GC roots and the object graph in order to guide users in finding the root causes of memory leaks. First, our approach encompasses an efficient technique to collect GC root information of different kinds (e.g., of static fields or thread-local variables) in a trace. Then, we show how to use this information to calculate object closures. AntTracks is able to calculate closures for arbitrary heap object groups, not just for single objects as other approaches that rely on dominator-tree-based analysis. The *transitive closure* encompasses all objects reachable from the given object group. The *GC closure* encompasses all objects kept alive by the given object group, i.e., those objects that could get garbage collected if the given object group was freed. Based on these closures, metrics such as the *transitive size* and the *retained size* can be calculated. We show how these metrics, in combination with AntTracks's user-driven classification system, can be used to detect memory leaks. Finally, we show in a quantitative evaluation based on three different well-known benchmark suites that collecting GC root information introduces about 1% overhead in terms of run time and trace file.

Thus, our scientific contributions are

(1) a concept to integrate information about GC roots into a trace-based memory monitoring tool such as AntTracks,
(2) algorithms to calculate the transitive closure and the GC closure of a single heap object as well as of heap object groups to derive meaningful metrics,
(3) various techniques to use this information in top-down and in bottom-up memory analysis,
(4) a quantitative evaluation of the tracing overhead and a functional evaluation of our approach based on typical memory anomaly detection use cases.

The paper is organized as follows: Section 2 provides the background of our work, Section 3 describes our approach as well as its concepts and techniques, Section 4 provides details on the implementation of these concepts in AntTracks, Section 5 presents a quantitative and a functional evaluation, Section 6 discusses related work, Section 7 outlines possible future work and discusses threats to validity, and Section 8 concludes the paper.

2 BACKGROUND

AntTracks consists of the *AntTracks VM*, a virtual machine based on the Java Hotspot™ VM [30], and the *AntTracks Analyzer*, a memory analysis tool. The AntTracks VM records memory events into trace files, which can then be analyzed offline with the tool. Since the concepts presented in this paper have been integrated into AntTracks, it is essential to understand AntTracks's architecture and workflow alongside basic garbage collection mechanisms.

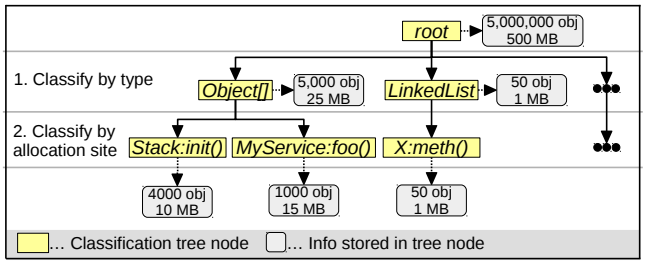

Figure 1: A heap state, consisting of 5 million heap objects, first classified by type followed by allocation site.

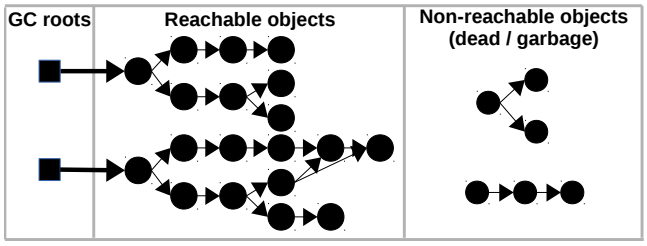

Figure 2: All objects directly or indirectly reachable by GC roots are considered live. Other objects are considered dead or garbage, and may be collected by the GC.

2.1 AntTracks VM: Trace Recording and Reconstruction

The AntTracks VM records memory events, e.g., events for object allocations and object movements executed by the GC, and writes them into trace files. After parsing such a trace file, the AntTracks Analyzer provides an overview of the memory behavior over time and can reconstruct the heap state for every garbage collection point by incrementally processing the events in the trace. A heap state is the set of heap objects that were live in the monitored application at a certain point in time. For every heap object, a number of properties can be reconstructed, including its address, its type, its allocation site, the heap objects it references, and the heap objects it is referenced by.

2.2 AntTracks Analyzer: Memory State Analysis

The AntTracks Analyzer uses user-defined object classifiers and multi-level grouping [42, 43] to enable user-driven heap state analysis. Object classifiers classify heap objects based on certain criteria such as their type, their allocation site, their allocating thread, and so on. For example, the *Type* classifier classifies a heap object based on its type's name, e.g. `java.util.HashMap`. Multi-level grouping is the process of applying multiple classifiers to a collection of objects (i.e., to a heap state), and grouping these objects based on the classification results into a hierarchical classification tree.

A typical example in AntTracks is to first group all heap objects by their types (using the *Type* classifier) and then by their allocation site (using the *Allocation Site* classifier). Figure 1 shows such a classification tree. Yellow rectangles represent tree nodes, and gray rounded rectangles represent data about all heap objects that were classified by the respective tree branch (basically the number of objects and the number of bytes).

2.3 Garbage Collection Roots

Certain objects in the heap are so-called *root objects*. Root objects must not be collected during a garbage collection, and neither must objects be collected that are directly or indirectly reachable from root objects. Whether a heap object is a root object or not is determined by the fact whether at least one *GC root* is referencing the object. There are different kinds of GC roots in Java (and in most other object-oriented languages), the most important ones are:

- *Local variables of threads:* Local variables reside on the call stack of a thread and may reference objects on the heap. Therefore, threads and all objects referenced by local variables of threads are always considered root objects.
- *Static variables:* Static fields of loaded classes are GC roots, i.e., objects referenced by static fields are considered root objects.

Figure 2 shows an example of how GC roots keep reachable objects alive whereas non-reachable objects are eligible for garbage collection.

2.4 Running example

For the algorithms in the following sections we will use a singly-linked list as a running example (see Listing 1). A singly-linked list is represented by its root object of type `LinkedList`. If a list contains at least one object, its `LinkedList` instance points to the head node of the list, an instance of `Node`. Every node may point to another node (its successor) and must point to an instance of `Data`. A `Data` instance contains some integer value and may point to another `Data` instance.

A heap state can be represented as an object graph, i.e., a directed graph in which the nodes correspond to heap objects and references between objects are the edges. Figure 3 shows the visualization of an object graph of two `LinkedList` instances and their referenced objects.

```
class LinkedList {
    private Node head;
    // ...
}
class Node {
    Node next;
    Data data;
    Node(Data d) { data = d; }
    // ...
}
class Data {
    int value;
    Data other;
    Data(int v) { this(v, null); }
    // ...
}
```

Listing 1: Code example for a singly-linked list

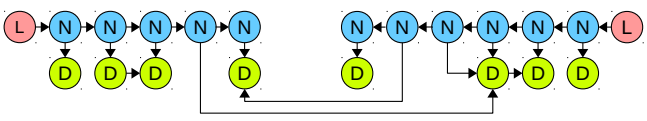

Figure 3: Object graph of two `LinkedLists` (`L`, red), one with five `Nodes` and one with six `Nodes` (`N`, blue) that point to eight `Data` instances (`D`, green).

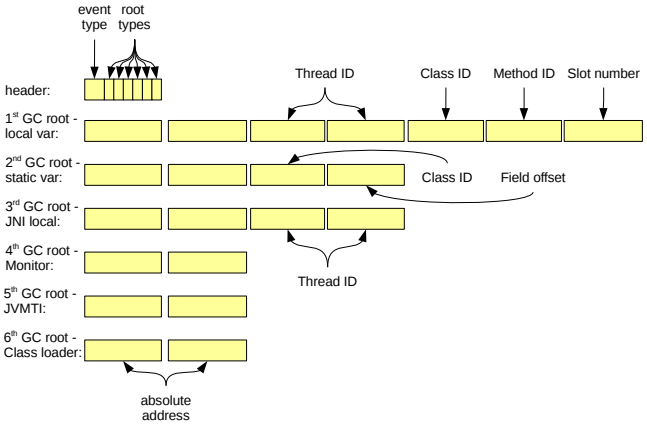

Figure 4: Example of a GC root event (with a size of $22 * 4 = 88$ *bytes*) that contains information about a local variable, a static field, a JNI GC root and 3 other GC roots without additional information.

3 APPROACH

In this section, we show which information can be retrieved from GC roots and how it can be integrated into our trace-based memory monitoring approach. We further present algorithms to calculate the transitive closure and the GC closure for arbitrary heap object groups. Finally, we discuss how closures and GC root information can be used as a guidance to users when tracking down memory anomalies and memory leaks.

3.1 Retrieving GC Root Information

There are two kinds of information that can be retrieved from GC roots: (1) information about the GC roots themselves and (2) information about the references between the objects that are directly or indirectly reachable from the GC roots. Approaches that rely on snapshot-based analysis create heap dumps, which contain information about the objects that were live at a certain point in time as well as the references between them and GC root information. However, if we are not just interested in the heap state at a single point in time, but in the development of the heap over time, heap dumps may prove insufficient.

Instead of creating multiple heap dumps, which would introduce enormous run time overhead, continuous tracing approaches such as AntTracks produce trace files. These allow the offline reconstruction of heap states for arbitrary garbage collection points in time. Trace files contain a sequence of events that have a certain encoding and are only allowed in a certain order. Lengauer et al. [21, 22] present a general event format to trace objects and the references between them. In the following, we propose an additional *GC root event* to trace information about GC roots and root objects.

GC root events are written before any other event at the start of every garbage collection. The format of a GC root event is shown in Figure 4, where each block represents a word of 4 bytes. Like every other event in AntTracks's event format, a GC root event starts with a 1-byte event type. The remaining 3 bytes of the first word are filled with six 4-bit numbers, each of which indicates the type of one GC root that is encoded in the event (see Table 1 for the different GC root types) or is set to 0 if less than six GC roots

ID	Type	Description	Info encoded in event
1	Class Loader	GC root referencing a class loader	-
2	Static Field	GC root representing a static field referencing a heap object	class id, field offset
3	Thread	GC root referencing a thread	thread id
4	Local variable	GC root representing a local variable referencing a heap object	thread id, class id, method id, slot number
5	Code Blob	GC root referencing a code object (e.g. a JIT-compiled method)	class id, method id
6	JNI	GC root referencing a heap object created via JNI	thread id
7	JVMTI	GC root referencing a heap object created via JVMTI	-
8	Monitor	GC root referencing a monitor object used for synchronizing	-
9	Management	GC root referencing internal objects used by MXBeans	-
10	Others	GC root referencing other JVM internal object (e.g., profilers)	-

Table 1: The different GC root types in the JVM.

are encoded in the event. Thus, the number of GC roots that can be encoded in a single GC root event is limited to six. For every GC root encoded in the event, the address of the referenced root object as well as additional information (depending on the type of the GC root) is appended after the header word (see Figure 4).

The types of the different GC roots encoded in the header word are necessary to correctly interpret the given GC root's information. As shown in Table 1, depending on the type of each GC root, additional information is added to the event's payload (the size of the different IDs is determined by the JVM):

- A *static field root* adds a 4-byte class ID and a 4-byte field offset.
- A *thread root* adds an 8-byte thread ID.
- A *local variable root* adds an 8-byte thread ID (i.e., the thread that holds the local variable), a 4-byte class ID (i.e., the class in which the local variable is defined), a 4-byte method ID (i.e., the method in which the local variable is defined) and a 4-byte slot number (i.e., the local variable stack index).
- A *code blob root* adds a 4-byte class ID and a 4-byte method ID.
- A *JNI local root* adds an 8-byte thread ID.

The connection between a thread's ID and its name is established by a separate event that is recorded whenever a new thread is started. This information can later be used to resolve a thread ID into the thread's name. Similarly, class IDs, method IDs, field offsets and slot numbers can be resolved to their identifiers and type signatures using symbol information that is written to a separate *symbols file* during program execution.

Based on local variable GC roots, the call stacks of all threads at the time of the garbage collection can be reconstructed. This is possible because (1) each local variable root stores the method and thread they belong to and (2) they are written in the same order as they are encountered when traversing their thread's stack frames from top to bottom. This excludes stack frames that do not contain at least one local variable.

The amount of information differs among GC root types, thus the total event size is variable. However, the maximum size of a GC root event is 43 words (336 bytes) which is reached in the case of 6 local variable GC roots.

Since GC roots are recorded at the start of every garbage collection, their referenced objects may be moved during garbage collection. AntTracks also records these object movements. When encountering a matching move event while parsing a trace file, GC roots have to be updated to the move's destination address.

3.2 Heap Object Closures

In general, a heap object closure is the set of heap objects that are directly or indirectly reachable from a given heap object or from a group of heap objects. A heap object closure may contain *all* reachable objects (i.e., the transitive closure) or only those that satisfy certain criteria (e.g., in the case of the GC closure). In this section, we show how to calculate closures for arbitrary object groups. This allows us to detect leaking objects that are kept alive, even by more than one object. Metrics derived from the closures such as the *retained size* can be displayed to help and guide users during heap state analysis. Closure calculations, as well as metrics derived from these closures, have been integrated into AntTracks. Nevertheless, this techniques could also be integrated into other heap profiling tools such as Elephant Tracks [39], as long as they are able to reconstruct heap object graphs, i.e., they record the heap objects themselves, pointers between them, as well as garbage collection roots.

3.2.1 Transitive Closure. The transitive closure [20, 41] of a single node in a graph is made up of the node itself and all other nodes that are directly or indirectly reachable from this node following all edges. Its calculation has been well studied, e.g., by Eve and Kurki-Suonio [13].

Instead of calculating the transitive closure of a single object, we argue that it is also useful to calculate the transitive closure of an object group. We call the objects for which we calculate a closure the *closure root objects* (which are not to be confused with *root objects*, i.e., objects that are referenced by GC roots). The transitive closure of an object group is made up of the closure root objects themselves and all other objects that are directly or indirectly reachable from at least one of the closure root objects. Figure 5 shows the transitive closure (gray background), first for a single closure root object (thick border), and second for a group of two closure root objects. The transitive closure is calculated based on the pseudocode presented in Listing 2.

Section 4.2 discusses how the transitive closure algorithm is computed in AntTracks based on AntTracks's internal heap state data structure.

The transitive closure can be used to detect objects that reference an unexpectedly large amount of other objects since it describes how many objects are *reachable* from the given closure root objects. Analogously, an object group with a small transitive closure, i.e., an object group that does not reach many other objects, can never be the root cause of a memory leak. In AntTracks, heap objects are separated into object groups that get arranged in a classification

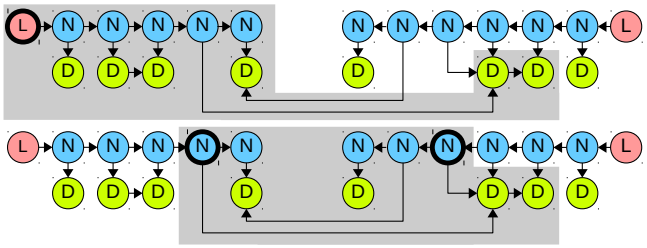

Figure 5: Two examples showing the transitive closures when using the thick bordered object(s) as closure root objects.

tree based on their classification results. If the user is only interested in finding memory leak roots, i.e., objects that keep a lot of other objects alive, tree nodes of object groups with small transitive closures can be hidden and excluded from further analysis. This reduces the tree's complexity, speeds up further computations, and thus eases the overall analysis process. Nevertheless, a large transitive closure does not automatically identify a leaking data structure or object group. Just because objects are reachable from a given set of closure root objects does not mean that only the closure root objects keep them alive. Therefore, the transitive closure can be used to further calculate the *GC closure*, a closure that describe the closure root objects' *ownership* over other objects.

3.2.2 GC Closure / Retained Closure. The transitive closure contains all objects reachable from the closure root objects. In contrast, the GC closure contains all objects that (1) are reachable from the given closure root objects *and* (2) could be garbage collected if all closure root objects were collected. Thus, the GC closure describes *object ownership*, i.e., all objects that are kept alive by a given set of closure root objects. Being able to detect heap object groups that keep large amounts of other objects alive bears large potential in helping users to resolve memory leaks.

To calculate the GC closure of a *single object*, the object graph's dominator tree can be used. In graph theory, a node d dominates a node n if every path from the root to n must pass through d [40]. Figure 6 shows the dominator tree of a sample object graph of two singly-linked lists. Some of the data is shared between the two lists, i.e., $D3$, $D4$ and $D5$, which results in those three objects not having a dominator. For example, assuming that node $N8$ could be collected by the garbage collector (by removing all references to it), all child objects in its dominator subtree (i.e, $D8$, $N9$, and $N10$) could be

```
// called with closure root objects as initial work list
transitiveClosure(List workList) {
  List closure = new List();
  mark all objects in workList as visited;
  foreach (obj in workList) {
    add obj to closure;
    foreach (child referenced by obj) {
      if(child is not yet marked as visited) {
        mark child as visited;
        add child to workList;
      }
    }
  }
  return closure
}
```

Listing 2: Pseudocode calculating the transitive closure for a given closure root object group

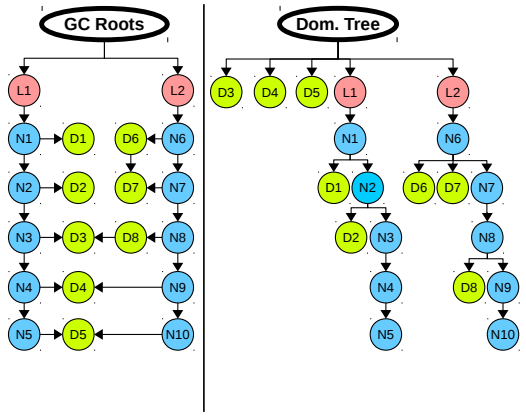

Figure 6: Object graph of two singly-linked lists and its dominator tree.

collected, too. However, it is not possible to use the dominator tree to answer which objects could be freed if a certain object group, e.g., $N4$ *and* $N9$, would be freed (which would be $N5$ and $N10$, but also $D4$ and $D5$).

While the dominator tree and its algorithms are well-studied, they are only suited to analyze single nodes and to detect the maximum *unique ownership* [27] within a graph, e.g., what happens if one specific heap object could be freed by the GC. We are therefore presenting a new algorithm that is able to calculate the GC closure for arbitrary closure root object groups.

Simple Approach. Our approach assumes that an heap object group's transitive closure is already known, e.g., by using the algorithm presented in Section 3.2.1. This transitive closure can then be reduced to the GC closure by following these steps:

(1) The initial GC closure is set to the transitive closure.
(2) To determine which objects could be freed if the closure root objects get freed, we have to simulate that the closure root objects are not reachable from root objects anymore. Thus, we ignore all references to these objects (i.e., assuming that all references to the closure root objects have been set to `null`).
(3) Then, the heap is recursively traversed, starting at the heap's root objects (i.e., the objects directly referenced by GC roots), visiting every reachable object exactly once. If the currently visited object is part of the GC closure (which has been initialized to the transitive closure, see Step 1), the current object and all objects reachable from it are removed from the closure. This is done because these objects would still be reachable from the heap's root objects and thus be kept alive, even if the closure root objects would be freed. Visited and removed objects are marked to avoid processing them multiple times.

At the end of this algorithm, the GC closure contains only those objects that are not referenced by GC roots from outside the transitive closure. The object in the GC closure are the objects that could be freed if the closure root objects were released.

A major problem with this approach is that its complexity depends on the object graph size, i.e., the number of objects in the heap, since the heap traversal starts at the root objects. Yet, with

minor adjustments to the algorithm, the complexity can be reduced to only depend on the object group's transitive closure size.

Improved Approach. The performance problems of the simple approach can be tackled by using the following technique. When a heap state is reconstructed from a trace file, we traverse the object graph once (instead of traversing the object graph on every GC closure calculation in the simple approach). Starting at the GC root objects, every visited object is marked. This allows us to identify all objects either as live (if they have been visited) or as dead (if they have not been visited). Since this additional information can be stored as a single bit per object (i.e., 0 if the object is not reachable from any GC root or 1 if it is) the additional memory overhead is negligible. For example, this information would take up around 12.5 MB for a heap state of 100, 000, 000 objects). The improved approach works as follows:

(1) The initial GC closure is set to the transitive closure.
(2) To determine which objects could be freed if the closure root objects get freed, we have to simulate that the closure root objects are not reachable from root objects anymore. Thus, we ignore all references to these objects (i.e., assuming that all references to the closure root objects have been set to `null`).
(3) The GC closure (which has been initialized to the transitive closure) is recursively traversed, starting at the closure root objects, visiting every object in the closure exactly once. The current object is checked for the following criteria:
    - Is it directly referenced by a GC root?
      This can easily be checked since we know every GC root.
    - Is it referenced by a live object that is not part of the closure?
      AntTracks enables access all objects that reference a given object (pointed-from analysis). For each of these referencing objects it can be checked whether it is part of the closure, and whether it has been marked as live during the heap state reconstruction.

    If at least one of the mentioned criteria holds, the current object and all objects reachable from it are removed from the closure. Visited and removed objects are marked to avoid processing them multiple times.

Instead of having to traverse the whole heap, the algorithm's complexity now only depends on the transitive closure's size. Figure 7 shows how objects that are reachable from live objects outside the transitive closure have to be removed to reduce the transitive closure to the GC closure.

3.3 Metrics

A single object's shallow size is the size of the object itself. This encompasses the object's header and its data, without taking into account any referenced objects. The shallow size of an object group is the sum of the shallow sizes of all contained objects. Using the transitive closure, the *transitive object count* (also called *deep object count*) as well as the *transitive size* (also called *deep size*) can be calculated. The *transitive object count* is the number of objects contained in the transitive closure, while the *transitive size* is the sum of the shallow sizes of all objects in the transitive closure. Similarly, using the GC closure, the *retained object count* as well as

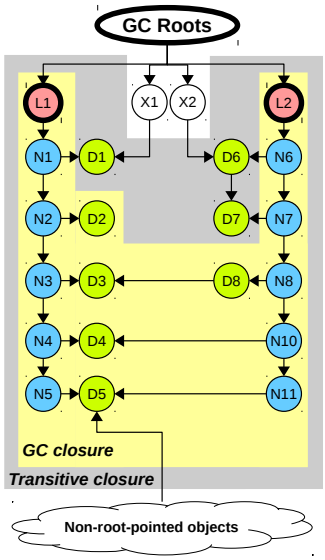

Figure 7: GC closure for $L1$ and $L2$, derived from the transitive closure and GC root information.

the *retained size* can be calculated. The *retained object count* is the number of objects contained in the GC closure, i.e., the number of objects that could be freed if the closure root objects were released, while the *retained size* is the sum of the shallow sizes of all objects in the GC closure.

As explained in Section 2, AntTracks applies user-specified object classifiers to aggregate heap objects in a classification tree (see Figure 1). Without information about an object group's transitive closure and GC closure, only the group's *object count* and *shallow size* could be calculated and displayed to the user. While this information is good enough to learn about the system under investigation (e.g., about the most frequently allocated types, the hot allocation sites, the object allocations per thread, etc.), it is not well suited to track down memory leaks. So far, the users' decisions on which object groups to analyze in more detail (e.g., by applying further classifiers) often strongly depended on prior knowledge about the system under analysis, e.g., by having an educated guess on what to look for. For example, a user unaware of the internals of a given system may not suspect a possible memory leak when discovering that there was a single instance of type A with a small shallow size. However, knowing that objects of type A are very complex (i.e., may keep a lot of other objects alive) and are not expected to be alive in the given heap state would lead to further investigation.

Thus, we extended AntTracks to also display the deep size as well as the retained size for every tree node in a classification tree. Even without knowing anything about the system under investigation, the user is now able to detect object groups that keep a lot of other objects alive. An evaluation of this feature will be described in Section 5.1.

3.4 Utilizing GC Root Information

In some cases, one might be aware of a group of objects that occupy a large portion of the heap. For example, by analyzing multiple heap states, we may detect that objects of a certain type that are

allocated at a certain allocation site accumulate over time. To free their memory, these objects must be made eligible for garbage collection. This can be done either by freeing all root objects that reference the object group or by cutting all connections to them. Therefore, bottom-up analysis is used to detect possible ways to free memory occupied by a given object group by inspecting the relation between the object group and its referencing root objects. For example, we may detect a suspiciously large object group. By following the object references from the object group back to the GC roots we may detect that all these objects are stored in a list, which in turn is referenced directly or indirectly by multiple GC roots. Ideally, we are able to remove the list head, and in turn the whole object group becomes unreachable and can thus be garbage collected. If this is not possible, suitable cutting points to remove single elements from the list have to be found.

3.4.1 Root Object Classification. Assume that we detect a suspicious object group. As a first step, we obtain all root objects that reference the object group. Since not every object of the suspicious object group must be referenced by every found root object, we have to detect those root objects that own most of them. This can be done by further analyzing the found root objects, e.g., by applying multiple classifiers on them (e.g., classifying them by their types, packages, allocation sites, etc.). Since the classified object groups can be sorted by their size or by their retained size, we can try to find object groups that are small but still keep a large number of other objects alive. This small number of root objects is then feasible to be traced and dealt within the code.

3.4.2 Bottom-up Visualization. Alternatively, instead of analyzing the root objects directly, one can also trace and visualize all paths from the suspicious object group to their root objects. These paths may reveal possible cutting points to make (parts of) the object group unreachable and thus eligible for garbage collection. To facilitate the task of locating high-impact cutting points, the paths can be displayed as a graph with objects as nodes and their pointers as edges. Additionally, since object groups can be very large, node aggregation would be necessary in many cases. The goal of node aggregation is to combine multiple objects into a single node by detecting typical reference patterns between them. This could, for example, be achieved by merging nodes that belong to the same data structure, which is a goal of our future work. Node aggregation decreases the complexity of the graph and thus enables users to draw meaningful conclusion from it more easily.

4 IMPLEMENTATION

4.1 Retrieving GC Root Information

AntTracks uses a modified Java Hotspot™ VM to detect events (e.g., object allocations) and to write information about these events into trace files. GC root information gets written at the beginning of every garbage collection. Since there are different types of GC roots, various VM-internal data structures are used to retrieve this information. The `ClassLoaderDataGraph` contains information about classes that are loaded by an application's class loaders. Other data structures include the `SystemDictionary` and the `Universe`. These data structures are needed to retrieve information about static fields, since static fields are stored inside Java class instances. To collect information about thread-local variables, we iterate all Java threads and extract each thread's stack trace. Each frame in the stack trace contains a `StackValueCollection` that allows to access information about the frame's local variables. Also, the thread object itself is marked as a root object. Furthermore, we handle `CodeBlobs` as GC roots. These may be JIT-compiled static references to certain addresses. `JNIHandles` and `JvmtiExport` are used to detect objects that are kept alive via JNI and JVMTI. Other types of GC roots are `ObjectSynchronizer` (i.e., monitors), `FlatProfiler` (i.e., a JVM profiling tool), and `Management` (i.e., objects used by MXBeans).

4.2 Closure Algorithms

Section 3.2 presented algorithms to calculate the transitive closure and the GC closure for arbitrary object groups. Implementing these algorithms efficiently is crucial for their application in end-user applications such as AntTracks Analyzer. After evaluating various approaches, we decided to use `BitSets` for the representation of closures. A `BitSet` is a vector of bits that is indexed by a nonnegative object number and keeps track of which objects are part of the closure.

To be able to use `BitSets`, we had to adjust AntTracks's heap data structure. It is now organized in such a way that objects can be identified by a unique number that is used as their index in the bit sets. The heap object at index 0 is the object with the lowest address, while the object at index n (where n is the number of objects in the heap) is the object with the highest address.

Another advantage of bit sets is their ability to be combined using logical operators such as `and` or `or`. For example, using these operators, we do not need to calculate the transitive closure for every node in the classification tree, but only for leaf nodes. Imagine classifying heap objects first by type and then by allocation site. This may result in a classification tree node for two objects of type A (intermediate node), one allocated at allocation site x (leaf node) and one at allocation site y (leaf node). It is sufficient to calculate the transitive closures for the x node and the y node based on the object graph, since the transitive closure for node A is just the union of the two closure for x and y. The union of these closures can be obtained by or-ing their bit sets.

4.3 Classifiers

In addition to the metrics presented in Section 3.3, we extended AntTracks by a number of new classifiers, which can be freely combined with any other existing classifier to analyze a heap state.

4.3.1 Directly-GC-Rooted Classifier. This classifier categorizes heap objects based on the GC roots by which they are directly referenced and splits them into multiple groups (e.g., *Root: Static field* or *Root: Thread-local variable*). Each of these groups is then split into further subgroups, depending on the root type, to provide further information. For example, objects in the group *Root: Static field* are further split by the static fields' classes, and then by the static fields themselves (as shown in Figure 8). The classifier can also be configured to only show variables (static fields, thread-local variables and JNI locals) instead of all roots. In addition, the user can chose if objects that are only indirectly referenced by GC roots or not reachable from GC roots at all should be shown in a separate group or be completely hidden.

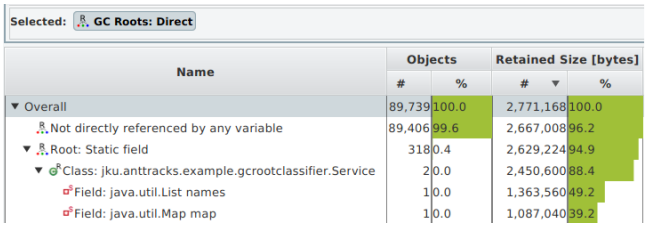

Figure 8: A heap state in AntTracks, classified using the directly rooted classifier.

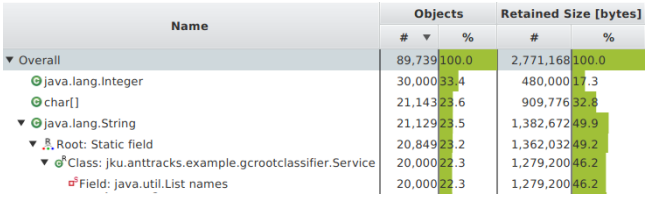

Figure 9: A heap state in AntTracks, classified using the type classifier followed by the indirectly rooted classifier.

Figure 8 shows an example for the use of the *directly-GC-rooted* classifier in AntTracks. The first line (with the key *Overall*) shows how many objects the heap contains. Its first child (with the key *Not directly referenced by any variable*) contains the 89.406 objects that are not directly root-pointed by a variable. The second child (with key *Root: Static field*) contains 318 objects that are directly root pointed by a static field. Two of them are referenced from a static field in the class `jku.anttracks.example.gcrootclassifier.Service` (4th line), one by the field `names` (5th line) and one by the field `map` (6th line).

4.3.2 Indirectly-GC-Rooted Classifier. Similar to the *directly-GC-rooted* classifier, the *indirectly-GC-rooted* classifier categorizes heap objects based on the GC roots by which they are referenced. However, instead of only taking direct references into account, also indirectly referencing GC roots are considered. The classifier can also be configured to only show variables instead of all roots, as well as to whether objects that are not root-pointed at all should be shown in a separate group or be completely hidden.

This classifier is especially useful when the user encounters an unexpectedly large group of objects that was not assumed to be alive or when such a group has been growing over time. In such situations the *indirectly-GC-rooted* classifier can be used to find out which root objects are keeping them alive. To improve performance, it is also possible to cut the calculation as soon as the first referencing root object is encountered.

Figure 9 shows an example for the use of the indirectly-GC-rooted classifier, following the type classifier. 21, 129 `String` objects exist at the given point in time, of which 20, 849 are reachable from static fields. 20, 000 of these strings are reachable from the static field `names` in the class `jku.anttracks.example.gcrootclassifier.Service`.

4.4 Object Group Inspection Window

The classification mechanism in AntTracks produces object groups that share certain properties based on the selected classifiers (e.g., objects of the same type allocated at the same allocation site). Beside further classifying such object groups (e.g., extending the classification tree by further splitting the objects by their allocating thread), a given object group can also be inspected in more detail in an *object group inspection window*.

A classification tree visualizes a particular heap state by showing the object count, the shallow size, the deep size, and the retained size of each object group. This is well suited for a fast overview. It either enables users to detect small groups of objects with large retained sizes (most probably heads of larger data structures) or large groups of objects (most probably contained in other data structures). The aim of the inspection window is now to provide more detailed information about a particular object group.

First of all, the view shows all root objects from which a selected object group can be reached and which keep the object group alive. These root objects can then be further classified (*root object classification*) or the paths to them can be visualized (*bottom-up visualization*) as described in Section 3.4. Second, the inspection window includes another classification tree showing the objects in the group's transitive closure or in its GC closure. Again, classifiers can be applied to assign the objects in the closures to the branches of the tree (e.g., according to the object types in the closures). This feature might prove useful in helping user to understand the ownership relations between the inspected object group and the closure objects.

5 EVALUATION

To evaluate the usefulness of our analyses we show how one can use the AntTracks Analyzer to detect memory leaks and compare it to a dominator-tree-based state-of-the-art memory monitoring tool, namely the Eclipse Memory Analyzer (MAT) [15].

We also evaluate the overhead that is caused by recording the GC root data in terms of *run time* and *trace file size*. All analyses were performed on well-known benchmarks from the DaCapo suite[1] [5] (version 9.12-bach), DaCapo Scala suite[2] (version 9.12-bach) and the SPECjvm2008 suite[3] (version 1.01).

Setup. All measurements were run on an Intel® Core™ i7-4790K CPU @ 4.00GHz x 4 (8 Threads) on 64-bit with 32 GB RAM and a Samsung SSD 850, running Ubuntu 17.10 with the Kernel Linux 4.13.0-16-generic. All unnecessary services (including graphical user interfaces) were disabled in order not to distort the experiments.

5.1 Functional Evaluation

In this section we are going to evaluate AntTracks's applicability to detect memory leaks and their root causes. The analyzed application, *mulit-cache*, is an artificial demo application to demonstrate AntTracks's ability to detect memory leaks caused by multiple objects, a situation in which dominator-tree-based approaches prove less useful. It stores products in a database which can be identified by a `long id` as well as a `String name`. Once a `Product` instance is queried, it gets stored in two caches (`HashMap`), one to access products via their id and one to access them via their name. Both `HashMaps` are stored in static fields, one in the class `IdCache` and

[1] http://dacapobench.org/(last accessed May 11, 2018)
[2] Info: http://www.scalabench.org/ (last accessed May 11, 2018)
[3] https://www.spec.org/jvm2008/ (last accessed May 11, 2018)

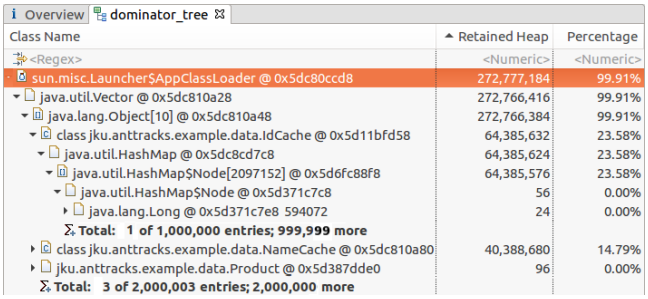

Figure 10: Dominator tree view in MAT.

one in the class `NameCache`. To simulate a memory leak, the caches are never cleared and thus grow over time. We compare AntTracks to the Eclipse Memory Analyzer (MAT), a state-of-the-art memory monitoring tool that employs dominator-tree-based memory leak detection. This comparison should demonstrate how tools that only perform dominator-tree-based analyses are not able to derive the root cause of a memory leak that is caused by multiple objects, i.e., by multi-object ownership.

As a baseline, we analyzed the application using MAT, which provides a hierarchical visualization of the dominator tree. Figure 10 shows the dominator tree view for the multi-cache demo application. It reports that the complete heap (about 273 MB) is kept alive by the `AppClassLoader`, which again holds a `Vector` that holds an `Object[]`, i.e., the loaded classes. The two classes with the largest retained size are the `IdCache` (about 64 MB) and the `NameCache` (about 40 MB). Both keep a `HashMap` alive (only the `IdCache` is shown in detail in Figure 10), which further keeps its `HashMap$Node` instances alive using an array. Yet, these `HashMap$Nodes` only dominate their keys, but not their data. Since the dominator tree is only suitable for detecting single-ownership, it is not possible to infer how much and which memory is kept alive by both caches together, or if further data structures are involved in the memory leak. This is also reflected by the fact that the `Object[]` that keeps the loaded classes alive is the dominator of $1,000,000$ `Product` instances, which is rather useless to infer ownership (this number is not directly visible in Figure 10 but is a part of the very last entry in the figure).

By default, AntTracks classifies all objects by type, which can be seen in Figure 11. The figure shows that the application's 22 `HashMaps` together keep 99.9% of the heap alive. When classifying the maps by allocation site (see Figure 12), the classification tree displays the retained sizes of the individual hash maps (23.6% for the one allocated in `IdCache`, 14.8% for the one allocated in `NameCache`, and 0.0% for the remaining 20 maps). The individual retained sizes do not add up to the combined retained size of 99.9%, which indicates shared ownership between the hash maps. To analyze the shared ownership of the two caches, both have to be combined into one object group, which in turn may then be analyzed. There are two options to do this: (1) Selecting the two rows in the tree and opening the object group inspection window (showing the object group information explained in Section 4.4 for the two maps combined) or (2) applying a classifier that groups the two caches into one node in the classification tree.

Figure 13 shows parts of the object group inspection window that gets displayed when using the first method. The upper part

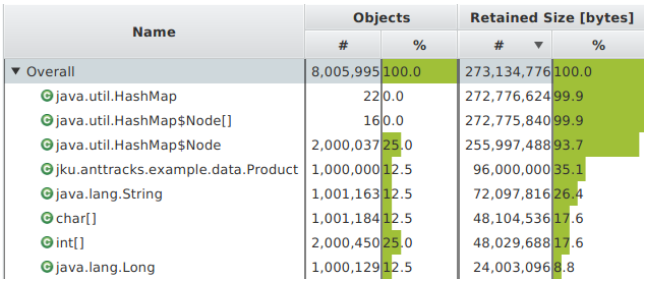

Figure 11: AntTracks's default heap state analysis view, classifying heap objects by type.

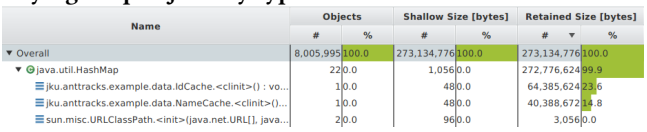

Figure 12: Classifying heap objects by type and allocation site in AntTracks.

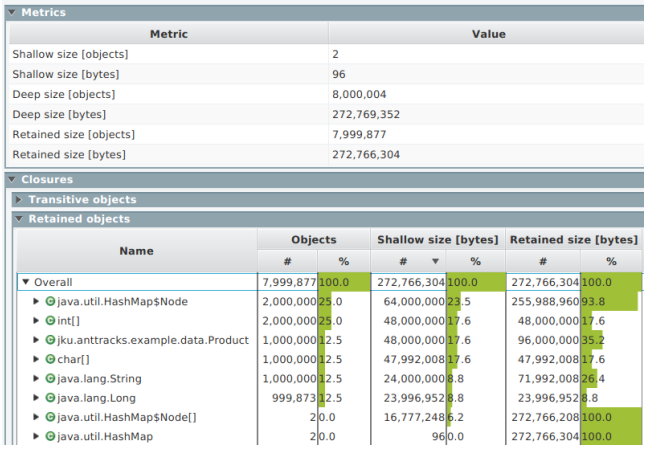

Figure 13: Parts of the object group inspection window displaying information about the hash maps allocated in `IdCache` and `NameCache`

of the figure shows various metrics, including the retained size. A retained size of about $272MB$ confirms our assumption that both caches together nearly keep the whole heap alive. In the lower part, the classification tree (by default classifying objects by their types) of the retained objects is shown. It reveals that, beside other objects, $1,000,000$ `Product` instances are part of the caches' shared GC closure.

When using the second method to analyze shared ownership, a suitable classifier has to be used to group the suspicious objects into one tree node. Since both caches are allocated in the same package, the *allocating package classifier* seems advisable to group them. Additionally, the type classifier, followed by the indirectly-GC-rooted classifier has been applied, and the result can be seen in Figure 14. We can see that $3,000,002$ objects have been allocated in the `jku.anttracks.example.data` package, and all objects together have a retained size of 99.9% of all memory. By classifying these objects by type, we can see that they split up into $1,000,000$ `Product` instances, $2,000,000$ `int[]` instances, and 2 `HashMaps`, i.e., the maps in the caches. The maps also have a retained size of 99.9%, which clearly identifies the two `HashMaps` as the source

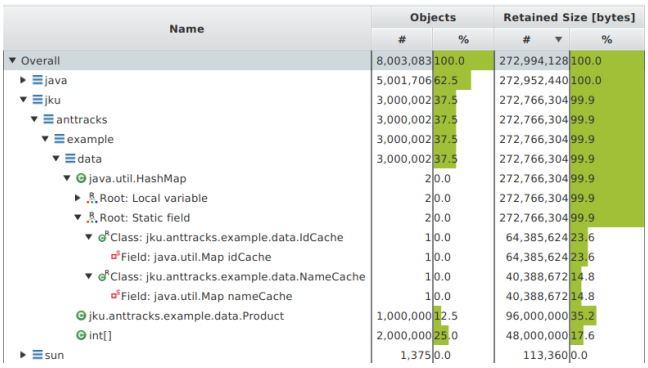

Figure 14: Classifying heap objects by allocating package, type and indirect GC roots in AntTracks.

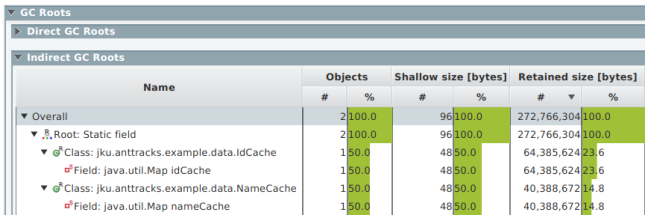

Figure 15: AntTracks's object group inspection window also allows the analysis of GC roots.

of the memory leak. To find out which GC roots keep certain objects alive, one can apply the indirectly-GC-rooted classifier. In the case of the two maps this classifier returns two static fields, one in the `IdCache` class of type `java.util.Map` called `idCache`, and one in the `NameCache` class of type `java.util.Map` called `nameCache`. Even if the user would not have had any prior knowledge about the system under investigation, we claim that this information is enough to find the corresponding locations in the source code to investigate the leak further on source code level.

This memory leak could have also been identified by using bottom-up analysis. By comparing multiple heap states regarding the frequency of types, one could detect that the number of `Product` instances is increasing over time. In Section 3.4, such an object group has been called *suspicious object group*. By opening the object group inspection window for such an object group, we can inspect the closures of the objects as well as the GC roots. The hierarchical view of the GC roots also allows us to define classifiers on them. By default, the root objects are classified using the directly-GC-rooted classifier, categorizing them based on their referencing GC roots (see Figure 15). This again identifies the two `HashMaps` as responsible for the memory leak.

5.2 Recording Overhead

In Section 3.1 we presented the events that are recorded by the AntTracks VM to collect information about GC roots. Information is recorded for every GC root at the start of every garbage collection. Therefore, the recording of this information may have an influence on the following metrics:

- *Run time*: The time it takes to execute a given benchmark.
- *Trace file size*: The size of the resulting trace file.

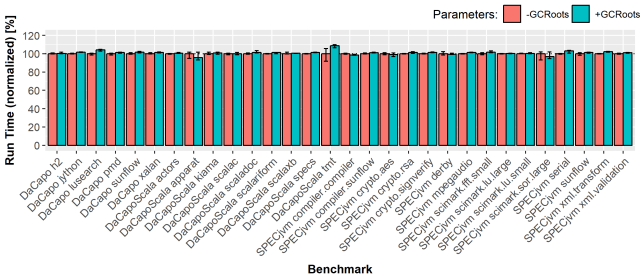

Figure 16: Median run time without (*-GCRoots*) and with (*+GCRoot*) GC root tracing, relative to the median run time without GC root tracing enabled.

The recording does not influence the number of garbage collections since we did not modify the GC's collection behavior, but only perform additional operations at the beginning of each garbage collection.

We evaluate the overhead using benchmarks from the DaCapo suite, the DaCapo Scala suite and the SPECjvm2008 suite, all fixed to a maximum heap size of 2GB (which is enough to run the benchmark with the largest live set, i.e., DaCapo h2) and using the Parallel Old GC. We reduced our selection to benchmarks that trigger at least one garbage collection per run.

As a baseline, we used the AntTracks VM to trace the applications, including pointer information but disabling GC root information tracing (parameter *-GCRoots*). According to Lengauer et al. [21], this introduces an average run time overhead of 15.0%. We measured the additional overhead introduced by enabling GC root information tracing (parameter *+GCRoots*). For warm-up, we ran a benchmark with a given parameter 10 to 40 times on a single VM instance using the largest input size to ensure stabilization of caching and JIT compilation. The largest possible input sizes and the necessary warm-up iterations depend on the benchmarks and have been taken from Lengauer et al., Figure 1 [24]. After warm-up, we ran the benchmark another 10 times on the same VM and calculated the median run time and median trace file size of these runs. We repeated this experiment 10 times for every benchmark and parameter combination, using a new VM instance every time. In the next sections, we report the median, the 25 percentile, and the 75 percentile of the 10 medians per benchmark and parameter setting.

5.2.1 Run time. Figure 16 shows the median run time for each benchmark, relative to the median run time without GC root tracing enabled. The error bars show the 25 percentile and the 75 percentile. On average (geometric mean), the run time increases by 1.00% across all benchmarks when turning GC root tracing on, with an outlier of a maximum median run time increase by 8.77% on the DaCapoScala tmt benchmark. The reason for this outlier may be that tmt is one of the most allocation-intensive benchmarks. The official documentation describes tmt as *externally single-threaded and internally multi-threaded. It creates a large number of threads, each of which is only very short-lived.*[4] Allocating a lot of objects may trigger many garbage collections, which, in combination with a large number of thread-local GC roots, may lead to a run-time

[4] http://www.benchmarks.scalabench.org/modules/tmt-dacapo-benchmark/ (last accessed May 11, 2018)

Figure 17: Trace file size without (*-GCRoots*) and with (*+GCRoot*) GC root tracing, relative to the median trace file size without GC root tracing enabled.

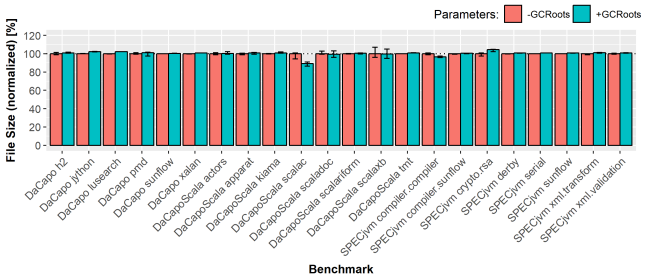

Figure 18: Trace file size without (*-GCRoots*) and with (*+GCRoot*) GC root tracing, relative to the median trace file size without GC root tracing enabled (excluding benchmarks generating trace files < 30MB).

increase. Nevertheless, tmt was the only outlier, and we claim that an average run-time increase of one percent makes our approach feasible for production systems.

5.2.2 Trace file size. Figure 17 shows the median trace file size for each benchmark, relative to the median trace file size without GC root tracing. The error bars show the 25 percentile and the 75 percentile. The extreme trace file size increases for certain benchmarks (over 1000% for SPECjvm scimark.fft.small) have to be set in relation to their absolute file sizes. For most benchmarks, the trace files have sizes between some 100 megabytes and some gigabytes, depending on how many objects are allocated by the application and moved by the garbage collector. All benchmarks that show a large relative increase of their trace sizes have an original trace size of less than 30MB. These benchmarks allocate and move very few objects, and the size of the GC root information exceeds the amount of the trace data for the actual objects.

Figure 18 shows the same data as Figure 17, but excludes all benchmarks that generate trace files smaller than 30MB. On average (geometric mean), the trace file size for these benchmarks increases by 0.21% across all benchmarks when turning GC root tracing on. Since most of the data that is written to the trace files concerns object allocations, object movements and pointer updates, the additional data written due to GC roots becomes nearly negligible.

6 RELATED WORK AND STATE-OF-THE-ART

Based on the dominator tree algorithm proposed by Lengauer and Tarjan [41], various variations have been presented [2, 7, 33]. Dominator trees are used in a wide range of applications. For example, dominance is used in compilers to analyze control flow graphs [10] or to visualize software dependencies [14].

In the domain of program understanding and memory leak detection, dominator trees (alongside other techniques) are often used as basis for ownership detection, component analysis and aggregation in heap graph visualization. Hill et al. [17, 18] reduce object graphs into *ownership trees* for visualization based on the dominator tree. Rayside et al. [34, 35] introduce *object ownership profiling*, a technique that uses the dominator tree alongside information about last object access times and object interactions to identify memory leaks and memory management anti-patterns in applications. Mitchell [27] summarizes an application's memory footprint with help from the dominator relation. He introduces a set of ownership structures, detects these structures in object graphs and aggregates them into concise *ownership graphs* that visualize responsibility and ownership of data structures. In further work, Mitchell and Sevitsky [29] introduce *health signatures* for data structures based on dominator tree reference analysis. They show how to judge data structure designs or implementations based on their relationship between actual data and structural overhead. In [28], Mitchell et al. present aggregation techniques using the domination relation in heap object graphs to perform progressive graph abstractions, alongside corresponding visualizations. Similarly, other approaches use dominator information to abstract object graph visualizations [26, 36–38] or to layout graphs [1].

Nevertheless, ample work in the domain of memory leak detection exists that does not rely on dominator trees but uses object graphs directly, as we do in AntTracks. Most of these approaches rely on certain types of pattern detection in the object graphs. De Pauw et al. [11, 12] extract patterns from object graphs, shifting the focus from individual objects to groups of objects to abstract their visualization. Jump and McKinley [19] developed the memory leak detection tool *Cork* that reduces the object graph to a *type-points-to-type* graph for analysis. Barr et al. [3] use the object graph to detect and classify typical reference patterns in real-world application heaps. Chis et al. [6] discuss the limitations of the dominator relation (mostly due to the issue of detecting shared ownership), alternatively describe a *ContainerOrContained* relation, and use this relation to detect various inefficient memory patterns.

State-of-the-art memory monitoring tools share the typical functionality to represent heap states as type histograms, showing the number of instances per class and their shallow sizes. While this enables users to detect large object groups of a certain type in the same way in all tools, they differ in how they support memory leak root cause detection.

VisualVM [32] is a general performance monitoring tool for Java applications that provides memory analysis based on heap dumps. In addition to a type histogram, it can display a list of all root objects, as well as the dominator tree. From each view, it is possible to inspect individual objects, including functionality to inspect an object's fields, accessing referencing objects, and finding the closest root object. Even though VisualVM can calculate the retained size for the objects of a given type, it is not possible to change that classification or to further split that group. Neither can the user select multiple objects that might have shared ownership for inspection, a functionality that is available in AntTracks by using its object group inspection window on arbitrary object groups. The

Netbeans profiler [31] is a slimmed down version of VisualVM and is integrated into the Netbeans IDE.

The focus of the the *Eclipse Memory Analyzer (MAT)* [15] tool is to provide a fast overview on possible memory leaks, reducing the need for complex user interaction. By default, it displays overview charts of the largest dominating objects, packages and class loaders. Their computations heavily rely on the dominator tree. In addition, MAT provides an automatic leak suspect analysis which detects and extracts the most suspicious objects from the dominator tree. While MAT provides easy-to-use automated analysis features and high-level abstractions based on the dominator tree, it shares a common problem with VisualVM: Memory leak root cause detection is not supported for leaking object groups with shared ownership.

7 FUTURE WORK AND THREATS TO VALIDITY

Heap object graphs, alongside closure, could also be reconstructed and calculated using normal heap dumps. Yet, the analysis of a single heap state may not be sufficient to detect and analyze the proliferation of objects. Thus, AntTracks utilizes a trace-based method to continuously record memory information. Trace files can then be used to analyze an application's memory behavior *over time* on *object-level*. Future work will make use of the temporal information that can be reconstructed from traces and will combine it with the approaches presented in this work. This includes automated analysis of changes to the heap object graph, i.e., how object references and GC root pointers change over time. This will enable us to automatically detecting continuously growing object groups, e.g., detecting growing object groups of certain types or allocation sites. It will also be possible to detect objects with a growing retained size, which might hint at growing data structures. We further plan to include a constraint DSL in AntTracks which would enable users to define memory constraints such as maximum retained sizes or checks for invalid reference patterns. Such constraints can then be checked during parsing of trace files.

After detecting growing object groups (e.g., due to a memory leak) or object groups that keep large portions of the heap alive, the user's goal is to make these objects eligible for garbage collection. This can be done by cutting references between objects on the paths to their GC roots. Analyzing these paths to find suitable cutting points is most effectively done by visualizing the object graph. We therefore plan to develop more sophisticated graph visualization techniques that offer convenient navigation and analysis of object graphs. Yet, without aggregation, such object graphs tend to grow big and become infeasible to analyze. Thus, our goal is to reduce this complexity by collapsing the object graph based on data structure membership, i.e., by aggregating objects that belong to a certain data structure into a single graph node. When searching for suitable cutting points, one is not interested in the internals of data structures. For example, a linked list maintains its elements via `Node` objects. These internals should not show up in the object graph. By aggregating all objects of the list into a single node, one could raise the abstraction level of the analysis. As a first step, data structure aggregation could be done for the well-known Java collection types. Later it could also be done for arbitrary user-defined data structures described by a domain-specific language.

The major threat to validity of our work is its currently restricted evaluation based on an artificial use case. It also lacks an evaluation on how often multi-object ownership occurs in real-world applications. There exist studies on the memory behavior of real-world applications [8, 9, 16], yet they do not evaluate the interaction between objects or data structures, e.g. they do not report multi-object ownership rates. We plan to conduct in-depth evaluations on open-source projects in the future to gain a deeper understanding of object interaction in real-world applications. Further, to prove AntTracks's applicability for finding the root cause of memory leaks in such real-world applications, we also plan to conduct a user study with our industry partner. In addition to comparing AntTracks to existing tools, e.g., in terms of found memory leaks, we also want to evaluate which classifier combinations are most useful when searching for memory leaks.

8 CONCLUSION

In this paper, we presented new techniques for collecting information about GC roots and how to use this information for computing the transitive closure and the GC closure of the object graphs referenced by these roots. A distinguishing feature of our approach is the fact that we can compute the GC size for whole object *groups* and not only for single objects, as is the case in dominator-tree-based approaches. From the closures we derived metrics such as the *retained size* of an object group (i.e., the amount of memory that is kept alive by this group). Finally, we integrated our techniques into a state-of-the-art memory monitoring tool (AntTracks) that provides classification and navigation facilities for analyzing the memory behavior of an application and finding the root causes of memory leaks.

The GC root information is written to a trace file at the start of every garbage collection and can thus be reconstructed offline for any garbage collection point. In a quantitative evaluation based on the DaCapo, DaCapoScala and SPECjvm2008 benchmark suites, we showed that tracing GC root information introduces 1.00% overhead on the application's run time and 0.21% overhead on the generated trace file size on average, which is low enough to be used in production systems.

A functional evaluation showed that our approach for computing the GC closure enables us to detect memory leaks even if the leaking objects are shared by multiple owner objects. In particular, the retained size metric proved useful to detect data structures that have shared ownership. Finally, we showed how bottom-up analysis can be used to find the GC roots that keep a set of leaking objects alive.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS '10)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/1879211.1879222

[2] Stephen Alstrup and Peter W. Lauridsen. 1996. *A Simple Dynamic Algorithm for Maintaining a Dominator Tree.* Technical Report.

[3] Earl T Barr, Christian Bird, and Mark Marron. 2013. Collecting a heap of shapes. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 123–133.

[4] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proc. of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. 76–89.

[5] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, Vol. 41. ACM, 169–190.

[6] Adriana E Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy. 2011. Patterns of memory inefficiency. In *European Conference on Object-Oriented Programming*. Springer, 383–407.

[7] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2001. A simple, fast dominance algorithm. *Software Practice & Experience* 4, 1-10 (2001), 1–8.

[8] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. [n. d.]. Empirical Study of Usage and Performance of Java Collections. ([n. d.]). https://doi.org/10.1145/3030207.3030221

[9] Diego Costa and Rivalino Matias Jr. 2015. Characterization of Dynamic Memory Allocations in Real-World Applications: An Experimental Study. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 93–101. https://doi.org/10.1109/MASCOTS.2015.28

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

[11] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. 2002. Visualizing the execution of Java programs. In *Software Visualization*. Springer, 151–162.

[12] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *ECOOP' 99 — Object-Oriented Programming*, Rachid Guerraoui (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 116–134.

[13] J. Eve and R Kurki-Suonio. 1977. On computing the transitive closure of a relation. *Acta Informatica* 8, 4 (oct 1977), 303–314. https://doi.org/10.1007/BF00271339

[14] R. Falke, R. Klein, R. Koschke, and J. Quante. 2005. The Dominance Tree in Visualizing Software Dependencies. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 1–6. https://doi.org/10.1109/VISSOF.2005.1684311

[15] Eclipse Foundation. 2018. Eclipse Memory Analyzer (MAT) (last accessed May 11, 2018). https://www.eclipse.org/mat/. (2018).

[16] Mohammadreza Ghanavati, Diego Costa, Artur Andrzejak, and Janos Seboek. 2018. Memory and Resource Leak Defects in Java Projects: An Empirical Study. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. ACM, New York, NY, USA, 410–411. https://doi.org/10.1145/3183440.3195032

[17] T. Hill, J. Noble, and J. Potter. 2000. Scalable visualisations with ownership trees. In *Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS-Pacific 2000*. 202–213. https://doi.org/10.1109/TOOLS.2000.891370

[18] Trent Hill, James Noble, and John Potter. 2002. Scalable visualizations of object-oriented systems with ownership trees. *Journal of Visual Languages & Computing* 13, 3 (2002), 319–339.

[19] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 31–38. https://doi.org/10.1145/1190216.1190224

[20] Donald E. Knuth. 1971. Top-down syntax analysis. *Acta Informatica* 1, 2 (01 Jun 1971), 79–110. https://doi.org/10.1007/BF00289517

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

[22] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proc. of the 6th ACM/SPEC Int'l. Conference on Performance Engineering (ICPE '15)*. 51–62.

[23] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2016. Efficient and Viable Handling of Large Object Traces. In *Proc. of the 7th ACM/SPEC on Int'l. Conference on Performance Engineering (ICPE '16)*. 249–260.

[24] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/3030207.3030211

[25] Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (Jan. 1979), 121–141. https://doi.org/10.1145/357062.357071

[26] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich. 2013. Abstracting runtime heaps for program understanding. *IEEE Transactions on Software Engineering* 39, 6 (June 2013), 774–786. https://doi.org/10.1109/TSE.2012.69

[27] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. Springer-Verlag, Berlin, Heidelberg, 74–98. https://doi.org/10.1007/11785477_5

[28] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2009. Making Sense of Large Heaps. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming (Genoa)*. Springer-Verlag, Berlin, Heidelberg, 77–97. https://doi.org/10.1007/978-3-642-03013-0_5

[29] Nick Mitchell and Gary Sevitsky. 2007. The Causes of Bloat, the Limits of Health. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 245–260. https://doi.org/10.1145/1297027.1297046

[30] Oracle. 2018. The HotSpot Group (last accessed May 11, 2018). http://openjdk.java.net/groups/hotspot/. (2018).

[31] Oracle. 2018. Netbeans Profiler (last accessed May 11, 2018). https://profiler.netbeans.org/. (2018).

[32] Oracle. 2018. VisualVM: All-in-One Java Troubleshooting Tool (last accessed May 11, 2018). https://visualvm.github.io/. (2018).

[33] G. Ramalingam and Thomas Reps. 1994. An Incremental Algorithm for Maintaining the Dominator Tree of a Reducible Flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 287–296. https://doi.org/10.1145/174675.177905

[34] Derek Rayside and Lucy Mendel. 2007. Object Ownership Profiling: A Technique for Finding and Fixing Memory Leaks. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 194–203. https://doi.org/10.1145/1321631.1321661

[35] Derek Rayside, Lucy Mendel, and Daniel Jackson. 2006. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis (WODA '06)*. ACM, New York, NY, USA, 57–64. https://doi.org/10.1145/1138912.1138924

[36] Steven P Reiss. 2009. Visualizing the Java heap - Demonstration Proposal. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 389–390.

[37] S. P. Reiss. 2009. Visualizing the Java heap to detect memory problems. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 73–80. https://doi.org/10.1109/VISSOF.2009.5336418

[38] Steven P. Reiss. 2010. Visualizing the Java Heap. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*. ACM, New York, NY, USA, 251–254. https://doi.org/10.1145/1810295.1810344

[39] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2013. Elephant Tracks: Portable Production of Complete and Precise Gc Traces. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 109–118. https://doi.org/10.1145/2464157.2466484

[40] C. Ruggieri and T. P. Murtagh. 1988. Lifetime Analysis of Dynamically Allocated Objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, NY, USA, 285–293. https://doi.org/10.1145/73560.73585

[41] Robert Tarjan. 1971. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. IEEE, 114–121. https://doi.org/10.1109/SWAT.1971.10

[42] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proc. of the 8th ACM/SPEC on Int'l. Conference on Performance Engineering (ICPE '17)*. 357–360.

[43] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018) (ICPE 2018)*.