JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Memory leak detection and diagnosis in .Net and Java applications using run-time memory dumps

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

Informatik

Eingereicht von:
Stefan Riha

Angefertigt am:
Institute for Formal Models and Verification

Beurteilung:
Univ.-Prof. Dr. Armin Biere

Linz, 11, 2009

Abstract

Programming errors that result in memory errors are hard to detect. Even today, with the availability of cheap memory modules and virtual machines supporting garbage collection, detecting such errors can be crucial because using more memory in computer systems or garbage collection in virtual machines only delay the occurrence of the problem.

When programming languages did not support automated memory management, a memory leak was a piece of memory allocated on the heap, that is not referenced anymore. This piece of memory will only be released when the program terminates. For programming languages with a managed heap, this definition needs to be updated: a memory leak is a piece of memory which is allocated and referenced on the managed heap but the program will never read or write the memory location on any path through the program. Memory errors on the managed heap are a subset of memory errors that occur on unmanaged heaps. They are very hard to detect and only a few tools support techniques to analyze problems on the managed heap.

This work presents the approach taken in dynaTrace, a monitoring and analysis tool for distributed heterogeneous JAVA and .NET application. dynaTrace helps developers detect memory errors, especially memory leaks, by creating, storing and analyzing different types of heap dumps. dynaTrace is designed to be used in both development and production environment, hence it must meet strict requirements towards time and memory overhead for the application under diagnosis. Usually, producing heap dumps generates heavy load for the virtual machine, therefore, to comply with these restrictions, different types of heap dumps are supported by dynaTrace: simple heap dumps, producing minor time and memory overhead, and extended heap dumps that stall the virtual machine for a longer time. This work presents the architecture of dynaTrace, discussing the advantages of the agent-server design and how overhead for the virtual machine is reduced to a minimum.

Heap dumps are organized as a graph data structure. To make memory analysis easier for the developer and to reduce the complexity of the heap dumps, several analysis algorithms were developed. These include the garbage collector size and the dynamic size

algorithms. To compute the gc size for each object the node which prevents the object from being garbage collected (the dominator node) must be computed. The dynamic size of an object can be calculated by summing up the sizes of all nodes reachable from the object. Additional algorithms that improve the runtime of the original dynamic size algorithm (strongly connected components, biconnected components, articulation points and dominating articulation points) are also presented. The strongly connected components algorithm groups nodes that are known to have the same dynamic size. By using the biconnected components and articulation points algorithms the graph can be partitioned into two or more independent sub graphs. The original articulation point dynamic size algorithm has a design error: It is possible to count certain nodes multiple times. To overcome this flaw, the notion of dominating articulation points and an algorithm to compute those was developed and is presented in this work.

An important conclusion can be drawn from the data generated to compare the different heap dump and analysis algorithms: The algorithms are tested using generated heap dumps and heap dumps from two sample applications. Apart from the running times, also statistic values were determined for the heap dump or strongly connected components graphs. One important insight gained from the data is that the use of dominating articulation points speeds up the computation of the dynamic size.

Kurzfassung

Programmierfehler die Speicherfehler nach sich ziehen sind schwer zu entdecken. Obwohl heute viele Programmiersprachen sogenannte Garbage Collectoren unterstützen und trotz der Verfügbarkeit von billigen Speichermodulen ist es essentiell, diese Speicherfehler zu identifizieren, denn sowohl Garbage Collectoren, als auch zusätzlicher Speicher, verzögern nur den Zeitpunkt an dem sich die Auswirkungen eines Speicherfehler bemerkbar machen.

Als Programmiersprachen noch keine Methoden zur automatischen Speicherverwaltung einsetzten, wurde ein Speicherbereich der nicht mehr referenziert wird als Speicherleck angesehen. Diese Definition muss für Programmiersprachen die Virtuelle Maschinen und Garbage Collectoren, also einen gemanagten Heap verwenden, angepasst werden: ein Speicherleck ist ein Speicherbereich der noch referenziert wird, aber auf den das Programm in keinem möglichen Pfad durch das Programm mehr zugreift. Speicherfehler auf dem verwalteten Heap sind eine Teilmenge der Speicherfehler die bei einem unverwaltetem Heap auftreten können. Sie sind sehr schwer zu entdecken und es gibt nur wenige Software Werkzeuge die Techniken zur Speicheranalyse anbieten.

Diese Arbeit stellt den Ansatz des Tools dynaTrace vor, ein monitoring und Analyse Tool für verteilte heterogene JAVA und .NET Anwendungen. dynaTrace hilft Entwicklern Speicherfehler, vor allem Speicherlecks, zu finden. Dazu können verschiedene Arten von Heap Dumps erstellt, gespeichert und analysiert werden. dynaTrace soll sowohl in Entwicklungs- als auch in Produktionsumgebungen laufen, daher muss das Tool strenge Regeln bezüglich zusätzlichem Zeit- und Speicherkonsum einhalten. Normalerweise verursacht das Erstellen eines Heap Dumps große Belastung für die Virtuelle Maschine, daher unterstützt dynaTrace das erstellen von verschiedene Typen von Heap Dumps, etwa einfache Heap Dumps, die die zusätzlich benötigte Zeit und Rechenleistung, im Gegensatz zu den erweiterten Heap Dumps, auf ein Minimum reduzieren.

Heap Dumps sind als Graphen organisiert. Um die Speicheranalyse für den Entwickler zu vereinfachen und um die Komplexität der Heap Dumps zu reduzieren, wurden mehrere Analyse Algorithmen entwickelt. Diese Algorithmen umfassen den Garbage-Collector-Größe- und den Dynamische-Größe-Algorithmus. Um die Garbage-Collector-

Größe berechnen zu können muss für jedes Objekt der Knoten berechnet werden, der verhindert, dass das Objekt vom Garbage Collector frei gegeben wird (der dominator Knoten). Die dynamische Größe wird berechnet, indem man die Größen aller Knoten zusammenzählt, die von einen Objekt erreicht werden können. Einige Algorithmen zur Verbesserung der Laufzeit des Garbage-Collector-Größe-Algorithmus (strongly connected components, biconnected components, Artikulationspunkte und dominierende Artikulationspunkte) werden in der Arbeit vorgestellt. Mit Hilfe des strongly connected components Algorithmus werden Knoten, die die gleiche dynamische Größe haben, zu einem Knoten zusammengefasst. Der biconnected components und Artikulationspunkte-Algorithmus teilt den Graphen in mehrere unabhängige Teile. Der ursprünglich in dieser Arbeit entwickelte Artikulationspunkte-Algorithmus hat einen Fehler: Es ist möglich die Größe von bestimmten Knoten mehrmals zu zählen. Um das Problem zu lösen, wurde in dieser Arbeit die Definition so genannter dominierender Artikulationspunkte und ein Algorithmus zur Berechnung dieser entwickelt.

Ein Schluss kann im Zuge dieser Arbeit aus den Testdaten der Algorithmen gezogen werden: Die Algorithmen werden mit selbst erzeugten Heap Dumps und Dumps von zwei Beispielanwendungen getestet. Abgesehen von den Laufzeiten wurden auch noch einige statistische Werte für die unterschiedlichen Heap Dump und strongly connected components Graphen ermittelt. Die Ergebnisse der Tests zeigen, dass die Verwendung von dominierenden Artikulationspunkten die Berechnung der dynamischen Größe beschleunigt.

Contents

1 Introduction **1**
1.1 Objective . . . 2
1.2 Preliminaries . . . 4
1.3 Control Flow Graph vs. Reference Graph . . . 5
1.4 Control Flow Graph vs. Reference Graph . . . 7
1.5 The Memory Diagnosis Process and Dump Types . . . 8

2 Dynamic vs. GC Size **10**

3 Heapdumps **13**
3.1 Requirements of the Heap Dumps . . . 13
3.2 Heap Dump Data . . . 14
3.3 Heap Dump Algorithms . . . 15

4 Algorithms **18**
4.1 Dominator Tree . . . 18
4.2 Garbage Collector Size . . . 22
4.3 Dynamic Size Simple . . . 23
4.4 Strongly Connected Components . . . 24
4.5 BFS based Dynamic Size Algorithm . . . 28
4.6 Articulation Points . . . 30
4.7 Delta Debugger . . . 33
4.8 Dominating Articulation Points . . . 33

5 Results **37**
5.1 Generated Heap Dumps . . . 38
5.2 SimpleLoadGenerator Heap Dumps . . . 43
5.3 dynaTrace Server Heap Dumps . . . 44
5.3.1 Differences between Dynamic and Garbage Collector Size . . . 47

6 Conclusion **48**

7 Future Work **50**

A Tables **52**

B Heapdumps Implementation **60**
B.1 JVMPI Heap Dump 60
B.2 JVMTI Heap Dump 64
B.3 .Net Profiling API Heap Dump 67

C Implementation **70**
C.1 Heap Dump Algorithms 70
C.2 Reference Graph Analysis Algorithms 73
C.2.1 Heap Dump Generators 74
C.2.2 Delta Debugger 75
C.2.3 Analysis Algorithms 76

Bibliography **84**

Chapter 1

Introduction

When memory in computer systems was sparse, programmers were required to monitor memory consumption carefully. Therefore memory analysis was very important. Increasing memory of a computer system or using an automatic garbage collection for the runtime environment of a programming language cannot solve memory problems. The effects of the problem are simply delayed. But the availability of inexpensive memory modules and virtual machines using garbage collection, seems to have reduced the awareness towards problems resulting from memory leaks. As a result, computer supported memory analysis techniques are an important topic today and will become even more important in the future.

Java as well as .Net virtual machines use three locations to allocate memory:

- **Stack:** The stack is used to store local variables and parameters of methods. A separate stack exists for each thread in the program. Memory problems caused by threads occur if a thread deadlocks or if the method executed loops forever and does not terminate. The result in both situations is that memory used by the thread is lost and can only be deallocated when the thread is terminated.
- **managed Heap:** The part of memory where the garbage collector (GC) collects unused memory. More about GC technologies can be found in [9] and [14]. The heap is used to store object data that is allocated at runtime, that is, during execution of the program.
- **unmanaged Heap:** The unmanaged heap is used by the virtual machine itself to store internal data structures. It can be accessed by the programmer via the Java Native Interface (JNI) in Java or using unmanaged code in .Net. Unmanaged Heap is not considered by the garbage collector.

This work focuses on the managed heap because in Java or .Net programs most memory issues occur there. For programming languages without a managed heap, a memory leak is a piece of memory allocated on the heap, that is not referenced anymore. This piece of memory will only be released when the program terminates. For languages with memory management the same definition of memory leaks can be used for the unmanaged heap, but it needs to be adjusted for the managed heap: a memory leak is a piece of memory which is allocated and referenced on the managed heap but the program will never read or write the memory location on any path through the program. Leaking memory that suffices the definition of a memory leak for the unmanaged heap will be deallocated by the garbage collector but it cannot handle memory defined as leaking on the managed heap.

The most common causes for memory leaks in managed code are: [15] [4]

- Objects forgotten in collections.
- The amount and time of data objects stored in HTTP sessions.
- Missed objects in self implemented lists, data structures or buffers.
- Objects assigned to static fields of classes are not cleaned up until the application shuts down or they are set to null. A special problem are statically held collections.

Various tools that help the programmer detect those memory problems are available, but most of them have drawbacks. Their disadvantage is that they need a lot of CPU time and memory. Only some of these tools support analysis algorithms for heap dumps.

1.1 Objective

The thesis has two main goals: The first is the development of a tool that creates heap snapshots to find memory leaks in .Net and Java applications. Memory information about the application is collected and forwarded to a central server for analysis and correlation. Memory information includes:

- **Classes:** store the classname, classID, size of all objects of a class and number of instances of a class.
- **Objects:** collect classID, objectID and size for all objects.
- **References:** find out referees and referrers for all objects of the heap.

The tool allows to create memory dumps with object references. Furthermore, low-memory programming techniques are applied in the implementation to handle low and out-of-memory situations in the virtual machine. The information collected helps to trace object references and to get an overall picture of the memory consumption of the application.

The second goal is to develop and implement algorithms to analyze the heap data collected. During analysis, the heap is represented as object reference graph. The objects and classes are represented by *nodes* and references are edges between nodes.

Most profilers and debuggers allow to create dumps of the heap, but without further sophisticated analysis techniques it is rather difficult to detect memory leaks. The following analysis techniques can be applied on heap dumps to allow efficient memory analysis:

- **Root Nodes Detection:** Finding the root nodes of a given object in the reference graph.
- **Garbage Collector Size Calculation:** Calculate the garbage collector size of a given object: the amount of memory deallocated when the object is garbage collected.
- **Dynamic Size Calculation:** Calculate the dynamic size of a given object or subgraph: the size of the object and the size of all other objects reachable from it. The dynamic and the gc size of an object can be used to identify objects that are possibly leaking memory.
- **Important Nodes Detection:** Important nodes are nodes that are referred by or refer to many other nodes. Usually nodes at those the graph can be divided into two independent parts (articulation points) are important nodes in the reference graph.
- **Memory Leak Detection:** Finding potentially leaking regions by using the dynamic and the gc size.
- **Diffing:** To detect potential memory leaks it is important to compare memory dumps. Diffing is not discussed in this work.

The results have been integrated into dynaTrace, a performance analysis tool for distributed applications, developed by dynaTrace Software GmbH [1].

1.2 Preliminaries

dynaTrace is a diagnostic tool for distributed heterogeneous J2SE/J2EE and .Net applications which detects performance and stability problems under production load levels. dynaTrace is designed to be used both in development environments and production systems. It has a three tiered architecture: the dynaTrace Agent(1), the dynaTrace Client(2), the dynaTrace Server(3) and an optional Repository(4).

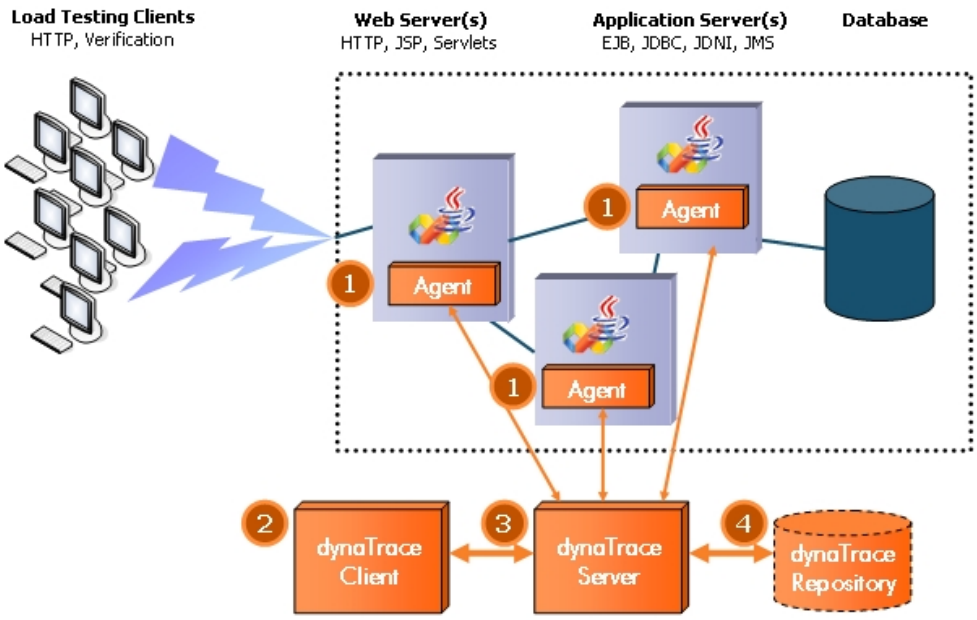

Figure 1.1: Architecture of dynaTrace [1]

- **dynaTrace Agent:** The dynaTrace agent is running within each process of the application under diagnosis. It is responsible for the transmission of collected information to the central server. In addition, it transmits memory snapshots to the server for analysis. The Diagnostics Agent and the dynaTrace Server communicate using three communication channels. For memory diagnosis, a fourth channel is opened, used only to transmit heap dumps from the agent to the server.
- **dynaTrace Server:** The dynaTrace server stores and analyses collected memory data. All analysis is performed on the server, sustaining overhead for the virtual machine at a very low level. To use data for memory diagnosis the server stores, manages and analyzes the collected heap dumps on the file system.
- **dynaTrace Client:** The dynaTrace Client requests the analysis results from the central server, displays it and allows the user to interact with it. The dynaTrace Client is implemented as an SWT application based on the eclipse platform.

Heap dumps are stored on the hard disk of the computer system running the server. Each dump is stored in its own directory.

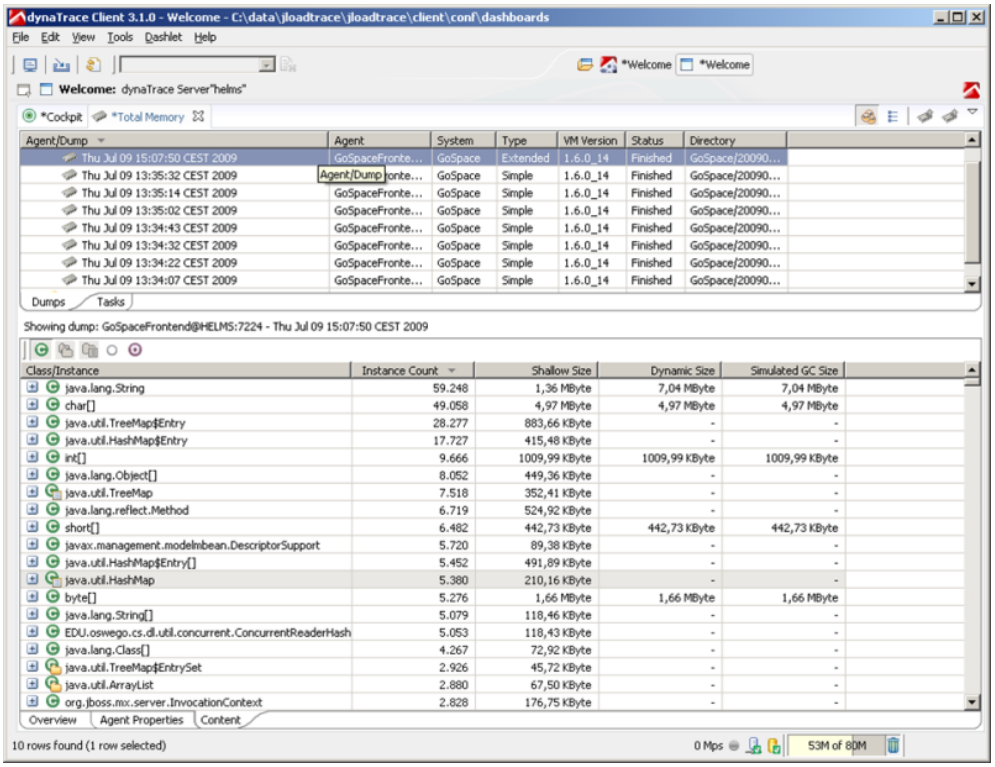

Figure 1.2: Heap Dump View of dynaTrace [1]

1.3 Control Flow Graph vs. Reference Graph

The most common analysis techniques are the control flow graph and reference graph memory analysis.

Control Flow Graph Memory Analysis

To perform control flow based memory analysis the execution paths trough the program must be recorded. For object oriented programming languages the paths contain all method calls and all constructor calls and destructor calls of the objects. For each constructor call, the size of the allocated object (static size) is recorded. For further analysis, all control paths are merged to one control flow graph, by merging all nodes that are results of calls to the same method or constructor. The size of the objects is propagated upwards to the root of the control flow graph. The size of each node

is the sum of the sizes of all children. The edges of the graph are weighted using the size information in bytes or percentage. The resulting control flow graph reveals paths through the program that require more memory than others and are worth detailed examination.

The Google perftools [2] are an implementation of the control flow graph based memory analysis algorithm for the programming language C. The advantage of this method is that it easily finds methods which are responsible for the largest memory consumption. Furthermore it is easy to calculate the size of the nodes and the propagated size. The drawback of this method is that collecting data, needed to construct the control flow graph, is very expensive. Each call of a method or a constructor must be monitored, resulting in unwanted runtime overhead. For the programming language C it is easy to monitor the allocation and the deallocation of each memory block. Only the alloc and free method must be adapted.

Reference Graph Memory Analysis

The reference graph based memory analysis uses heap dumps. In contrast to the control flow graph technique the reference graph method produces less overhead because CPU time and memory is only consumed when producing a heap dump. The drawback is that no control flow information is available. Without control flow information, new algorithms to analyze the heap dump and to calculate size information are necessary. The dynamic size and the garbage collector size are the most common size information calculated for nodes in the heap graph. The reference graph based memory analysis was implemented by hprof [13] and MAT [6].

The control flow based memory analysis has advantages if it is possible to identify the method which caused the memory allocation to an object or array. If it is impossible to identify those methods, the reference graph based memory analysis yields better results. Consider a situation where a factory method is used to construct objects with the exact characteristics of the objects specified later. For example a compiler using a factory method to allocate objects that are used to store information about identifier and keywords. They are stored in two different lists, depending on whether they represent keywords or identifiers, defined outside of the factory method. A memory leak in one of the lists can be found w

1.4 Control Flow Graph vs. Reference Graph

The most common analysis techniques are the control flow graph and reference graph memory analysis methods.

Control Flow Graph Memory Analysis

To perform control flow based memory analysis the execution paths trough the program must be recorded. For object oriented programming languages the paths contains all method calls and all constructor calls and destructor calls of the objects. For each constructor call, the size of the allocated object (static size) is recognized. For further analysis, all control paths are merged to one control flow graph, by merging all nodes that are results of calls to the same method or constructor. The size of the objects is propagated upward to the root of the control flow graph. The size of each node is the sum of the sizes of all children. The edges of the graph are weighted using the size information in bytes or percentage. The resulting control flow graph reveals paths through the program require more memory than others and are worth for further investigation.

The Google perftools [2] are an implementation of the control flow graph based memory analysis algorithm for the programming language C. The advantage of this method is that it easily finds methods which are responsible for the largest memory consumption. Furthermore it is easy to calculate the size of the nodes and the propagated size. The drawback of this method is that the collection of data to construct the control flow graph is very expensive. Each call of an method or constructor must be monitored, resulting in unwanted runtime overhead. For the programming language C it is easy to monitor the allocation and the deallocation of each memory block. Only the alloc and free method must be adapted.

Reference Graph Memory Analysis

The reference graph based memory analysis uses heap dumps. In contrast to the control flow graph technique the reference graph method produces less overhead because CPU time and memory is only consumed when producing a heap dump. The drawback is that no control flow information is available. Without control flow information, new algorithms to analyze the heap dump and to calculate size information are necessary. The dynamic size and the garbage collector size are the most common size information

calculated for nodes in the heap graph. The reference graph based memory analysis was implemented by hprof [13] and MAT [6].

The control flow based memory analysis has its advantages if it is possible to identify the method which caused the memory allocation to the object or array. If this is impossible the reference graph based memory analysis is better. Consider a situation where a factory method is used to construct objects with the exact characteristics specified later. For example a compiler using a factory method to allocate objects that are used to store information about identifier and keywords. They are stored in two different lists, depending on whether they represent keywords or identifiers what is defined outside of the factory method. A memory leak in one of the lists can be found by applying the reference graph based memory analysis. It is impossible to detect the leaking list with the control flow graph method, for it cannot differ between the size of keywords and the size of identifiers in the factory method. The only conclusion that can be drawn from the control flow graph method is, that the memory consumption of the factory method is unusually high.

Combined Memory Analysis

The two methods can be combined. The basis for the combined method is the reference graph based memory analysis. The heap dump is required to contain information about root objects. For those that are referenced from the stack, stack traces can be collected. Using stack traces, the control flow graph can be created similar to the control flow based memory analysis. It is not investigated which size information, used as edge weights, yields good results. Possible size information are the dynamic size or the garbage collector size of the root node. Details about this method can be found in Chap. 7.

1.5 The Memory Diagnosis Process and Dump Types

Memory diagnosis is a complex and time consuming task. Analyzing heap dumps allows to monitor application behavior over time and to find trends before errors occur. Two heap dumping techniques are implemented: the first method collects class names and the number of bytes needed by all objects of that class, as well as the number of instances for each class. This technique is called *simple heap dump*. The second method collects all classes, objects and references between them. It is called *extended heap dump*.

To identify memory leaks using simple and extended heap dumping techniques, the first step is to create simple heap dumps. They can be compared and entries can be sorted to find the class with the largest number of bytes needed by all objects of the class. After collecting a few dumps, it is possible to identify trends in memory consumption for specific classes. If the size needed by all objects of a class is growing each dump, this class is a candidate for a memory leak.

The second step is to create extended heap dumps to identify the root cause of the memory leak. They must be used cautiously because an extended memory dump stalls the virtual machine significantly longer than a simple memory dump. Using the objects of the classes identified as potential memory leaks as starting points, the root cause can be found by following the references to objects of known classes or by using algorithms described in Chap. 4 (Root Nodes Detection, Garbage Size Calculation, Dynamic Size Calculation and some other). If no trend in memory consumption emerges or if the classes are to common like `java.lang.String`, another way to search for memory leaks is to calculate the garbage collector size or dynamic size for all objects and analyze the biggest objects and arrays. If memory consumption is considered too high these sizes can be drawn into account when profiling the applications.

Chapter 2

Dynamic vs. GC Size

The size of an object or array is important for almost all strategies to find memory leaks. The size of an object can be defined in several ways. In this work, three different sizes are defined:

- **Static-Size:** The static size of an object is the size of the piece of memory necessary to store one instance of the class of this object. The static size is the sum of the sizes needed to store all primitive fields, all references and some overhead to manage the object with the garbage collector. Some virtual machines align the static size on a multiple of four byte.
- **GC-Size:** The garbage collection size of an object is the amount of memory deallocated if the object is garbage collected.
- **Dynamic-Size:** The dynamic size of an object is the size of the object and the size of all other objects reachable from it.

The static size is the smallest of the three sizes and the dynamic size is always greater or equal to the garbage collector size.

The garbage collector size is used by the Eclipse Memory Analyzer (MAT) to find big chunks (potential memory leaks) in the heap graph. In MAT, the garbage collector size is called *retained size* and the set of all objects included in the dynamic size of an other object is called the *retained set*. [6]

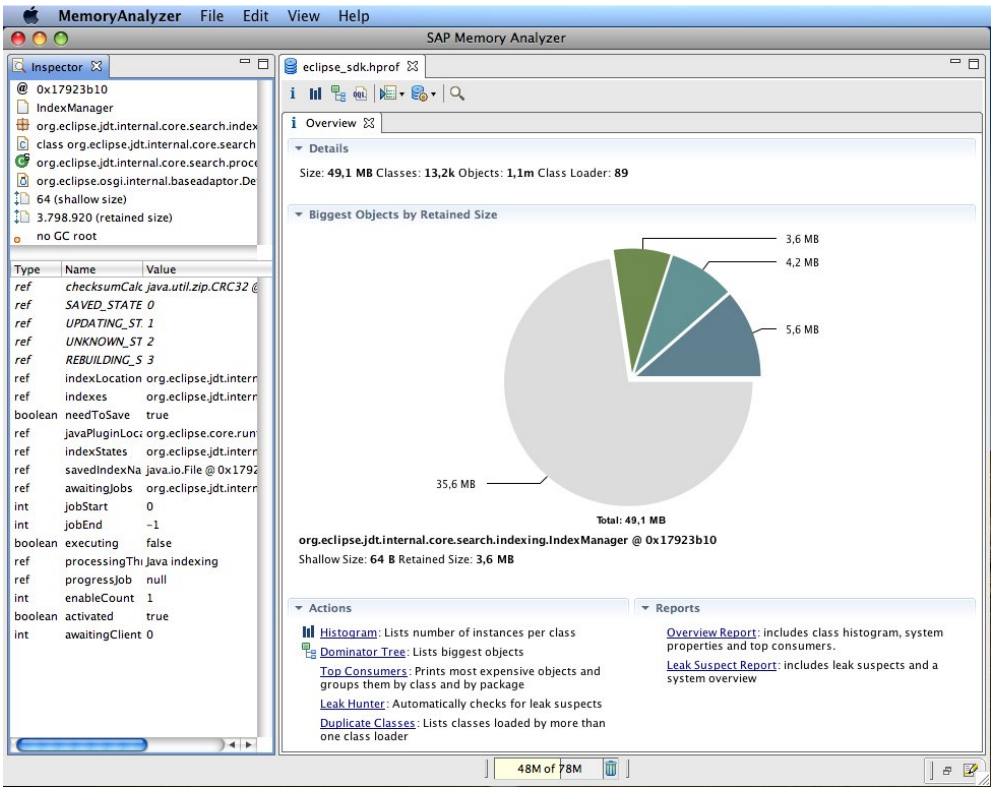

Figure 2.1: Eclipse Memory Analyzer [6]

The assumption in this work is that using the dynamic size, rather than the garbage collector size, to find big chunks in the heap graph yields better results because the garbage collector size does not consider shared objects.

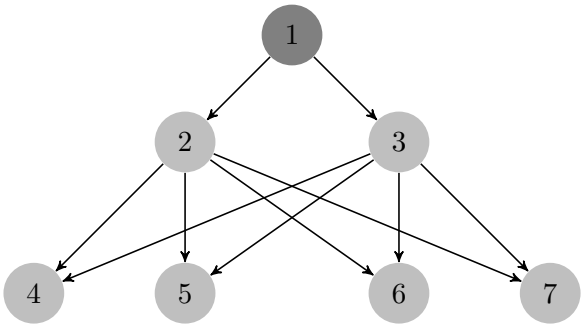

Figure 2.2: Heap Dump Example Graph

The nodes 4, 5, 6 and 7 share references from the nodes 2 and 3. All nodes are dominated by node 1. Knowing the dynamic size for nodes helps to find the biggest nodes. Knowing only the garbage collector size, nodes that contain many shared nodes appear less important.

A real world example for such a situation is a system which uses a buffer or a cache to store user data received and an analysis algorithm which shallow copies the data in the buffer and stores it in a separate list.

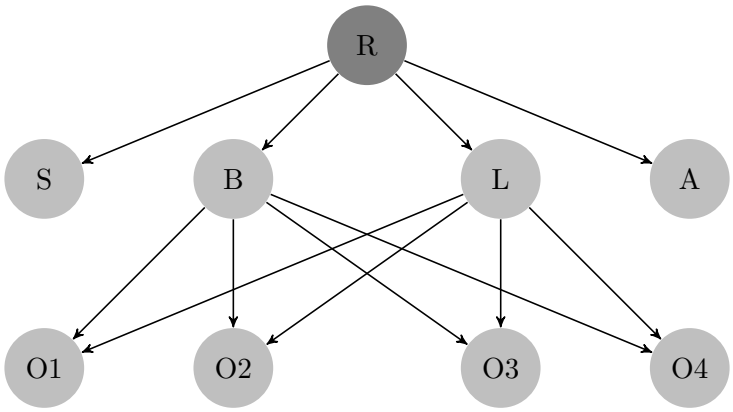

Figure 2.3: Real World Heap Dump Example Graph

- **R:** The root node of the heap dump graph. The parent node for all garbage collector roots on the heap.
- **S:** This node represents the server which allocates data objects and adds them to the buffer.
- **B:** The buffer to store data objects.
- **A:** The object implementing the algorithm to analyse the data objects in the buffer. The algorithm reads the objects from the buffer, performs some operations and stores them in the list.
- **L:** This list stores the data objects for the analysis algorithm.
- **On:** The data objects. Only four of them are displayed in the example graph 2.3. In heap dumps of real world applications many more of them exists.

When considering only the garbage collector size, the size of the node representing the buffer and the node representing the list will be small. If the dynamic size is used the two nodes will be the biggest nodes, right after the global root node R.

Chapter 3

Heapdumps

Heap dumps are the basis for memory analysis considered in this thesis. Heap dumps are used to store the structure of the heap and to make it available for later analysis. Another benefit of heap dumps is that they can be compared. Heap dumps can be represented as object reference graphs, objects and classes are represented as nodes, references are edges between nodes. To create a heap dump, all classes, objects and arrays, as well as all field, static or array references must be collected. Additional information about the objects, classes or arrays is collected depending on the later use of the heap dumps. For some applications it can be useful to know which nodes are gc-root nodes and of which type they are.

3.1 Requirements of the Heap Dumps

To be able to use heap dumps with dynaTrace in a production environment the following requirements must be met for hep dump algorithms:

- **Low CPU and memory overhead:** The CPU time and memory overhead of the application under observation should be as small as possible. If no memory information is collected no overhead should be produced. Otherwise the CPU time and memory overhead should be smaller than five percent of the original applications runtime and memory consumption.
- **Temporal memory analysis:** The memory information collected should be stored so that they can be compared later on.
- **The virtual machine must not run in debug mode**.

- **Cross platform:** The code for heap dumps is required to be portable and 32 and 64 bit compatible. Furthermore the code has to be executable on the following operating systems and platforms:
  - **Windows** 2000 or higher on x86 and amd64 CPUs
  - **Linux** on x86, amd64 and Itanium CPUs
  - **Solaris** on SPARC and x86 CPUs
  - **AIX** on PowerPC CPU
  - **HP-UX** on Itanium and Parisc CPUs
- **Support for most Java and .Net virtual machines:** As many virtual machines as possible must be supported (SUN, IBM, BEA JRockIt, HP). Java versions 1.4, 5 and 6 and .Net versions 1.1 and 2.0 must be supported.

3.2 Heap Dump Data

These requirements are very restrictive. As few as possible data should be collected to minimize CPU time and memory overhead needed for collecting and storing it. To create an extended heap dump with size information, the following data is collected for each object, class and reference:

- **Class:**
  - **ClassID:** used to identify the class.
  - **Class Name:** only needed to be displayed in the user interface.
- **Object:**
  - **ObjectID:** used to identify the object.
  - **ClassID:** for identifying the class of the object
  - **Instance size:** the static size of the instance of the object
- **Reference:**
  - **FromID:** used to identify the referrer.
  - **ToID:** for identifying the referree.

Reference fromID and toID can be a classID or an objectID. If the referrer and the referree are objects the collected reference is a field or array reference. If the reference is a static reference, the referrer must be a class and the referree an object. In Java all three reference types can also appear in combination with a class as referree. Then the referree must be an object of the type `java.lang.Class`.

To minimize the CPU time and memory consumption overhead, no information about root nodes is transferred. It can be calculated on the server side. If no referree nodes exists for a node, the node is called a root node in the heap dump reference graph. If the garbage collector root node is part of a cycle the node with the lowest ID is selected as the root node in the reference graph.

For the simple heap dump, the following information is collected for each class or array class:

- **Class Name:** The name of the class. Only needed to be displayed in the user interface.
- **Instances:** The number of all instances of the class.
- **Size:** The sum of the static size of all objects of the class.

The low CPU and memory overhead requirement is weakened somewhat for the benefit of a lower network traffic: The information for the classes are stored in a hash map with the class name as key. After the heap dump, data is transfered to the server by iterating over the hash map. The number of loaded classes in the virtual machine is much smaller than the number of objects. If data for each object were to be transfered separately, large network traffic would result. Depending on the number of loaded classes, extra memory is needed to store the information collected. Under normal conditions the memory required additionally is much smaller than the heap that will be dumped.

An advantage of the extended and simple heap dump is that they both uses native memory only. The Java or .Net managed heap will not be changed during the dump process.

3.3 Heap Dump Algorithms

Three different heap dump algorithms are implemented. Two for the Java and one for the .Net virtual machine. The first Java heap dump algorithm is based on the Java Virtual Machine Profiler Interface (JVMPI) and is used for Java 1.4 virtual machines. The second algorithm make use of the Java Virtual Machine Tool Interface (JVMTI) and was implemented to create heap dumps for Java 5 or higher virtual machines. Each algorithm is divided into two parts. One part for the extended and one for the simple dump. The basis for the Java heap dump algorithm was the HPROF profiling tool enclosed in the Java virtual machines. HPROF profiler itself does not meet the requirements for production systems because CPU and memory overhead is very high. The HPROF profiling tool holds all available informations about classes and objects in

memory. A lot of extra CPU time is needed to retrieve information. The algorithm must be reduced to the basics to suffice the requirements. The .Net heap dump algorithm was implemented with the .Net Profiling API.

The algorithms and interfaces used are illustrated in detail in Chap. B. Short descriptions in pseudo code follow:

```
1  jvmpi_extended_heap_dump () {
2    Records r = vm_request_heap_dump ();
3    for (Record rec in r) {
4      send_data (dtServer, retreive_extended_data (rec));
5    }
6  }
```

Listing 3.1: extended jvmpi heap dump in pseudo code

```
1  jvmpi_simple_heap_dump () {
2    Data d[];
3    Records r = vm_request_heap_dump ();
4    for (Record rec in r) {
5      d[rec.id] = retreive_simple_data (rec);
6    }
7    send_data_array (dtServer, d);
8  }
```

Listing 3.2: simple jvmpi heap dump in pseudo code

```
1  jvmti_extended_heap_dump () {
2    Classes c[] = vm_get_all_classes ();
3    for (Class cla in c) {
4      tag_class (cla);
5      send_data (dtServer, retreive_extended_data (cla));
6    }
7    // object_tag_function:
8    // tags objects and transfers data to server
9    vm_iterate_objects (object_tag_function ());
10   vm_iterate_references (send_data_function ());
11   untag_objects_and_classes ();
12 }
```

Listing 3.3: extended jvmti heap dump in pseudo code

```
1  jvmti_simple_heap_dump () {
2    Data d[]
3    Classes c[] = vm_get_all_classes ();
4    for (Class cla in c) {
5      tag_class (cla);
6      d[cla.id] = retreive_simple_data (cla);
7    }
8    // object_tag_function:
9    // tags objects and transfers data to server
10   vm_iterate_objects (update_class_data_for_object ());
11   untag_classes ();
12 }
```

Listing 3.4: simple jvmti heap dump in pseudo code

```
1  net_extended_heap_dump () {
2    // gc calls ObjectReferences for each object
3    vm_garbage_collect ();
4  }
5
6  ObjectReferences ( ... ) {
7    send_object_data (dtServer);
8    if (!class_was_sent ()) {
9      send_class_data (dtServer);
10   }
11 }
```

Listing 3.5: extended .net heap dump in pseudo code

```
1  static Data d[];
2
3  net_simple_heap_dump () {
4    // gc calls ObjectReferences for each object
5    vm_garbage_collect ();
6    send_data (dtServer, d);
7  }
8
9  ObjectReferences ( ... ) {
10   update_class_data_for_object ();
11 }
```

Listing 3.6: simple .net heap dump in pseudo code

Chapter 4

Algorithms

For a complete memory analysis heap dumps are not sufficient. It is hard for human users to find links between the nodes of the heap dump graph only with the reference information and the static size. To reduce the problem analysis algorithms are needed, which are the topic of this chapter.

4.1 Dominator Tree

The first analysis algorithm is the dominator tree algorithm. First some definitions:

Definition 1 (dominator) *A vertex v dominates another vertex $w \neq v$ in graph $G = (V, E, r)$ if every path from root r to w contains v. [5]*

Definition 2 (immediate dominator) *Every vertex of a graph G except root r has a unique immediate dominator $idom(w)$. [5]*

Definition 3 (dominator tree) *The edges $\{(idom(w), w) | w \in V - \{r\}\}$ from a directed tree rooted at r, called the dominator tree of graph G, such that v dominates w if and only if v is a proper ancestor of w in the dominator tree. [5]*

To make the definitions easier to digest, the following example is used. Fig. 4.1 shows the example graph form [5] and Fig. 4.2 shows the corresponding dominator tree. The node R is the root node of the graph. From this and the dominator tree definition follows that it is also the root node of the dominator tree. From the dominator definition and

the Fig. 4.1 follows that the node R is the dominator for all other nodes but it is not the immediate dominator for all nodes.

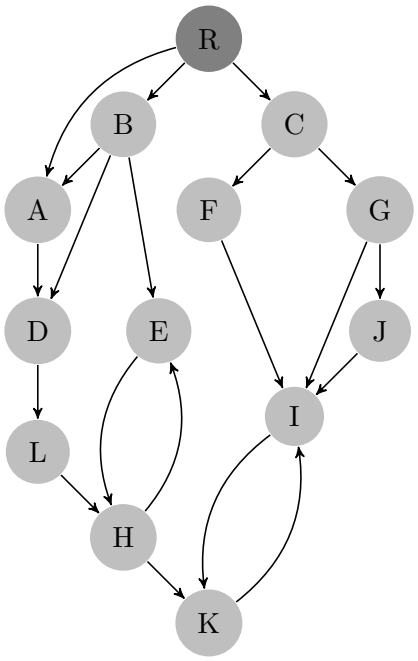

Figure 4.1: Example Graph

For example the node L can be reached by the following paths:

1. $R \rightarrow B \rightarrow A \rightarrow D \rightarrow L$
2. $R \rightarrow A \rightarrow D \rightarrow L$
3. $R \rightarrow B \rightarrow D \rightarrow L$

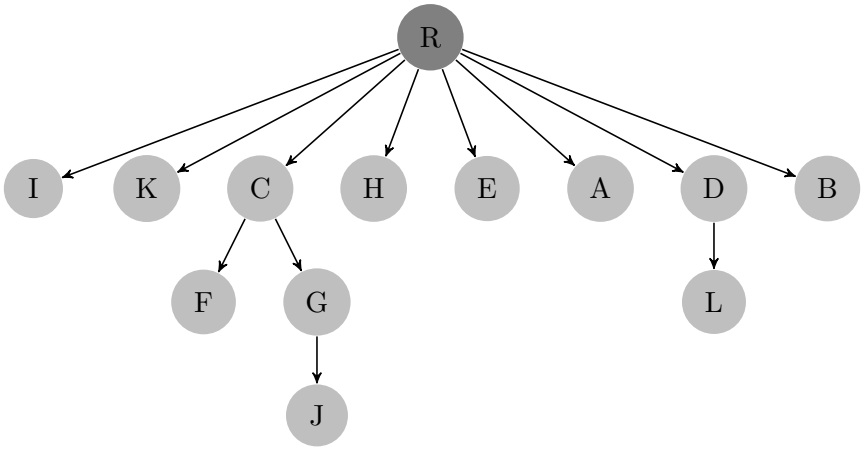

Figure 4.2: Dominator Tree Example

If analyzing the paths using the definition of dominators and immediate dominators, one can determine that the nodes R and D are part of all three paths in the example. This means that these two nodes are dominators of the node L. Node D is further on the right side than node R in all paths. From the definition of immediate dominators it follows that node D is the *immediate dominator* of node L. If the immediate dominator is known for each node the dominator tree can easily be constructed.

The dominator tree algorithm requires that there exists exactly one root node. The heap dump reference graph may violate this property, it may contain more than one root node. To restore the property, a super root node must be created. The super root is the referrer of all garbage collector roots in the heap dump graph.

The dominator tree and the dominator relation have some interesting characteristics as to the garbage collector information:

- The dominator of an object prevents that the object is garbage collected by referencing it.
- If an object is garbage collected all direct or indirected dominated objects are garbage collected, too. All paths from the root node to the object contain the dominator. If all references to the dominator are removed, all paths from the root to the object are broken, leaving the object as garbage.

Interesting information can be deduced from the dominator tree:

- The rootpath from some object to the root object is shorter and more significant in the dominator tree than in the original heap graph.
- The garbage collector size of an object can be calculated easily from the dominator tree. It is the sum of the static sizes of all direct or indirect dominated objects and the static size of the object.

The dominator tree was first used by MAT to calculate the retained size and set [6].

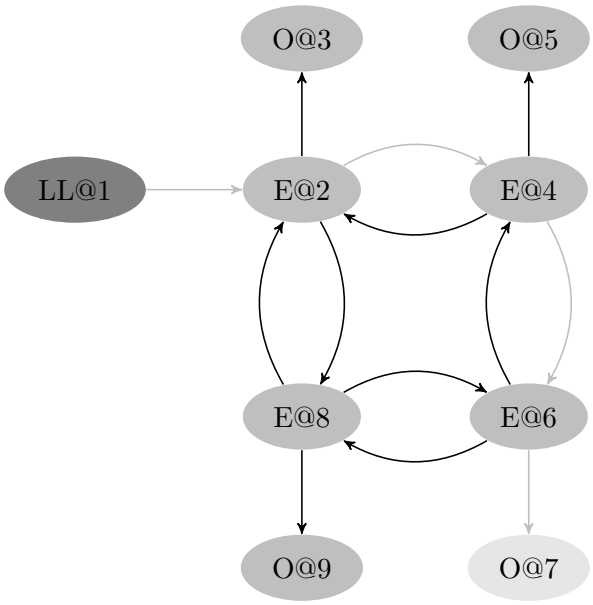

Figure 4.3: Real World Example Graph

The paths from an object to the root nodes are important inputs to the memory analysis process. The root paths allows to deduce why the selected object was not garbage collected. Root paths for heap dump reference graphs can be very long. An example for a long root path is a linked list. Fig. 4.3 shows a double linked list. The light grey arrows mark the root path *O@7* → *E@6* → *E@4* → *E@2* → *LL@1*. Fig. 4.4 shows the dominator tree of the graph presented in Fig. 4.3. The same root path is also marked with light grey arrows but the path *O@7* → *E@6* → *LL@1* is much shorter than the root path in the reference graph. The length of the root path in the dominator tree is independent from the size of the linked list. In contrast, the length of the root path in the reference graph is directly dependent on the number of objects in the linked list.

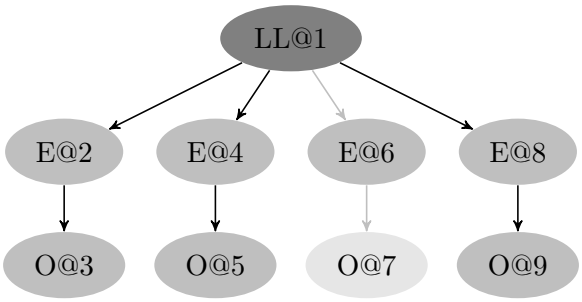

Figure 4.4: Real World Dominator Tree Example

In this work the two algorithms described by Thomas Lengauer and Robert Tarjan in [5] were implemented. The first simple algorithm has a runtime complexity of

$O(m * log(n))$, where m is the number of edges and n is the number of vertices. The second algorithm has a runtime complexity of $O(m * \alpha(m, n))$, where $\alpha(m, n)$ is a functional inverse of Ackermann's function. The runtime complexity of the second sophisticated algorithm is almost linear. The original algorithm described in the paper is a recursive one. It was developed for the use with control flow graphs. Heap dump reference graphs are much larger than control flow graphs in general. Both algorithms must be adapted from a recursive to an iterative implementation for the use with heap dumps. An iterative version is achieved by using stacks to store local variables. The memory consumption of the algorithm can be calculated: Nine integer arrays are used in the sophisticated algorithm, seven in the simple algorithm. The size of the arrays is the number of objects, arrays and classes. Some extra memory is necessary to buffer access to the heap dump files. After calculating the dominator tree, it is written into a file in the according heap dump directory.

4.2 Garbage Collector Size

The dominator tree is the basis for the garbage collector size algorithm. The garbage collector size for all nodes of the heap graph can be calculated by traversing the dominator tree in depth first order. For each node, the garbage collector size of all children must be summed up with the static size of the node.

```
1  int[] gcsize[n];
2  dfs(root);
3
4  void dfs(int id) {
5    gcsize[id] = getSize(id);
6
7    for (int childid : getChildren(id)) {
8      gcsize[id] += dfs(childid);
9    }
10 }
```

Listing 4.1: garbage collector size algorithm

A pseudo code representation of the garbage collector size algorithm can be seen at Lst. 4.1. The original algorithm is implemented recursively. For the use with heap dumps the algorithm must be changed, similar to the dominator tree algorithm, to an iterative implementation.

Each node must only be visited once. It follows, that the runtime complexity of this algorithm is $O(n)$, where n is the number of vertices. The memory consumption of the algorithm is low because only one array with the number of objects, arrays and classes

as size is used to store the calculated garbage collector size. Furthermore, some extra memory is needed to buffer file access on the heap dump files. After gc size calculation the array containing the sizes is written to a file in the heap dump directory. To optimize memory consumption of the algorithm the garbage collector size array can be saved. If a gc size value is computed, it can be written to the corresponding file at once.

4.3 Dynamic Size Simple

Apart from the garbage collector size algorithm, the dynamic size algorithm is the second algorithm in this thesis. Differences between the two sizes are discussed in detail in Chap. 2.

The simplest way to calculate the dynamic size is to sum up the static sizes of all nodes reachable from it.

```
1  int[] dynsize[n];
2  boolean[] visited[n];
3
4  for (int id : getAllNodes()) {
5    dynsize[id] = dfs(id);
6    for (int i; i<visited.length; i++) {
7      visited[i] = false;
8    }
9  }
10
11 int dfs(int id) {
12   int size = getSize(id);
13
14   for (int childid : getChildren(id)) {
15     if (!visited[childid]) {
16       visited[childid] = true;
17       size += dfs(childid);
18     }
19   }
20
21   return size;
22 }
```

Listing 4.2: simple object dynamic size algorithm

The pseudo code representation of the simple dynamic size algorithm can be seen in Lst. 4.2. The presented algorithm is a recursive implementation, for the use with heap

dumps the algorithm must be changed, similar to the other algorithm, to an iterative version.

The runtime complexity of the dynamic size algorithm a node is similar to the garbage collector size algorithm, $O(n)$, where n is the number of vertices. The runtime complexity of the algorithm to calculate the dynamic size for all nodes is $O(n^2)$ in the worst case. This happens if all other nodes are visited to calculate the dynamic size of one node. The memory consumption of the algorithm is as small as the consumption of the garbage collector size algorithm. Only one integer array is used to store the calculated size. Buffers are used to speed up the access to the file which contains the reference information. After computation, the size information is stored in a file in the heap dump directory. The memory consumption can be optimized by omitting the dynamic size array, as it is done for garbage collector size algorithm,.

4.4 Strongly Connected Components

The first idea to reduce runtime complexity is to calculate strongly connected components (SCC) and use the identified components instead of nodes. First some definitions:

Definition 4 (path) *If $G = (V, E)$ is a graph, a path $p : v \overset{*}{\Rightarrow} w$ in G is a sequence of vertices and edges leading from v to w [16].*

Definition 5 (simple path) *A path is simple if all its vertices are distinct [16].*

Definition 6 (closed path) *A path $p : v \overset{*}{\Rightarrow} v$ is called a closed path. A closed path $p : v \overset{*}{\Rightarrow} v$ is a cycle if all its edges are distinct and the only vertex to occur twice in p is v, which occurs exactly twice [16].*

Definition 7 (strongly connected) *Let G be a directed graph. Supposed that for each pair of vertices v, w in G there exist paths $p_1 : v \overset{*}{\Rightarrow} w$ and $p_2 : w \overset{*}{\Rightarrow} v$. Then G is said to be strongly connected [16].*

Definition 8 (strongly connected components) *Let G be a directed graph. Two vertices v and w are equivalent if there is a closed path $p : v \overset{*}{\Rightarrow} v$ which contains w. Let the distinct equivalence class under this relation be $\mathcal{V}_i$, $1 \leq i \leq n$. Let $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$,*

where $\mathcal{E}_i = \{(v, w) \,|v, w \epsilon \mathcal{V}_i\}$. *Then each* $\mathcal{G}_i$ *is strongly connected and no* $\mathcal{G}_i$ *is a proper subgraph of a strongly connected components of* G. *The subgraphs* $\mathcal{G}_i$ *are called strongly connected components [16].*

The definitions are explained by the following example from [16]. Fig. 4.5 shows the example graph, Fig. 4.6 shows strongly connected components in the graph and Fig. 4.7 shows the reduced graph. A lot of paths can be found in Fig. 4.5. An example is the path $4 \rightarrow 5 \rightarrow 6$. This path is a simple path: it contains each node exactly once. It is not a closed path because the first node (4) occurs only once. The path $4 \rightarrow 5 \rightarrow 3 \rightarrow 7 \rightarrow 4$ is a closed path.

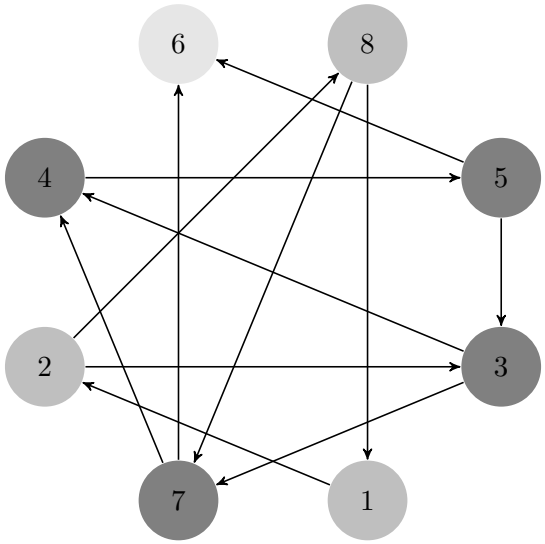

Figure 4.5: SCC Example Graph

Strongly connected heap dump reference graphs are very rare. There exist no strongly connected heap dump graphs from real world applications. But nearly all heap dump graphs contain SCCs.

Each node of the strongly connected components in Fig. 4.6 is part of one or more *closed paths*. It follows, that each SCC consists of at least one cycle. The light grey cycle of the SCC consists of only one node. This is the simplest possible cycle. The grey SCC consists of one cycle and the dark gray SCC of two cycles.

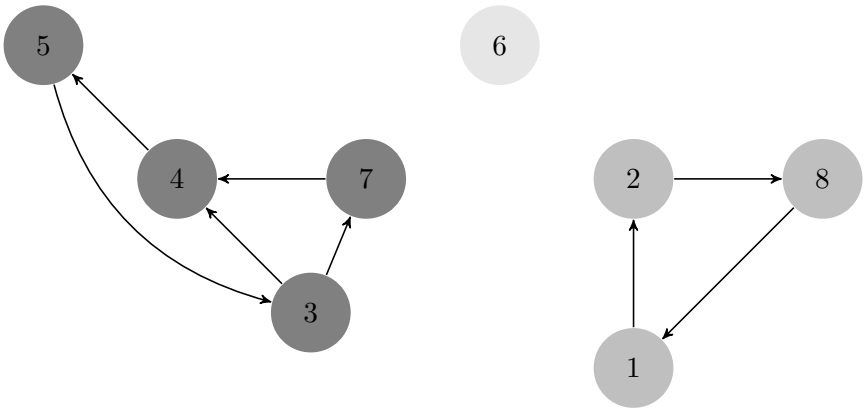

Figure 4.6: SCCs

For heap dump analysis two characteristics are of interest.

- It is possible to reach each other node from each node of the strongly connected component.
- If each SCC is represented by one node, the resulting graph will be a DAG.

The first characteristic is important to reduce the number of nodes for which the dynamic size must be calculate. The dynamic size of all nodes contained in a strongly connected component is equal for each node. This follows from the fact that each node of the SCC can reach all other nodes of the same SCC. A node can be reached by all nodes of the SCC if it is reachable by one node of the SCC. The simple dynamic size algorithm can be optimized by using the strongly connected components graph in place of the heap dump reference graph. Fig. 4.7 shows the SCC graph of the reference graph from Fig. 4.5. The strongly connected components graph is created by representing each SCC as a single node and only edges between two SCC's are registered. To reduce the number of references, double references between the SCC's are removed. The advantage of this characteristic is that for the dynamic size algorithm the number of nodes and references can be reduced. On the other hand overhead computing time for the calculation of strongly connected components graph is needed.

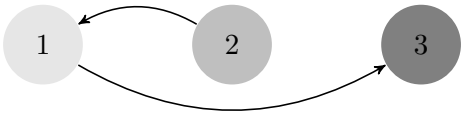

Figure 4.7: SCC Graph

The second characteristic of the strongly connected components is that the SCC graph is a DAG, hence no back edges or loops exist. This property is used by the next dynamic size calculation algorithm. The *BFS based Dynamic Size Algorithm.*

As example Fig. 4.8 shows the SCC graph of the reference graph from Fig. 4.1.

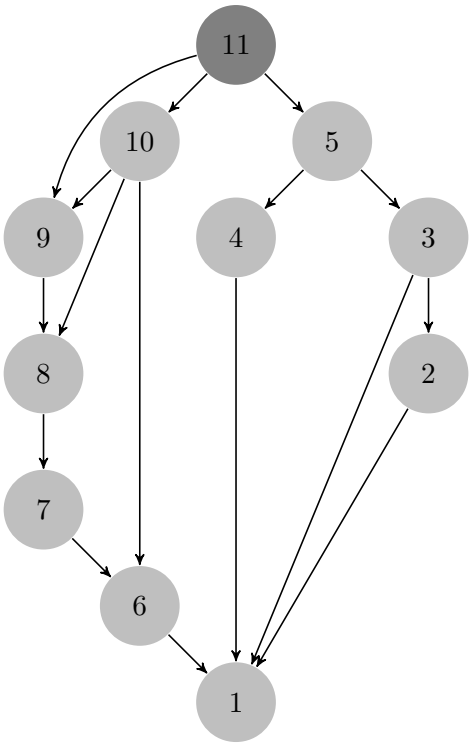

Figure 4.8: Example DAG

To calculate the strongly connected components the algorithm from Robert Tarjan, described in [16], was implemented. The runtime complexity of the strongly connected components algorithm is $O(n)$, where n is the number of vertices in the original heap dump reference graph. The algorithm must be adapted to an iterative implementation. The algorithm uses three integer arrays of size n.

Applying the strongly connected components algorithm does not reduce the complexity of the dynamic size algorithm, but it reduces the number of nodes and references for which the dynamic size must be calculated. Furthermore, the memory consumption of the algorithm is smaller because the array consists of fewer nodes.

4.5 BFS based Dynamic Size Algorithm

This algorithm traverses the SCC graph in reverse breadth first order. The algorithm starts with the leaves and follows the references back to the roots. All nodes at the same breadth level know the ids of all the other nodes which the dynamic size consists of. The ids are stored in the ids hash set. In the beginning only the leaves hold such a hash set. Lst. 4.3 shows the algorithm.

```
1  int[] dynamicsize[n], componentsize[n];
2  boolean[] visited[n];
3
4  NodeData vTemp, currentTemp;
5  Queue queue = new LinkedList();
6  HashMap hashMap = new HashMap();
7  componentsize = calculateStaticComponentSize();
8  addLeaves(); // add leaves to queue and hashmap
9
10 while (!queue.isEmpty()) { // run bfs based graph traversal
11   int current = queue.remove();
12   currentTemp = hashMap.get(current);
13   if (currentTemp.total == currentTemp.visited) {
14     hashMap.remove(current);
15     currentTemp.value.add(current);
16     for (int index : currentTemp.value)
17       dynamicsize[current] += componentsize[index];
18   } else { queue.offer(current); continue; }
19
20   for (int v : getReferrers(current)) {
21     if (!visited[v]) { visited[v] = true; queue.offer(v); }
22
23     if (hashMap.containsKey(v)) {
24       vTemp = hashMap.remove(v);
25     } else {
26       vTemp = new NodeData();
27       vTemp.value = new HashSet(currentTemp.value.size());
28       vTemp.total = getRefereesSize(v);
29     }
30     vTemp.value.addAll(currentTemp.value);
31     vTemp.visited++;
32     hashMap.put(v, vTemp);
33   }
34 }
```

Listing 4.3: BFS based dynamic size algorithm

The `queue` and the `visited` array are used for the BFS based graph traversal algorithm. The `dynamicsize` array stores the calculated dynamic size of each node and `componentsize` array stores the static size each component. The static size of all components is calculated at the beginning of the algorithm (`calculateStaticComponentSize()`). The `hashMap` is used to store information about all nodes at the same breadth level. The id of a component is used as key for the hash map. The value is an object with the following three data fields:

1. **total:** The number of the referrees of the node
2. **visited:** The number of the visited referrees.
3. **value:** The ids hash set, which was mentioned above, to store all the ids on which they dynamic size of the node exists.

The graph must not contain loops, a property guaranteed when using the SCC graph instead the heap dump reference graph.

Before a node can be visited all of its children must be visited. If all children were visited the node data is removed from the `hashMap` and the dynamic size is calculated by summing the static sizes of the components which ids are enclosed in the ids hash set. See Lst. 4.3 line 11 to 18.

For each child (referrer) of the visited node the hash set of the child must be merged with the hash set of the node and stored in the `value` entry of the child record. Furthermore, the `visited` entry of the child record must be incremented. If no record exists in the `hashMap`, a new one must be created. See Lst. 4.3 line 23 to 32.

The biggest advantage of this algorithm is that each node has to be visited only once. The downside of this algorithm is its massive memory consumption. It can be used only for heaps with few objects. Fig. 4.9 shows the memory usage from a Java process running the BFS based dynamic size calculation algorithm. The maximum memory used was limited to 3 GB. The heap dump analyzed was generated by a dynaTrace Server with a heap size of 85 MB. The dump contains 1409000 vertices and 3081501 edges.

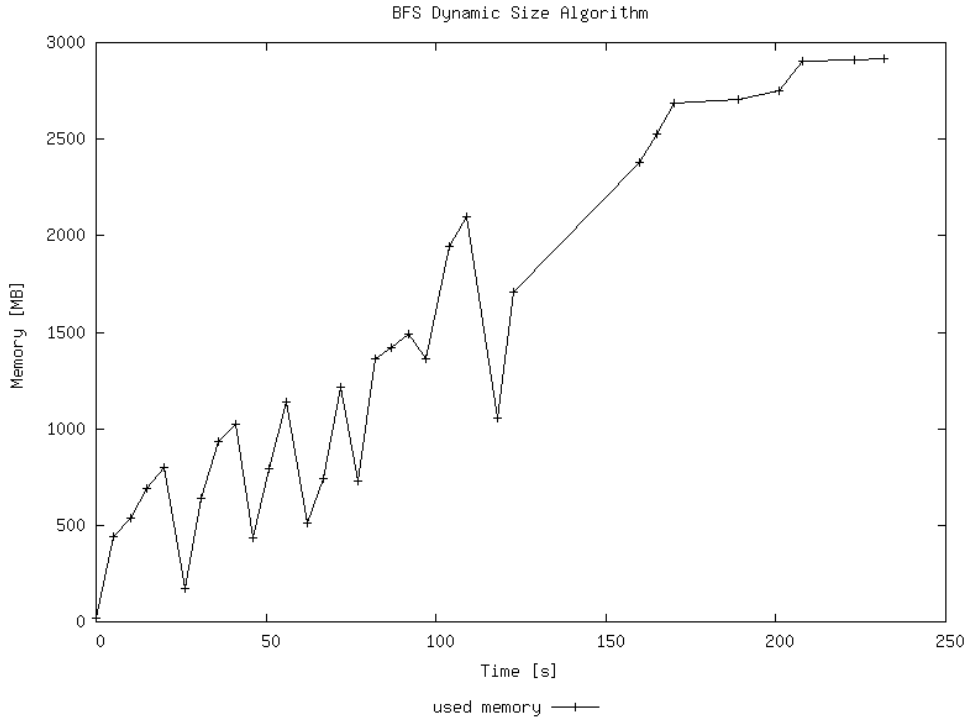

Figure 4.9: BFS based dynamic size algorithm memory consumption

4.6 Articulation Points

The next idea to reduce runtime complexity of the dynamic size algorithm is to calculate the articulation points of the SCC graph. First some definitions:

Definition 9 (biconnected) *Let $G = (\mathcal{V}, \mathcal{E})$ be an undirected graph. Suppose that for each triple of distinct vertices v, w, a in $\mathcal{V}$ there is path $p : v \overset{*}{\Rightarrow} w$ such that a is not on the path p. Then G is biconnected [16].*

Definition 10 (articulation point) *If there is a triple of distinct vertices v, w, a in $\mathcal{V}$ such that a is on any path $p : v \overset{*}{\Rightarrow} w$, and there exists at least one such path, then a is called an articulation point of G [16].*

The basis for the articulation point algorithm is an undirected graph, but the strongly connected components graph is directed. To create a undirected graph from the directed one, for each reference in one direction, a second one must be produced in the opposite direction. Furthermore, all double references which were produced by the transformation of the graph must be eliminated.

Fig. 4.10 shows the example graph from the paper [16]. The articulation points are marked light gray and the different biconnected components are marked with different grey levels.

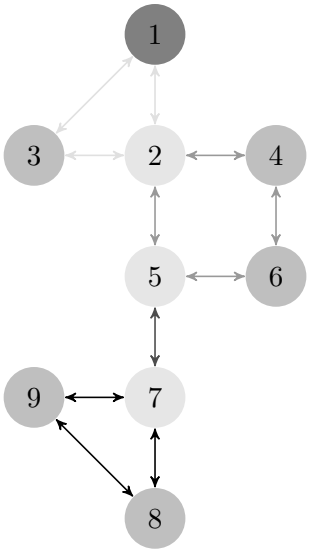

Figure 4.10: BCCs and Articulation Points

Using articulation points allows to divide the graph in two independent parts at the articulation point. It is possible to use the dynamic size of the articulation points without following references. The maximum number of nodes reachable form a node is the number of nodes in the biconnected component (BCC). But the articulation point dynamic size algorithm has a defect. It is possible to count some nodes multiple times. If the dynamic size for node *1* in Fig. 4.11 is computed and the dynamic size from articulation point *2* is used, the size of node *4* will be counted twice.

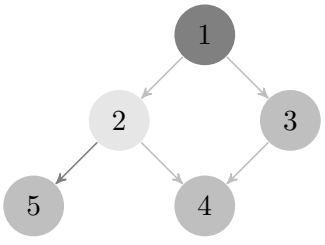

Figure 4.11: Articulation Points Defect

The defect was detected by using the delta debugger implemented for this work, explained later in this chapter.

The defect can be fixed in two ways. For example by using articulation points which are not part of the same BCC as the node for which the dynamic size is calculated. The advantage of this method is that all articulation points can be used and that checking whether an articulation point is in the same BCC as the used node is simple. If the node is not an articulation point, it is part of exactly one BCC. The ids of the BCCs of each articulation point must be stored in a hash set. The drawback of this method is the high complexity when testing if the articulation points belong to the same BCC. It can be done by intersecting the two BCC hash sets of the articulation points. If the intersection is the empty set, the articulation points are not part of the same BCC. This test increases the run time of the algorithm significantly. For this reason it is not used.

The second method uses articulation points where all children in the original directed SCC graph are part of the same biconnected component only. In this case, all parents in the SCC graph of the point are in one BCC and all children are in another. It follows, that all children of the point in the SCC graph can only be reached by using this articulation point. The advantage of this method is that the points can be calculated once and stored in a file. It is not necessary to check if the size can be used each time the point is reached, as it is done in the first method. The drawback is that a lot of articulation points must be canceled.

To calculate the biconnected components the algorithm presented by Robert Tarjan in [16] was implemented. The runtime complexity of the biconnected components algorithm is $O(n)$, where n is the number of vertices in the strongly connected components graph. The algorithm needs to be adapted, similar to the other algorithms, to an iterative implementation. A large portion of the memory used is required for two integer and one hash set arrays and one list. The list is used to store articulation points and the hash set array stores the membership to any biconnected components for each node. Another big part of the memory is used to remove double references in the undirected graph by using a bit set. For each new recursive step a new bit set is needed. This means, that many bit sets must be produced and garbage collected.

Two articulation point dynamic size algorithms were implemented. Two fixes were implemented to overcome the problem with the original definition of articulation points. Both algorithms did not reduce the runtime complexity of the original dynamic size algorithm, but they reduced the number of nodes needed to calculate the dynamic size of a particular node.

4.7 Delta Debugger

A delta debugger was implemented to find bugs in the heap dump analysis algorithms. For example, delta debugging was used to find a bug in the first implementation of the articulation point dynamic size algorithm, described previously in this chapter.

The delta debuger uses the `ddmin` algorithm described by Andreas Zeller in the book *Why programs fail* [17]. The `ddmin` algorithm simplifies the input of a program, containing a defect, to the minimum size. For the heap dump analysis algorithms delta debugging minimizes the number of references needed to reproduce a defect. The number of objects and the number of classes remain the same at each step of delta debugging. The algorithm attempts, similar to binary search, to eliminate the largest blocks of a given input. For each temporary input created, the test must be executed. In a `ddmin` run either all references of an object are contained or non at all. The result of the run decides whether the selected part of the input is needed to reproduces the bug. For the articulation point dynamic size algorithm, the test first calculates the dynamic size of the root node using articulation points then without articulation points. If the sizes differ the bug can be reproduced with the selected input.

4.8 Dominating Articulation Points

To overcome the problem that emerges from using articulation points dominating articulation points can be used. First some definitions:

Definition 11 (dominating articulation points) *A dominating articulation point is an articulation point which dominates all other nodes that can be reached directly or indirectly.*

The definition is explained by the following example. Fig. 4.12 shows the example graph and Fig. 4.13 the corresponding dominator tree. Articulation points are marked dark grey and dominating articulation points light grey.

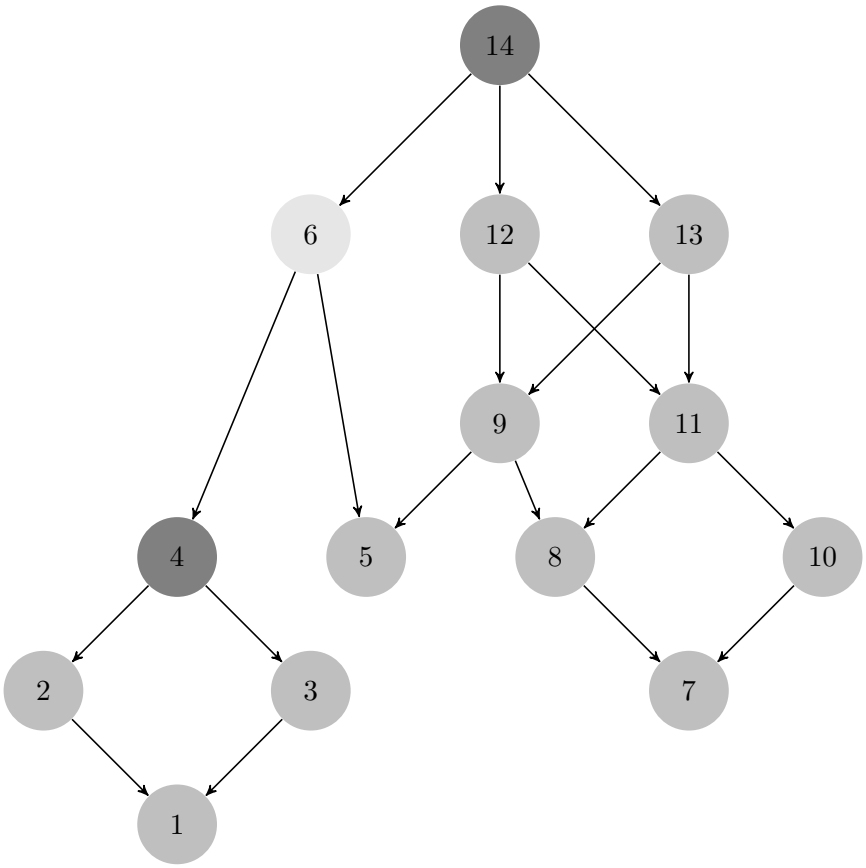

Figure 4.12: Dominating Articulation Points Example

The dominating articulation point algorithm consists of three steps. The first two steps are preprocessing steps for the actual algorithm.

1. Topologically sort the nodes of the strongly connected components graph.
2. Calculate the dominator tree for the SCC graph.
3. Compute the dominating articulation points.

The first step can be omitted if the strongly connected components algorithm from Robert Tarjan is used. The numbers of the components are in a topological order because the algorithm is based on the DFS algorithm. The dominator tree for the SCC graph is calculated via the simple dominator tree algorithm, described previously.

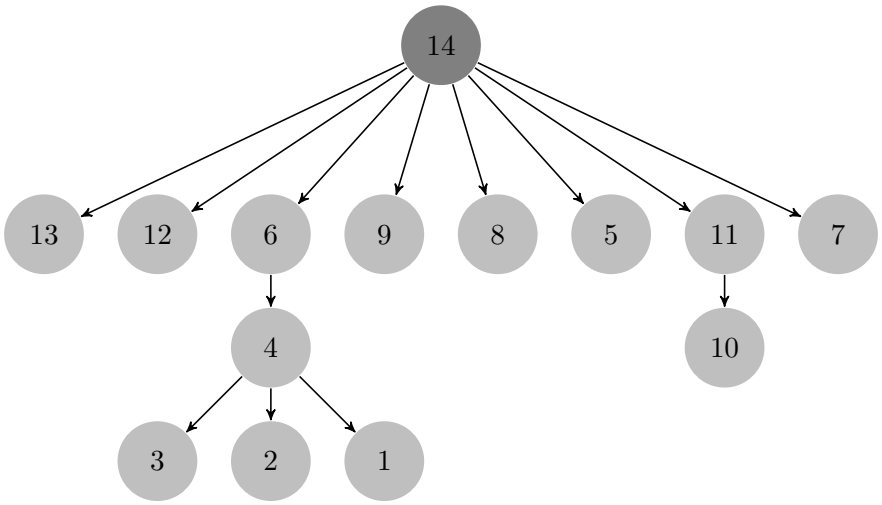

Figure 4.13: Dominator Tree

The dominating articulation points algorithm reads the dominators from the file and iterates over the SCCs, starting at the first SCC (number one). For each node the maximum of the ids of the dominators of all nodes which can be reached from this node is calculated. In a first step, the maximum of the values of all children in the SCC graph is computed. If the calculated maximum is equal to the SCC number of the node, then the node is a dominating articulation point. The last step is to calculate the number for the node by building the maximum of the maximum value of the children and the SCC number of the dominator of the node. Lst. 4.4 shows the algorithm in pseudo code.

```
1  int max;
2  int[] value[n];
3  List articpoint;
4
5  for (int i = 1; i < value.length; i++) {
6    max = 0;
7
8    for (int element : getReferees(i)) {
9      max = max(max, value[element]);
10   }
11
12   if (max == i) {
13     articpoint.add(i);
14   }
15
16   value[i] = max(max, dom[i + 1] - 1);
17 }
```

Listing 4.4: dominating articulation points algorithm

The value of each node is equal to the id of the dominator which is closest to the root node. If the highest of the values of all child nodes is equal to the SCC id of the node, then the node is a dominator for all directly or indirectly reachable nodes. It follows, that the node is also an articulation point, because all paths to any node behind the node contain this node. Therefore, the graph can be splitted into two independent parts at this point.

The maximum of the values of all children of the node *1* is 0 because the node has no children. Zero is not equal to one, therefore the node is not a dominating articulation point and the number of the node is 4, the maximum of the dominator id 4 and 0. The same can be done for nodes *2* and *3*. The maximum value of the children is 4 and their own number is also 4. Neither are dominating articulation points. The maximum of the children for node *4* is 4. It follows, that node *4* is a dominating articulation point.

The runtime complexity of the dominating articulation point algorithm is $O(n)$, where n is the number of vertices in the strong connected components graph. The memory consumption of the algorithm is low, only one integer array of size n and a list is needed. The integer array stores the calculated numbers for each node and the list stores the dominating articulation points found.

The use of the dominating articulation points does not reduce the complexity of the dynamic size algorithm, but it reduces the number of nodes that are necessary to compute the dynamic size for one node further.

Chapter 5

Results

The algorithms are implemented as JUnit tests. Each test loads the necessary data from files in the heap dump directory, executes the algorithm and stores the results in other files in the directory. For this thesis, the following algorithms have been implemented:

- **GCRootsAndLeaves:** The basis for all other algorithms. To meet the requirements from Chap. 3, no root node information is stored in the collected heap dump data. This implementation computes the roots and leaves of the graph and stores them in different files in the heap dump directory.
- **SophisticatedDominator** and **SimpleDominator:** The implemented dominator tree algorithms. They read the roots and leaves from the file created by the *GCRootsAndLeaves* algorithm and stores the calculated dominators into an file.
- **StrongConnectedComponents:** It depends on the *GCRootsAndLeaves* algorithm and computes strongly connected components and stores them in the heap dump directory.
- **SCCGCRootsAndLeaves:** Similar to the *GCRootsAndLeaves* algorithm. It uses the SCC graph instead of the heap dump reference graph.
- **ArticulationPoints:** Calculates the articulation points and biconnected components on the heap graph and stores them.
- **SCCDominatorTree:** The simple dominator tree algorithm used on the SCC graph.
- **DomArticulationPoints:** Computes the dominating articulation points and stores them in the heap dump directory.
- **GCSize:** The garbage collector size algorithm depending on the dominator tree.
- **BFSDynamicSize:** The BFS based dynamic size algorithm, using the SCC graph as basis.

- **SimpleObjectsDynamicSize:** The simple dynamic size algorithm depending on the heap dump reference graph.
- **SimpleCompDynamicSize:** The simple dynamic size algorithm. It depends on the SCC graph.
- **DomArticPointsDynamicSize:** Computes the dynamic size by using the SCC graph and the dominating articulation points.
- **ArticPointsDynamicSize1:** The dynamic size algorithm which uses the articulation points and the biconnected components. The articulation points are only used if they are in another BCC than the node for which the dynamic size is calculated.
- **ArticPointsDynamicSize2:** Similar to the *ArticPointsDynamicSize1* algorithm, but articulation points are only used if all of their children are part of the same BCC.

The tests were executed using an ant script on a computer with an Intel Xeon Duo 3,2 GHz CPUs and four GB RAM running Windows XP. Each test was run three times for each algorithm. From the running times of the individual runs, the average value was calculated and used for charting and tables. Running time of the algorithms is roughly the execution time of the JUnit test which also includes the time needed to read data and to store results in a file in the heap dump directory. The charted values are the sum of the running times of all algorithms, necessary to compute the size value. For example, the charted values of the *GCSize* algorithm is the sum of the running times of the *GCRootsAndLeaves*, the *SimpleDominator* and the *GCSize* algorithm. The implemented algorithms are tested with the following heap dump types.

5.1 Generated Heap Dumps

The generated heap dumps are used to test the algorithms with simple artificial heap dumps. An advantage of the generated dumps is, that the computed sizes for nodes can be examined easily. For the DAG and the unbalanced tree example graphs, the sizes can be computed from the id of the node. A drawback of these dumps is, that they do not have much in common with heap dumps of genuine applications. Furthermore it is not possible to test the strongly connected components algorithm. For the generated heap dumps, each node is its own component. From this follows that the *SimpleObjects-DynamicSize* algorithm is always faster than the *SimpleCompDynamicSize* algorithm. Twenty Heap dumps were generated. The smallest is of size 1001 nodes and the largest of size 10001 nodes.

Balanced Tree Example Graph

The balanced tree based heap dump graph is the simplest of the generated graphs. The structure is a simple balanced binary tree. All nodes are objects of the same class. The class and the root object for the tree are garbage collector roots. For these heap dumps, not only the SCC graph is equal to the heap dump reference graph, but also dominator tree. The number of reachable from a given node is much smaller than in all other generated heap dumps.

Fig. 5.1 shows measured running times. The simple objects dynamic size algorithm is the fastest, the dominating articulation point algorithm is the slowest. The running times of the algorithms are sorted by the number of the other algorithms which are a prerequisite for the algorithm. For example the *SimpleCompDynamicSize* algorithm has three prerequisite. The *GCRootsAndLeaves*, *StrongConnectedComponents* and *SCCGCRootsAndLeaves* algorithms. The *SimpleObjectsDynamicSize* algorithm has only one prerequisite. The *GCRootsAndLeaves* algorithm. Not even the dominating articulation points algorithm brings improvement, though each node, except the leaves, are dominating articulation points. The time to calculate the dominating articulation points is higher than the time gained by using the dominating articulation points.

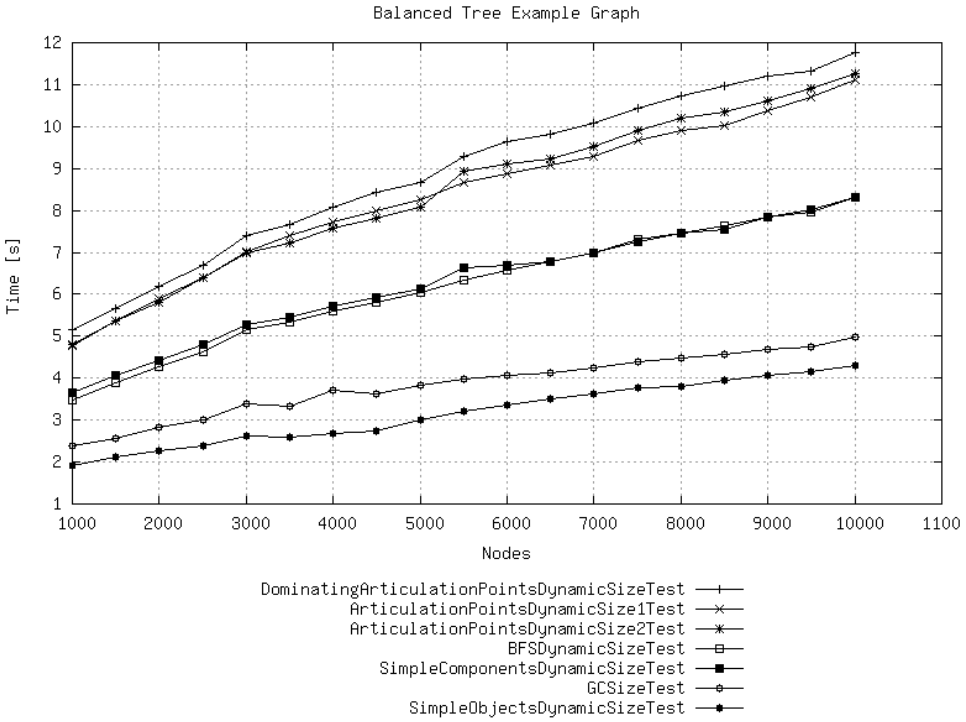

Figure 5.1: Balanced Tree Chart

The result of tests using the balanced tree example graph heap dumps show that the average number of nodes necessary to calculate the dynamic size of a node is too small to show improvement. The overhead of the algorithms is higher than the improvement. Tab. A.5 contains the average running times of all algorithms.

Unbalanced Tree Example Graph

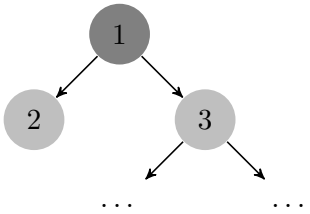

Figure 5.2: Unbalanced Tree Example Graph

The unbalanced tree example graph is similar to a simple linked list with data objects. The nodes with odd ids represent list entries, even ids represent data objects. All nodes are objects of the same class. The class and the root object for the tree (the head of the list) are garbage collector roots.

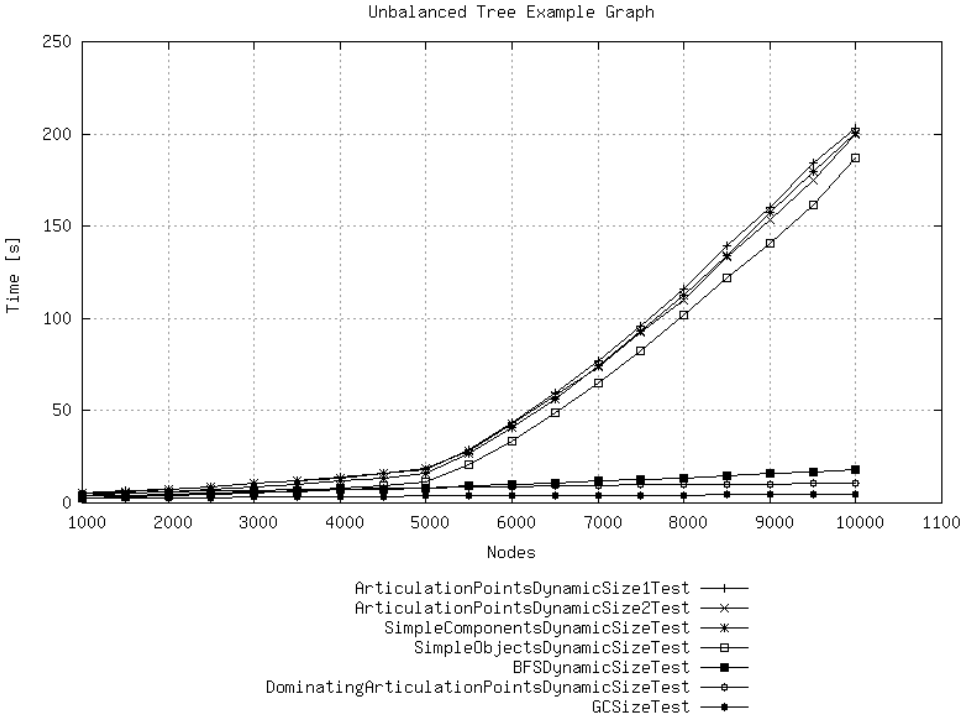

Figure 5.3: Unbalanced Tree Chart

Heap dumps organized in this manner are well suitable for the dominating articulation points dynamic size algorithm, because to calculate the dynamic size of each entry node, only the data node and the next entry node are needed. Each entry node is a dominating articulation point.

Fig. 5.3 shows the running times for unbalanced tree heap dumps. As predicted, the dominating articulation points dynamic size algorithm was the fastest dynamic size implementation. Another interesting observation is that the articulation points and simple dynamic size algorithms have nearby quadratic run time complexity, visualized in the next figure.

Tab. A.6 contains the average running times of all algorithms.

DAG Example Graph

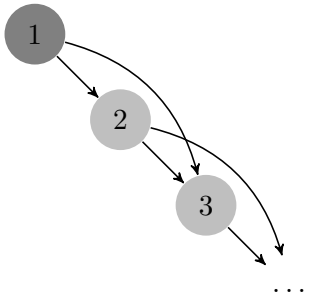

Figure 5.4: DAG1 Example Graph

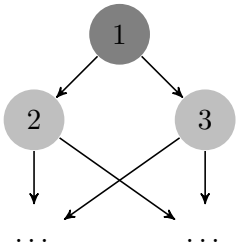

Figure 5.5: DAG2 Example Graph

The DAG example heap dumps are used to test the heap dump analysis algorithms on more complex heap dumps. Fig. 5.6 and 5.7 shows running times measured for produced DAG heap dumps. In neither examples articulation or dominating articulation points were found. As a result, it is not possible to decrease running times of the algorithms. Tab. A.7 and A.8 present the average running times of all algorithms.

The BFS based dynamic size algorithm works well for all tests with the generated heap dumps. It is the fastest for the DAG heap dumps. For the unbalanced tree heap dumps only the dominating articulation points algorithm was faster. The generated heap dumps are small enough that the algorithm can get along with the available memory.

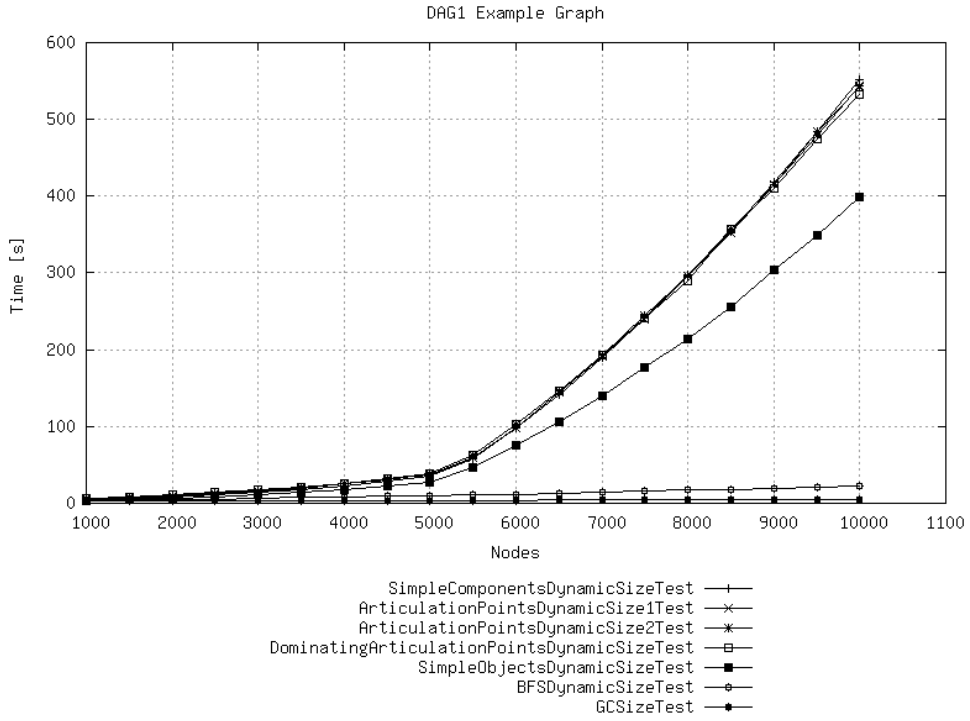

Figure 5.6: DAG1 Chart

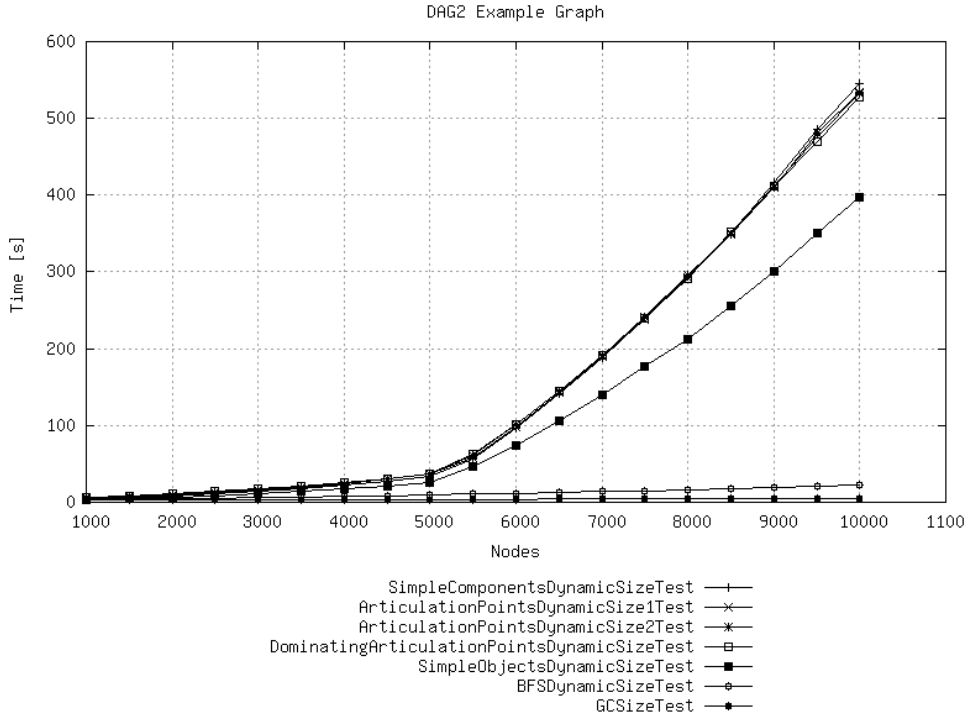

Figure 5.7: DAG2 Chart

5.2 SimpleLoadGenerator Heap Dumps

The SimpleLoadGenerater generates an object graph of a given size in a Java virtual machine. The objects graph generally consists of two parts. The first part is a *City* objects containing *Person* objects. The ABC objects use the majority of the memory. They consist of one 1 MB large double array. The number of *City* objects depends on the size of memory. 90 % of the memory is filled with *City* objects. The remaining memory is filled up with an tree of *GrandMother*, *GrandFather*, *Mother*, *Father*, *Son* and *Daughter* objects.

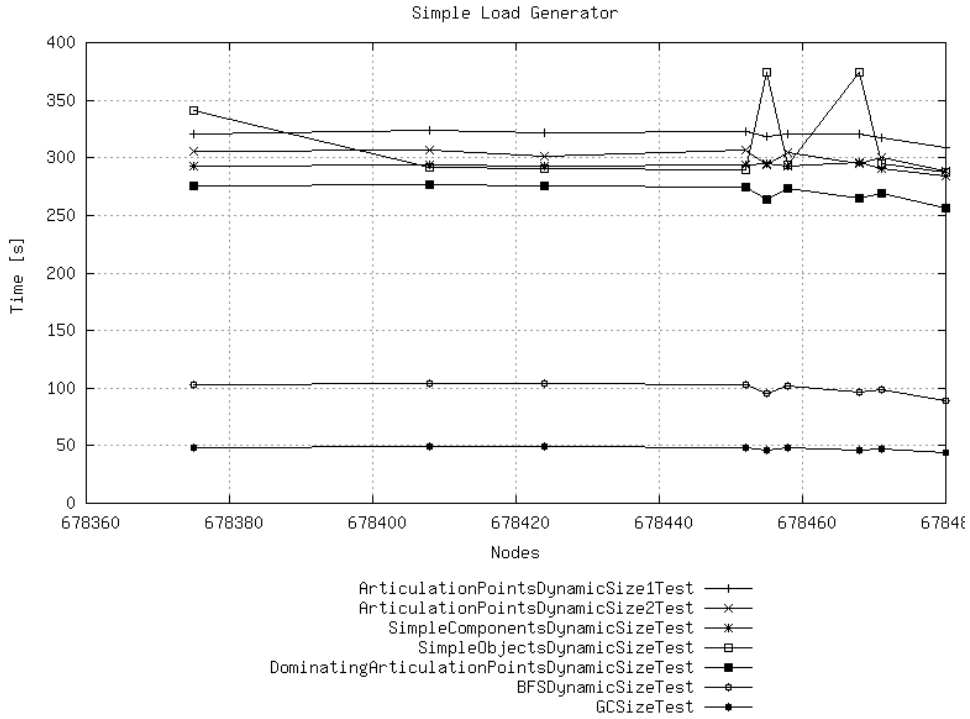

Figure 5.8: Simple Load Generator Data Chart

Nine heap dumps were provided. The smallest heap dump generated by the SimpleLoadGenerator was 10 MB and the largest 90 MB. For all these dumps, the number of nodes is almost the same for all of them. The smallest dump consists of 678375 nodes and the largest of 678480 nodes. The difference only 105 nodes. This can be explained by the 1 MB large double array in the *Person* object.

The fastest dynamic size algorithm was the BFS based one. It is the result of the small number of conditions and that each node is used only once. All other dynamic size algorithms are equivalently fast. Only the dominating articulation point algorithm

was a little faster than the others. The generated object graph is similar to a tree. It follows, that it contains many dominating articulation points, which improves the running time of the dominating articulation points dynamic size algorithm. Tab. A.3 contains the average running times of all algorithms.

Apart from the running times of the algorithms also some statistical values for the graphs were determined. The values can be found in Tab. 5.1.

Nodes	Classes	Objects	References	Roots	none trivial SCC	average SCC size	Articulation Points	Dominating Artic. Points	average SCC DAG height	average SCC DAG width	used Dom. Artic. Points
678375	844	677531	675428	4493	30	8.03	323414	323028	12.97	2.10	331411
678452	844	677608	675462	4542	30	8.03	323435	323047	12.97	2.10	331634
678424	844	677580	675465	4511	30	8.03	323444	323056	12.97	2.10	331643
678455	844	677611	675494	4514	30	8.23	323464	323076	12.97	2.10	331852
678458	844	677614	675496	4514	30	8.03	323474	323086	12.97	2.10	331673
678408	844	677564	675468	4486	30	8.03	323471	323087	12.97	2.10	331437
678468	844	677624	675528	4493	30	8.23	323503	323115	12.97	2.10	331891
678471	844	677627	675530	4493	30	8.03	323513	323125	12.97	2.10	331712
678480	844	677636	675533	4499	30	8.03	323522	323134	12.97	2.10	331721

Table 5.1: Simple Load Generator Statistics Table

A trend which can be read from the table is, that the number of strongly connected components for the *SimpleLoadGenerater* heap dumps is relatively small. This explains why the *SimpleComponentsDynamicSize* algorithm is not faster than the *SimpleObjectsDynamicSize* algorithm. Another observation is that the number of dominating articulation points is rather high. Roughly each second node is a dominating articulation point. Furthermore they were frequently used.

5.3 dynaTrace Server Heap Dumps

Six different heap dumps of the dynaTrace server were provided. The smallest heap size was 20 MB and the largest was 115 MB. For these heap dumps, only two of the

six dynamic size algorithms return a result. The BFS based and the articulation point algorithm run out of memory for all dumps. The simple object dynamic size algorithms did not produce results because twice the running time of the simple components dynamic size algorithm was considered timeout. Tab. A.1 contains the average running times of all algorithms.

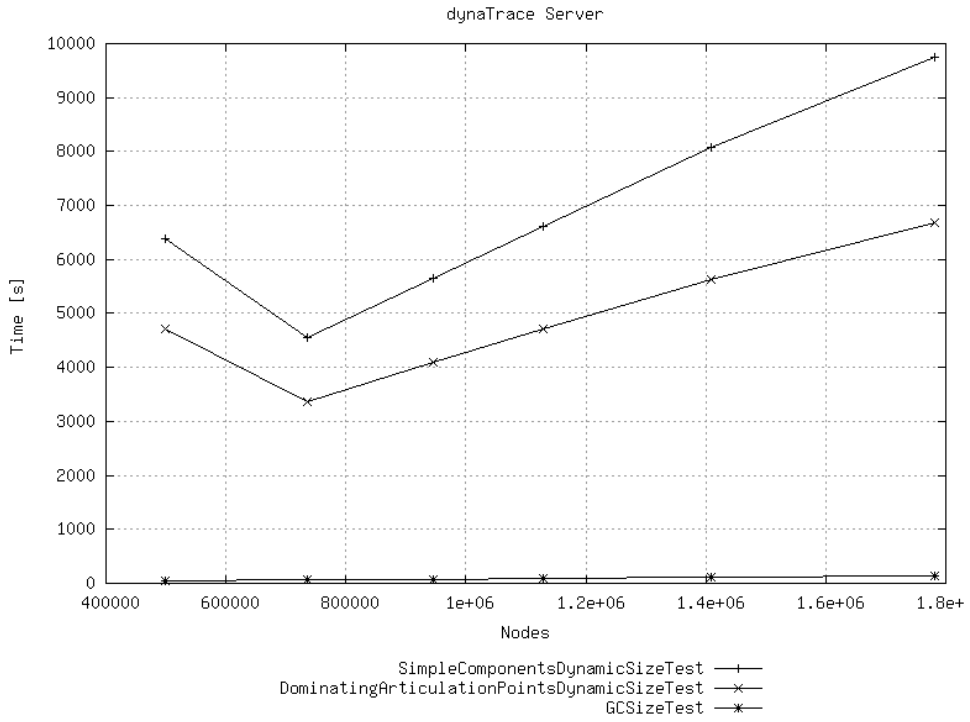

Figure 5.9: dynaTrace Server Data Chart

The experiments helped to conclude that the use of dominating articulation points improves the running time of the dynamic size algorithm. As shown in Fig. 5.9 the dominating articulation points algorithm working on heap dumps from the dynaTrace server runs twenty to thirty percent faster than the simple components dynamic size algorithm. Faster runtime can be explained with the statistic values, which were collected for the heap dumps. Tab. 5.2 contains the statistic values for the dynaTrace Server heap dump graphs.

Nodes	Classes	Objects	References	Roots	none trivial SCC	average SCC size	Articulation Points	Dominating Artic. Points	average SCC DAG height	average SCC DAG width	used Dom. Artic. Points
497860	7136	490724	772425	47882	11327	9.37	138027	106238	14.06	2.91	58702402
734962	7792	727170	1285127	46687	11478	13.98	213998	180596	15.71	3.23	52783542
945087	7792	937295	1841021	46301	11478	19.99	283929	250708	15.60	3.55	78544958
1128750	7792	1120958	2342892	45705	11478	25.61	348937	315532	15.55	3.80	103443382
1409000	7792	1401208	3081501	45938	11481	33.58	441648	408216	15.46	4.00	138311806
1780673	7795	1772878	4060032	46161	11482	44.15	564439	531073	14.46	4.17	173207168

Table 5.2: dtServer Statistics Table

One can observe that the number of dominating articulation points and the number of dominating articulation points used is rather high. The value of the points used is 80 to 100 times higher than the number of nodes in the heap dump reference graph. This observation explains why the *DominatingArticulationPointsDynamicSize* algorithm is faster than the *SimpleComponentsDynamicSize* algorithm. The *DominatingArticulationPointsDynamicSize* algorithm hits dominating articulation points very often when the dynamic size of a node is computed. This means, that the algorithm can use the dynamic size of the node without following the references of them. The *SimpleComponentsDynamicSize* algorithm must follow these references. Therefore, the simple algorithm must traverse a much larger tree and needs more runtime for the computation of the same size.

Another observation is that the number and the size of strongly connected components is much higher than for the *SimpleLoadGenerater* heap dumps. This helps to explains why the *SimpleObjectsDynamicSize* algorithm for the dynaTrace server heap dumps always time out. The *SimpleComponentsDynamicSize* algorithm must visit all nodes in the SCC graph which can be reached from the node for which the dynamic size is computed. Similarly, the *SimpleObjectsDynamicSize* algorithm must visit all reachable nodes in the dump reference graph. If the number and size of strongly connected components is high, the number of nodes traversed by the *SimpleComponentsDynamicSize* algorithm is much smaller than the number of nodes traversed by the *SimpleObjectsDynamicSize* algorithm. It follows, that the *SimpleObjectsDynamicSize* algorithm needs much more computing time to calculate the same sizes.

The average SCC DAG height and width can be used to make a rough estimate for the runtime of the *SimpleComponentsDynamicSize* algorithm. The two values influence the running time but they do not determine it solely. A formula approximating the algorithms run time must be found for an improved version of the heap dump analysis algorithm.

5.3.1 Differences between Dynamic and Garbage Collector Size

The computation of the dynamic size takes much longer than the computation of the garbage collector size. It must be investigated whether the computation of the dynamic size is necessary. It is possible that the garbage collector size is sufficient. This could be found out by calculating the difference between the two sizes, sort the differences and group them. The number of nodes with a large differences between the sizes is interesting. Memory leaks in such nodes can often only be found by comparing the dynamic size of them because usually the garbage collector size of these nodes is small.

Nodes	Heap Size [MB]	to 1 KB	1 to10 KB	10 to 100 KB	100 to 1000 KB	1 to 10 MB	10 to 25 MB	25 to 50 MB	50 to 75 MB	75 to 100 MB	greater 100 MB
497861	20	383225	33010	23887	28140	12	29587	0	0	0	0
734963	35	561765	37643	24927	33862	14	1	76751	0	0	0
945088	50	703069	37639	24930	33669	14	1	145766	0	0	0
1128751	65	822092	37659	24926	33846	14	1	3	210210	0	0
1409001	85	1010766	37659	24926	33863	14	1	0	12	301760	0
1780674	115	1261157	37673	25488	33781	14	1	0	0	6	422554

Table 5.3: Size Difference Statistics Table

Tab. 5.3 shows the size difference for the heap dumps generated from the dynaTrace Server. The differences between the dynamic size and the gc size is smaller than one MB for most nodes. In all heap dumps examined it is possible to reach almost all other nodes from a fifth to a quarter of nodes. For these nodes the dynamic size is an important information to find memory leaks because the garbage collector size of them is much smaller. Garbage collector size is equal to the static size form most of these nodes. For future work this investigation must be made for as much as possible real world applications and heap dumps.

Chapter 6

Conclusion

The results presented in the previous chapter show that it was possible to improve the computation time for the dynamic sizes. Improvements were gained by introducing the following changes to the algorithm:

Running time was decreased by identifying and using strongly connected components instead of the original nodes of the graph. The SCC graph was smaller than the original heap dump reference graph for all heap dumps tested. It consists not only of less nodes, but also of fewer references. Because nodes in the original graph are grouped together in the SCC graph, double references between certain nodes in the SCC graph can be omitted. The smaller number of nodes and references has a shorter running time of the *SimpleComponentsDynamicSize* algorithm as consequence. It performs better than the *SimpleObjectsDynamicSize* algorithm. The magnitude of the improvement depends on the number and the size of the not trivial SCC nodes. For dynaTrace server heap dumps, the *SimpleComponentsDynamicSize* algorithm was more than twice as fast as the *SimpleObjectsDynamicSize* algorithm.

Another improvement for the running time is to identify nodes in the SCC graph (articulation points) that allow to divide the SCC graph into serveral independent subgraphs. The advantage of using such nodes is that they allow to reuse the dynamic size of the nodes without computing them again. The original articulation point dynamic size algorithm had a defect. It was possible to count some nodes multiple times (for details check Fig. 4.11). To overcome this problem, the definition of dominating articulation points was introduced in this work. No comparable definitions were found in literature about graph algorithms. A dominating articulation point must comply the following properties:

1. The node is required to be an articulation point.
2. All reachable nodes must be dominated.

An algorithm that computes dominating articulation points was developed in this work and is presented in Chap. 4. They help to overcome the problem of counting certain nodes multiple times because all nodes reachable from a dominating articulation point can only be reached via the dominating articulation point.

The *DominatingArticulationPointsDynamicSize* algorithm runs twenty to thirty percent faster than the *SimpleComponentsDynamicSize* algorithm on heap dumps generated from the dynaTrace server.

Another insight gained in this work is, that it is meaningful to compute the dynamic size, additionally to the garbage collector size. The experiments with the dynaTrace server heap dumps show that the garbage collector size differs significantly from the dynamic size for many objects.

Chapter 7

Future Work

Test of Dynamic Size and GC Size in the Memory Leak Detection Process

The benefits of using the dynamic size and the garbage collector size must still be evaluated on real world applications with memory leaks. Moreover, the analysis algorithms should be tested more extensively on heap dumps of real world applications. Some tests suggest that the dynamic size algorithm could be improved: The possibility to cluster nodes in order to reduce the calculation time of the dynamic size seems worth further investigation. It seems possible to cluster objects using their classes.

Heap Dump diffing

For heap dump analysis, it is important to be able to compare heap dumps. No research towards heap dump comparison was done in this work. Heap dump diffing can be realized by making use of graph isomorphisms. Two Graphs G and H are isomorphic graphs if there exists a structure-preserving vertex bijection $f : V_G \rightarrow V_H$ [3]. A well known tool for the computation of graph isomorphisms is *nauty*, presented in [7].

Root Node and Control Flow Information

The heap dump algorithms, introduced in Chap. 3, do not contain root node or control flow information in order to suffice the strict time and memory consumption requirements. For further analysis of the heap dumps this information could be useful. It

still needs to be investigated if it is possible to improve the implementation of the heap dump algorithm such that it collects root node information, too.

For root nodes originating from the stack stacktraces are interesting. A stacktrace helps the user of the memory analysis tool to differ between long and short living stack root nodes. The lifetime of a stack root node depends directly on the lifetime of the containing method. Because each stacktrace belongs to a thread, it can be helpful to know which thread or thread group is responsible for the stacktrace. Different approaches are required because of the differences between Java and .Net managed and native JNI or .Net unmanaged stacks.

In the Java VM threads and monitors are garbage collector root nodes. For threads, the name and the thread group are of interest. If the root node is a monitor, memory analysis can be combined with thread analysis. If, for example, a monitor is not released, it is helpful to know which threads wait for the monitor and which thread holds it. With this information it seems easier to detect why a monitor is not released.

Appendix A

Tables

Appendix A list all tables of all measured running times and statistic values. Tab. A.1 and Tab. A.2 shows running times and statistical values for the tests using the dtServer heap dumps. Check Tab. A.3 and Tab. A.4 for the results of the Simple Load Generator heap dump tests. The results of the generated heap dump tests are listed in Tab. A.5, Tab. A.6, Tab. A.7 and Tab. A.8

Nodes	497860	734962	945087	1128750	1409000	1780673
GCRootsAndLeaves	14.08	17.89	20.51	23.04	26.83	31.12
SimpleDominator	23.60	33.36	43.13	52.32	64.42	81.96
SophisticatedDominator	24.37	33.58	43.93	52.35	65.88	82.50
StrongConnectedComponents	15.47	26.94	28.38	36.23	42.19	50.14
SCCGCRootsAndLeaves	11.98	15.63	18.05	20.46	23.85	28.35
ArticulationPoints	39.92	67.79	97.83	125.67	240.78	389.65
SCCDominatorTree	23.89	32.65	41.31	49.35	60.84	77.27
DomArticulationPoints	12.60	15.79	19.34	21.88	26.45	31.59
GCSize	5.86	8.03	9.96	10.81	14.50	17.89
BFSDynamicSize	0.00	0.00	0.00	0.00	0.00	0.00
SimpleCompDynamicSize	6354.19	4502.66	5590.59	6537.21	7981.86	9628.27
DomArticPointsDynamicSize	4634.36	3250.94	3958.70	4552.88	5449.21	6472.95
SimpleObjectsDynamicSize	0.00	0.00	0.00	0.00	0.00	0.00
ArticPointsDynamicSize1	0.00	0.00	0.00	0.00	0.00	0.00
ArticPointsDynamicSize2	0.00	0.00	0.00	0.00	0.00	0.00

Table A.1: dynaTrace Server Table

Nodes	497860	734962	945087	1128750	1409000	1780673
Classes	7136.00	7792.00	7792.00	7792.00	7792.00	7795.00
Objects	490724.00	727170.00	937295.00	1120958.00	1401208.00	1772878.00
References	772425.00	1285127.00	1841021.00	2342892.00	3081501.00	4060032.00
Roots	47882.00	46687.00	46301.00	45705.00	45938.00	46161.00
Leaves	182838.00	277293.00	347797.00	410762.00	505067.00	630750.00
SCC	403026.00	585965.00	727077.00	846290.00	1034983.00	1285279.00
none trivial SCC	11327.00	11478.00	11478.00	11478.00	11481.00	11482.00
maximum SCC size	26180.00	75658.00	144673.00	209117.00	300675.00	421530.00
average SCC size	9.37	13.98	19.99	25.61	33.58	44.15
SCC Roots	47882.00	46687.00	46301.00	45705.00	45938.00	46161.00
SCC Leaves	183200.00	277655.00	348159.00	411124.00	505429.00	631118.00
Articulation Points	138027.00	213998.00	283929.00	348937.00	441648.00	564439.00
Dominating Articulation Points	106238.00	180596.00	250708.00	315532.00	408216.00	531073.00
maximum Dominator Tree width	74705.00	75941.00	77323.00	92735.00	92735.00	92735.00
average Dominator Tree width	2.12	2.00	1.87	1.79	1.73	1.68
maximum SCC Dominator Tree width	51499.00	93584.00	162670.00	226900.00	318447.00	439315.00
average SCC Dominator Tree width	2.25	2.26	2.20	2.17	2.14	2.11
maximum Dominator Tree height	285.00	344.00	344.00	344.00	344.00	337.00
average Dominator Tree height	7.22	6.17	6.30	6.41	6.50	6.58
maximum SCC Dominator Tree height	33.00	33.00	33.00	33.00	33.00	33.00
average SCC Dominator Tree height	5.59	5.66	5.38	5.21	5.02	4.85
maximum SCC DAG height	61.00	60.00	60.00	60.00	60.00	59.00
average SCC DAG height	14.06	15.71	15.60	15.55	15.46	14.46
maximum SCC DAG width	47882.00	241980.00	518074.00	775479.00	1141553.00	1625076.00
average SCC DAG width	2.91	3.23	3.55	3.80	4.00	4.17
Dominating Articulation Points used	58702402.00	52783542.00	78544958.00	103443382.00	138311806.00	173207168.00
Articulation Points 1 used	0.00	0.00	0.00	0.00	0.00	0.00
Articulation Points 2 used	0.00	0.00	0.00	0.00	0.00	0.00

Table A.2: dynaTrace Server Full Statistics Table

Nodes	678375	678408	678424	678452	678455	678458	678468	678471	678480
GCRootsAndLeaves	7.61	7.79	7.73	7.66	8.07	7.99	7.85	7.72	7.77
SimpleDominator	33.47	34.12	33.85	33.07	30.27	32.99	30.33	31.97	28.51
SophisticatedDominator	34.14	34.60	33.79	34.15	30.30	33.39	30.67	32.30	28.91
StrongConnectedComponents	23.63	23.16	23.08	22.86	20.95	22.57	21.17	22.29	19.68
SCCGCRootsAndLeaves	18.66	18.84	18.55	18.67	16.70	18.26	17.06	17.70	15.78
ArticulationPoints	56.17	56.71	55.86	56.16	54.46	55.51	54.55	55.24	53.44
SCCDominatorTree	31.81	31.78	32.57	31.36	28.96	31.74	29.73	30.68	27.67
DomArticulationPoints	16.92	17.22	17.06	16.83	15.29	16.60	15.61	16.36	14.80
GCSize	7.46	7.44	7.55	7.51	7.52	7.51	7.49	7.45	7.49
BFSDynamicSize	53.43	54.33	54.57	53.52	49.51	52.71	50.23	51.40	46.06
SimpleCompDynamicSize	243.20	244.13	243.20	244.82	249.52	243.85	250.06	243.18	240.67
DomArticPointsDynamicSize	177.14	177.96	176.86	176.77	173.39	176.09	173.57	174.81	170.70
SimpleObjectsDynamicSize	333.69	284.34	282.64	282.22	366.49	286.07	366.93	287.23	279.37
ArticPointsDynamicSize1	215.10	217.10	217.03	217.50	218.47	216.64	220.51	214.51	212.59
ArticPointsDynamicSize2	199.78	200.31	196.42	201.57	194.05	200.51	194.68	197.46	192.05

Table A.3: Simple Load Generator Table

Nodes	678375	678452	678424	678455	678458	678408	678468	678471	678480	1347237
Classes	844.00	844.00	844.00	844.00	844.00	844.00	844.00	844.00	844.00	844.00
Objects	677531.00	677608.00	677580.00	677611.00	677614.00	677564.00	677624.00	677627.00	677636.00	1346393.00
References	675428.00	675462.00	675465.00	675494.00	675496.00	675468.00	675528.00	675530.00	675533.00	1344293.00
Roots	4493.00	4542.00	4511.00	4514.00	4514.00	4486.00	4493.00	4493.00	4499.00	4500.00
Leaves	353987.00	354035.00	353998.00	354007.00	354002.00	353962.00	353981.00	353976.00	353976.00	701729.00
SCC	678165.00	678242.00	678214.00	678239.00	678248.00	678198.00	678252.00	678261.00	678270.00	1347020.00
none trivial SCC	30.00	30.00	30.00	30.00	30.00	30.00	30.00	30.00	30.00	30.00
maximum SCC size	69.00	69.00	69.00	69.00	69.00	69.00	69.00	69.00	69.00	69.00
average SCC size	8.03	8.03	8.03	8.23	8.03	8.03	8.23	8.03	8.03	8.27
SCC Roots	4493.00	4542.00	4511.00	4514.00	4514.00	4486.00	4493.00	4493.00	4499.00	4500.00
SCC Leaves	353989.00	354037.00	354000.00	354009.00	354004.00	353964.00	353983.00	353978.00	353978.00	701731.00
Articulation Points	323414.00	323435.00	323444.00	323464.00	323474.00	323471.00	323503.00	323513.00	323522.00	644523.00
Dominating Articulation Points	323028.00	323047.00	323056.00	323076.00	323086.00	323087.00	323115.00	323125.00	323134.00	644135.00
maximum Dominator Tree width	26749.00	26747.00	26745.00	26744.00	26742.00	26740.00	26739.00	26737.00	26735.00	53485.00
average Dominator Tree width	2.10	2.10	2.10	2.10	2.10	2.10	2.10	2.09	2.09	2.09
maximum SCC Dominator Tree width	26749.00	26747.00	26745.00	26744.00	26742.00	26740.00	26739.00	26737.00	26735.00	53485.00
average SCC Dominator Tree width	2.10	2.10	2.10	2.10	2.10	2.10	2.10	2.10	2.09	2.09
maximum Dominator Tree height	21.00	21.00	21.00	21.00	21.00	21.00	21.00	21.00	21.00	21.00
average Dominator Tree height	11.96	11.96	11.96	10.00	11.96	11.96	10.00	11.96	11.96	10.08
maximum SCC Dominator Tree height	20.00	20.00	20.00	20.00	20.00	20.00	20.00	20.00	20.00	20.00
average SCC Dominator Tree height	10.02	10.02	10.02	10.02	10.02	10.02	10.02	10.02	10.02	10.09
maximum SCC DAG height	38.00	38.00	38.00	38.00	38.00	38.00	38.00	38.00	38.00	38.00
average SCC DAG height	12.97	12.97	12.97	12.97	12.97	12.97	12.97	12.97	12.97	13.06
maximum SCC DAG width	26749.00	26747.00	26745.00	26744.00	26742.00	26740.00	26739.00	26737.00	26735.00	53485.00
average SCC DAG width	2.10	2.10	2.10	2.10	2.10	2.10	2.10	2.10	2.10	2.09
Dominating Articulation Points used	331411.00	331634.00	331643.00	331852.00	331673.00	331437.00	331891.00	331712.00	331721.00	652924.00
Articulation Points 1 used	2082.00	2163.00	2163.00	2201.00	2163.00	2078.00	2201.00	2163.00	2163.00	2209.00
Articulation Points 2 used	333130.00	333456.00	333495.00	333739.00	333585.00	333335.00	333868.00	333714.00	333753.00	654947.00

Table A.4: Simple Load Generator Full Statistics Table

Nodes	1001	1501	2001	2501	3001	3501	4001	4501	5001	5501	6001	6501	7001	7501	8001	8501	9001	9501	10001
GCRootsAndLeaves	0.83	0.88	0.93	0.99	1.19	1.07	1.12	1.15	1.21	1.25	1.29	1.29	1.32	1.38	1.41	1.45	1.48	1.50	1.57
SimpleDominator	0.92	1.01	1.16	1.25	1.38	1.44	1.73	1.58	1.65	1.78	1.79	1.86	1.91	1.98	2.03	2.07	2.13	2.17	2.24
SophisticatedDominator	0.93	1.03	1.13	1.23	1.35	1.43	1.51	1.56	1.65	1.76	1.83	1.86	1.93	2.01	2.04	2.09	2.16	2.18	2.24
StrongConnectedComponents	0.81	0.90	1.00	1.09	1.19	1.28	1.32	1.40	1.43	1.54	1.58	1.61	1.63	1.70	1.77	1.79	1.84	1.85	1.98
SCCGCRootsAndLeaves	0.86	0.97	1.06	1.15	1.24	1.34	1.41	1.47	1.51	1.54	1.59	1.62	1.69	1.73	1.75	1.80	1.83	1.87	1.89
ArticulationPoints	1.05	1.19	1.34	1.53	1.66	1.78	1.90	1.96	2.01	2.07	2.07	2.16	2.22	2.32	2.31	2.32	2.37	2.50	2.65
SCCDominatorTree	0.93	1.04	1.15	1.24	1.38	1.43	1.53	1.59	1.59	1.71	1.75	1.77	1.86	1.94	1.99	2.06	2.10	2.09	2.12
DomArticulationPoints	0.81	0.88	0.95	1.02	1.08	1.13	1.18	1.23	1.27	1.36	1.41	1.44	1.46	1.50	1.57	1.59	1.62	1.64	1.72
GCSize	0.63	0.68	0.73	0.77	0.81	0.83	0.85	0.89	0.96	0.95	0.99	0.99	1.02	1.04	1.04	1.04	1.08	1.08	1.17
BFSDynamicSize	0.99	1.13	1.27	1.40	1.55	1.64	1.74	1.79	1.89	2.01	2.12	2.26	2.36	2.49	2.52	2.60	2.68	2.75	2.87
SimpleCompDynamicSize	1.14	1.33	1.42	1.58	1.66	1.76	1.87	1.90	1.98	2.29	2.22	2.27	2.34	2.43	2.52	2.52	2.69	2.79	2.88
DomArticPointsDynamicSize	0.91	0.98	1.10	1.20	1.32	1.41	1.53	1.59	1.66	1.89	2.01	2.09	2.11	2.18	2.24	2.28	2.33	2.36	2.47
SimpleObjectsDynamicSize	1.08	1.25	1.34	1.41	1.43	1.51	1.55	1.59	1.80	1.97	2.07	2.21	2.29	2.38	2.40	2.51	2.58	2.66	2.73
ArticPointsDynamicSize1	1.22	1.42	1.56	1.65	1.75	1.92	1.98	2.02	2.10	2.26	2.35	2.40	2.44	2.54	2.67	2.67	2.87	2.97	3.04
ArticPointsDynamicSize2	1.26	1.41	1.49	1.63	1.70	1.76	1.82	1.83	1.91	2.52	2.58	2.55	2.66	2.77	2.95	3.00	3.08	3.18	3.16

Table A.5: Balanced Tree Table

Nodes	1001	1501	2001	2501	3001	3501	4001	4501	5001	5501	6001	6501	7001	7501	8001	8501	9001	9501	10001
GCRootsAndLeaves	0.82	0.90	0.96	1.00	1.06	1.12	1.17	1.22	1.29	1.30	1.37	1.40	1.44	1.45	1.50	1.54	1.56	1.59	1.66
SimpleDominator	0.94	1.06	1.10	1.15	1.29	1.34	1.38	1.43	1.49	1.58	1.61	1.66	1.68	1.77	1.78	1.90	1.93	1.97	1.97
SophisticatedDominator	0.92	1.00	1.10	1.18	1.29	1.36	1.39	1.44	1.46	1.60	1.62	1.64	1.70	1.75	1.82	1.87	1.93	1.95	1.99
StrongConnectedComponents	0.81	0.90	0.96	1.02	1.10	1.15	1.20	1.23	1.29	1.36	1.40	1.47	1.50	1.56	1.56	1.59	1.64	1.66	1.85
SCCGCRootsAndLeaves	0.86	0.96	1.01	1.10	1.15	1.19	1.30	1.33	1.37	1.42	1.46	1.51	1.54	1.59	1.62	1.67	1.73	1.71	1.74
ArticulationPoints	1.07	1.21	1.36	1.55	1.67	1.75	1.88	1.91	1.98	2.09	2.13	2.12	2.15	2.22	2.26	2.30	2.34	2.39	2.54
SCCDominatorTree	0.93	1.03	1.14	1.25	1.33	1.44	1.49	1.54	1.56	1.63	1.68	1.68	1.76	1.81	1.82	1.87	1.90	1.92	1.97
DomArticulationPoints	0.83	0.89	0.92	0.98	1.02	1.06	1.10	1.12	1.19	1.25	1.28	1.32	1.35	1.38	1.39	1.44	1.42	1.47	1.52
GCSize	0.64	0.70	0.73	0.77	0.79	0.82	0.85	0.91	0.94	0.97	0.97	0.99	1.02	1.03	1.04	1.05	1.07	1.09	1.14
BFSDynamicSize	1.21	1.41	1.69	2.03	2.46	2.85	3.43	3.89	4.38	5.11	5.83	6.65	7.50	8.08	8.91	9.87	10.90	11.90	12.80
SimpleCompDynamicSize	1.81	2.44	3.34	4.13	5.25	6.51	8.10	9.90	11.97	22.52	36.34	51.79	69.69	88.65	107.85	129.35	152.33	174.34	195.46
DomArticPointsDynamicSize	0.88	0.98	1.11	1.16	1.28	1.36	1.42	1.47	1.55	1.61	1.74	1.77	1.85	1.94	2.02	2.10	2.10	2.17	2.19
SimpleObjectsDynamicSize	1.62	2.13	2.85	3.63	4.58	5.58	6.86	8.24	9.91	19.44	32.46	47.60	63.35	80.78	100.50	120.17	139.30	159.90	185.37
ArticPointsDynamicSize1	1.89	2.44	3.32	4.13	5.46	6.73	8.34	10.08	12.33	22.40	36.98	53.16	70.42	88.69	109.24	132.06	152.82	176.70	195.24
ArticPointsDynamicSize2	1.88	2.38	3.36	4.36	5.60	6.55	8.11	10.10	12.51	22.18	36.62	51.81	67.36	85.36	103.07	126.10	145.91	167.41	191.67

Table A.6: Unalanced Tree Table

Nodes	1001	1501	2001	2501	3001	3501	4001	4501	5001	5501	6001	6501	7001	7501	8001	8501	9001	9501	10001
GCRootsAndLeaves	0.83	0.89	0.93	0.99	1.04	1.07	1.12	1.16	1.19	1.25	1.28	1.33	1.32	1.34	1.35	1.43	1.44	1.49	1.50
SimpleDominator	0.97	1.06	1.18	1.25	1.37	1.41	1.45	1.51	1.56	1.67	1.74	1.76	1.80	1.86	1.88	1.89	1.93	2.05	2.05
SophisticatedDominator	0.97	1.07	1.17	1.24	1.35	1.37	1.44	1.48	1.58	1.68	1.74	1.77	1.79	1.86	1.88	1.93	2.00	2.03	2.06
StrongConnectedComponents	0.84	0.93	1.01	1.13	1.13	1.22	1.28	1.34	1.41	1.49	1.54	1.55	1.58	1.63	1.65	1.67	1.69	1.75	1.77
SCCGCRootsAndLeaves	0.87	0.92	0.98	1.02	1.08	1.10	1.15	1.19	1.22	1.24	1.26	1.29	1.31	1.32	1.36	1.36	1.38	1.39	1.39
ArticulationPoints	1.04	1.16	1.36	1.46	1.54	1.64	1.68	1.76	1.83	2.03	2.12	2.16	2.14	2.11	2.14	2.15	2.18	2.25	2.28
SCCDominatorTree	0.98	1.09	1.22	1.31	1.41	1.46	1.53	1.59	1.68	1.76	1.82	1.91	1.94	2.00	2.02	2.09	2.14	2.19	2.20
DomArticulationPoints	0.82	0.88	0.92	0.95	0.99	1.04	1.07	1.11	1.12	1.15	1.17	1.22	1.24	1.28	1.27	1.29	1.32	1.31	1.37
GCSize	0.63	0.68	0.71	0.75	0.80	0.81	0.84	0.87	0.93	0.94	0.98	0.99	1.01	1.01	1.03	1.07	1.06	1.07	1.18
BFSDynamicSize	1.24	1.57	1.99	2.47	3.05	3.88	4.55	5.25	5.95	7.01	7.82	8.86	10.05	11.26	12.54	13.87	15.30	16.80	18.51
SimpleCompDynamicSize	2.51	3.98	5.92	8.57	11.54	14.94	19.20	24.76	31.51	53.49	94.00	137.75	185.71	237.76	293.19	349.28	414.16	479.11	547.73
DomArticPointsDynamicSize	2.49	3.96	5.98	8.58	11.83	14.90	19.54	26.02	31.84	56.46	95.30	138.94	185.06	234.42	281.90	348.49	403.00	465.60	523.90
SimpleObjectsDynamicSize	2.27	3.60	5.26	7.35	9.89	13.00	16.62	20.83	25.45	45.45	73.60	105.32	138.86	175.57	212.93	254.04	302.22	348.05	397.21
ArticPointsDynamicSize1	2.48	4.03	5.57	7.94	11.03	15.14	20.02	26.11	32.11	52.72	92.35	138.12	185.63	233.94	289.99	346.37	407.77	477.52	535.12
ArticPointsDynamicSize2	2.52	4.01	6.13	8.79	11.67	15.02	19.71	25.28	32.24	56.47	96.21	140.51	186.73	238.76	288.83	350.43	407.72	472.36	537.33

Table A.7: DAG1 Table

Nodes	1001	1501	2001	2501	3001	3501	4001	4501	5001	5501	6001	6501	7001	7501	8001	8501	9001	9501	10001
GCRootsAndLeaves	0.82	0.88	0.93	1.00	1.03	1.08	1.11	1.16	1.19	1.24	1.26	1.35	1.35	1.34	1.38	1.42	1.44	1.46	1.48
SimpleDominator	0.97	1.07	1.18	1.26	1.35	1.42	1.48	1.49	1.59	1.67	1.73	1.75	1.83	1.83	1.87	1.92	1.95	2.06	2.05
SophisticatedDominator	0.97	1.06	1.17	1.26	1.35	1.38	1.45	1.48	1.58	1.66	1.71	1.76	1.81	1.86	1.88	1.93	1.97	2.02	2.08
StrongConnectedComponents	0.85	0.93	1.02	1.07	1.14	1.20	1.27	1.33	1.38	1.49	1.57	1.57	1.58	1.62	1.64	1.69	1.74	1.74	1.80
SCCGCRootsAndLeaves	0.85	0.90	0.94	1.00	1.03	1.08	1.14	1.17	1.16	1.22	1.24	1.26	1.25	1.28	1.33	1.31	1.32	1.32	1.32
ArticulationPoints	1.03	1.16	1.33	1.45	1.52	1.62	1.67	1.73	1.84	2.05	2.10	2.23	2.10	2.14	2.12	2.11	2.17	2.23	2.22
SCCDominatorTree	1.00	1.09	1.23	1.33	1.44	1.50	1.51	1.56	1.66	1.76	1.82	1.92	1.96	1.97	2.03	2.08	2.13	2.18	2.21
DomArticulationPoints	0.83	0.87	0.93	0.96	0.99	1.04	1.08	1.11	1.13	1.16	1.17	1.24	1.23	1.26	1.28	1.28	1.32	1.34	1.36
GCSize	0.63	0.67	0.71	0.76	0.79	0.82	0.84	0.86	0.91	0.93	0.96	0.98	0.99	1.01	1.07	1.03	1.06	1.08	1.15
BFSDynamicSize	1.23	1.55	1.96	2.40	2.95	3.67	4.42	5.01	5.74	6.71	7.52	8.52	9.70	10.86	12.16	13.50	14.84	16.35	18.02
SimpleCompDynamicSize	2.47	3.94	5.83	7.85	11.32	14.55	19.03	24.38	30.48	52.42	91.98	137.87	183.27	235.90	290.69	344.73	411.73	480.73	540.27
DomArticPointsDynamicSize	2.48	3.86	5.74	8.46	11.71	14.50	18.95	24.43	31.03	55.44	94.03	136.98	183.73	232.94	283.71	344.86	404.27	462.38	518.81
SimpleObjectsDynamicSize	2.28	3.52	5.20	7.30	9.94	12.88	16.50	20.49	24.91	45.09	72.48	104.33	138.69	175.56	211.33	254.45	298.89	348.60	395.61
ArticPointsDynamicSize1	2.49	4.04	5.81	7.82	10.81	14.89	19.29	25.07	30.86	52.18	91.63	136.28	183.91	232.02	286.09	343.47	405.05	469.49	526.09
ArticPointsDynamicSize2	2.56	3.88	5.84	8.54	11.54	14.81	19.16	24.69	31.05	55.81	95.37	138.19	185.38	235.47	290.21	342.86	403.08	473.52	526.86

Table A.8: DAG2 Table

Appendix B

Heapdumps Implementation

The algorithms and the used interfaces will now be illustrated:

B.1 JVMPI Heap Dump

The Java Virtual Machine Profiler Interface (JVMPI) [11] is available since version 1.2 in the Java Hotspot virtual machine. It is a bidirectional native interface. With the profiler interface it is possible to get memory profiling information from the Java virtual machine.

To create the extended or simple heap dump the `JVMPI_EVENT_HEAP_DUMP` with the `JVMPI_DUMP_LEVEL_0` is requested. The response of the event is a pointer to an array of records of the following format [11]:

- **ty:** type of object
- **jobjectID:** object ID

The type of the object field reports if the object is an object, a class or an object- or primitive array. The type field is not required for the JVMPI extended heap dump algorithm. Only the object ID field is used. To get more information about the object the `JVMPI_EVENT_OBJECT_DUMP` with the object ID as parameter is used. The response of the event, depends on the type of the object and is one of the following byte arrays.

The dump level 0 was selected to reduce the size of the array in the heap dump event result for the extended dump. If another dump level is used, the response array of the heap dump event contains byte arrays for classes, objects and arrays presented later.

The use of the dump level 0 reduces the needed memory but increases the CPU time which is needed for the creation of the heap dump. The extra CPU time is spent to request the object dump events. A benefit of this method is that it is possible to create heap dumps of larger virtual machines without hitting the border of the underlying machines memory. Nevertheless, information about the root objects is lost.

Class Dump

If the parameter of the object dump event was a class the response will be a byte array with the following structure where `(...)*` signals zero, one or more occurrences:

```
JVMPI_GC_CLASS_DUMP
jobjectID class
jobjectID super
jobjectID class loader
jobjectID signers
jobjectID protection domain
jobjectID class name (a String object, may be NULL)
void* reserved
u4 instance size (in bytes)
(jobjectID)* interfaces
u2 size of constant pool
(u2, constant pool index,
ty, type,
vl)* value
(vl)* static field values
```

For the heap dump only the `jobjectID class` and the static field values are of interest. The jobjectID pointer can be directly interpreted as the classes id. To get the name of the class and the types of the static references the `JVMPI_EVENT_CLASS_LOAD` can be used. The response of the class load event is the `class_load` struct. The name of the class can be taken from the struct by using the `char *class_name`. With the fields `jint num_static_fields`, `JVMPI_Field *instances` from the struct and the `(vl)* static field values` from the byte array, it is possible to iterate over the static fields of the class. If the type of the static field is a reference then the id of the referree must be sent to the server.

```
struct {
```

```
        char *class_name;
        char *source_name;
        jint num_interfaces;
        jint num_methods;
        JVMPI_Method *methods;
        jint num_static_fields;
        JVMPI_Field *statics;
        jint num_instance_fields;
        JVMPI_Field *instances;
        jobjectID class_id;
    } class_load;
```

Object Dump

If the parameter was an object, the byte array has the following structure:

```
JVMPI_GC_INSTANCE_DUMP
jobjectID object
jobjectID class
u4 number of bytes that follow
(vl)* instance field values
     (class, followed by super, super's super ...)
```

The object ID and the class ID are represented by reading the fields `jobjectID object` and `jobjectID class`. To get the static size the `JVMPI_EVENT_OBJECT_ALLOC` is used. The size is taken from the `jint size` field of the response struct `obj_alloc` of the object alloc event. The iteration over the fields of the object is very similar to the static fields of a class. The `jint num_instance_fields` and `JVMPI_Field *instances` fields from the response of the `JVMPI_EVENT_CLASS_LOAD` and the `(vl)* instance field values` from the array are used to get all field references of the object. The only difference is, that for parsing all instance field values, the class load event must be called for each class on the inheritance path from the class's object to `java.lang.Object`. This means that the event must be requested for the class of the object, for the super class, for the super super class and so on.

```
struct {
     jint arena_id;
     jobjectID class_id;
```

```
        jint is_array;
        jint size;
        jobjectID obj_id;
    } obj_alloc;
```

Object Array Dump

If the parameter of the object dump event was an object array, the response will be a byte array of following structure

```
JVMPI_GC_OBJ_ARRAY_DUMP
jobjectID array object
u4 number of elements
jobjectID element class ID
(jobjectID)* elements
```

The id of the array are represented by the `jobjectID array object` value. To get the size and the class ID the `JVMPI_EVENT_OBJECT_ALLOC` is used. To iterate over the array references, the values `u4 number of elements` and `(jobjectID)* elements` are used. Using class ID from the object alloc event can be a drawback for object arrays. For Sun and IBM virtual machines the returned class ID is the ID of the class of the elements of the array, for Oracle virtual machines the returned class ID is the ID of the class of the array. Special treatment is necessary at the server side!

Primitive Array Dump

If the parameter was a primitive array, the byte array has the following structure:

```
JVMPI_GC_PRIM_ARRAY_DUMP
jobjectID array object
u4 number of elements
ty element type
(vl)* elements
```

The id of the array object are represented, similar to object arrays, by the `jobjectID array object` value. To get the class ID and the size the `JVMPI_EVENT_OBJECT_ALLOC` is used.

For the simple heap dump both fields of the records in the response array of the heap dump event are used. If the type of the object is a primitive array the classname is known. To get the classname for objects or object arrays the class id must be determined. This is done by using the `JVMPI_EVENT_OBJECT_ALLOC` with the object id as parameter. With the class id as parameter for the `JVMPI_EVENT_CLASS_LOAD` the classname is determined. The size of the object or array is determined independently of the type by requesting the `JVMPI_EVENT_OBJECT_ALLOC`. With the classname it is possible to check if a matching entry in the hash map exists. If such an entry is found the number of instances must be incremented and the object size is added to the size value of the entry in the hash map. If such an entry does not exist a new one with the classname and the size of the object must be created and added to the hash map.

More about the Java Virtual Machine Profiler Interface (JVMPI) can be found at [11].

B.2 JVMTI Heap Dump

The Java Virtual Machine Tool Interface (JVMTI) [12] was integrated in the Java virtual machine version 5.0. It is also a bidirectional native interface. It is the successor of the JVMPI and the JVMDI (Java Virtual Machine Debug Interface).

The basis for heap dumps with JVMTI is the tag mechanism. For each loaded class, object or array instance, a tag can be set. The tag is a value of type long. To create an extended heap dump with the tool interface, each class and object must be tagged. The tag then is used to assign an unique ID to each node.

To create a extended heap dump the following steps must be performed:

1. Get all classes.
2. Iterate over the classes, tag each of them, get the name of the class and send the informations to the server,
3. Iterate over all objects, set the tag of each object, get the size of the object and send the information to the server.
4. Iterate over all references and send them to the server.
5. Iterate over all objects and untag them.
6. Iterate over the classes and untag them.

The function `GetLoadedClasses` returns the number of the classes (`jint* class_count_ptr`) and an array of all loaded classes (`jclass** classes_ptr`).

```
jvmtiError GetLoadedClasses(jvmtiEnv* env,
     jint* class_count_ptr,
     jclass** classes_ptr)
```

To tag each class and to get then name of them the functions `SetTag` and `GetClassSignature` can be used. The `GetClassSignature` returns the name of the class in the output parameter `char** signature_ptr`.

```
jvmtiError SetTag(jvmtiEnv* env,
     jobject object,
     jlong tag)
```

```
jvmtiError GetClassSignature(jvmtiEnv* env,
     jclass klass,
     char** signature_ptr,
     char** generic_ptr)
```

The `IterateOverHeap` function and the `jvmtiHeapObjectCallback` is used to iterate over all objects and to set the tag of them. The callback must be implemented and handed over to the function as a function pointer with the parameter `heap_object_callback`. The function calls the callback for all objects and arrays on the heap. All information about the object for which the callback was called are supplied as parameter. The tag of the object is set by changing the value of the reference parameter `tag_ptr` to the desired value. Values available as parameters `size`, `class_tag` and `tag_ptr` are transmitted to the server.

```
jvmtiError IterateOverHeap(jvmtiEnv* env,
     jvmtiHeapObjectFilter object_filter,
     jvmtiHeapObjectCallback heap_object_callback,
     void* user_data)
```

```
typedef jvmtiIterationControl (JNICALL *jvmtiHeapObjectCallback)
     (jlong class_tag, jlong size,
     jlong* tag_ptr, void* user_data);
```

The `IterateOverReachableObjects` function and the callback `jvmtiObjectReferenceCallback` are used to iterate over all references. The `jvmtiStackReferenceCallback` and the `jvmtiObjectReferenceCallback` are not used because no root information is collected. The `jvmtiObjectReferenceCallback` is called for each reference

exactly once. The tag of the referrer is the value of the `referrer_tag` and the tag of the referree is the value of the `tag_ptr` parameter.

```
jvmtiError IterateOverReachableObjects(jvmtiEnv* env,
     jvmtiHeapRootCallback heap_root_callback,
     jvmtiStackReferenceCallback stack_ref_callback,
     jvmtiObjectReferenceCallback object_ref_callback,
     void* user_data)
```

```
typedef jvmtiIterationControl (JNICALL *jvmtiObjectReferenceCallback)
     (jvmtiObjectReferenceKind reference_kind,
     jlong class_tag, jlong size,
     jlong* tag_ptr, jlong referrer_tag,
     jint referrer_index, void* user_data);
```

The values of the tags of all objects, arrays and classes needs be set to zero, deallocating the memory used by the tags. The functions `SetTag` and `IterateOverHeap` and the `jvmtiHeapObjectCallback` callback are used.

The JVMTI simple heap dump has minor differences to the other simple heap dump algorithms, instead of using a hash map an array is used, since the number of loaded classes is known.

Creating a simple heap dump is done the following way:

1. Get all classes.
2. Iterate over the classes, tag each of them, get the name of the class and initialize the array element on the tag position with the classname.
3. Iterate over all objects, get the size of the object, add them to the value of the corresponding array element and increment the instance count value of the array element.
4. Iterate over the classes and untag them.
5. Send the information to the server

Similar to the extended heap dump, the functions `GetLoadedClasses`, `SetTag` and `GetClassSignature` are used to tag all classes and to get their names. The `IterateOverHeap` function and the `jvmtiHeapObjectCallback` are used to collect the data from the objects and arrays. The `class_tag` parameter of the callback is used to identify the correct array element. The value of the `size` parameter is added to the variable

in the array element. Furthermore the instance counter of the array element must be incremented.

More about the Java Virtual Machine Tool Interface (JVMTI) can be found at [12].

B.3 .Net Profiling API Heap Dump

The profiling API [10] within CLR allows to the user to monitor execution and memory usage of a running application. Typically, the API is used in profilers, for example the Profiler [8]. The profiling API is implemented as two COM interfaces. One is implemented by the Runtime (`ICorProfilerInfo`), the other is implemented by the profiler (`ICorProfilerCallback`).

The `ICorProfilerCallback` interface must be implemented by the profiler DLL. The interface methods are called by CLR to notify the profiler of events in the profiled process. The methods in the `ICorProfilerInfo` interface can be used by the profiler to gather information about profiled application.

To create heap dumps with the profiling API, the garbage collection functionality of the `ICorProfilerCallback` interface is used. The `COR_PRF_MONITOR_GC` flag must be set in the `initialize` method of the interface. This has a drawback: after setting this flag, the concurrent garbage collection is turned off. Creating a memory dump is quite similar to a garbage collector run. To create a memory dump the garbage collector must be started. If this flag is set the `ObjectReferences` method of the `ICorProfilerCallback` interface is called by the CLR for each object or array seen by the garbage collector. The `ObjectReferences` methods are called by each garbage collector run. If no memory dump is issued the method can return constant `E_ABORT` to abort the execution of the method for this garbage collector run.

```
HRESULT ObjectReferences(ObjectID objectId,
     ClassID classId, ULONG cObjectRefs,
     ObjectID objectRefIds[])
```

The .Net heap dump algorithm uses the `GetObjectSize` method of the `ICorProfiler-Info` interface to get the size of the considered object.

```
HRESULT GetObjectSize(ObjectID objectId,
     ULONG* pcSize);
```

Then the algorithm checks if the `classId` identifies a real class or the class of an array. It is done via the `IsArrayClass` method of the `ICorProfilerInfo` interface. `IsArrayClass` returns `S_OK` if the given classId is the id of an array class or `S_FALSE` if the classId is the id of a class. The parameter `pBaseClassId` of the method returns the classId of the next lower dimension of the array.

```
HRESULT IsArrayClass(ClassID classId,
     CorElementType* pBaseElemType,
     ClassID* pBaseClassId,
     ULONG* pcRank);
```

With the output parameter `pcRank` it is possible to test if the array is jagged or multi dimensional. The elements of multi dimensional arrays are stored one after the other in a continuous memory block. A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. Sometimes it is called an „array of arrays “. If the `pcRank` parameter returns 1, the considered object is a multi dimensional array. Otherwise the parameter returns the dimension of the array, since the object is a jagged array.

The classname can be collected with the `GetTypeDefProps` method of the IMeta-DataImport Interface. The parameter `szTypeDef` contains a pointer to a wide char array with fully qualified class name in it. This method needs as an input parameter the type def token (`mdTypeDef td`) of the class.

```
HRESULT GetTypeDefProps (mdTypeDef td,
     LPWSTR szTypeDef, ULONG cchTypeDef,
     ULONG* pchTypeDef, DWORD* pdwTypeDefFlags,
     mdToken* ptkExtends);
```

To call this method, a pointer to a `IMetaDataImport` struct must be acquired first by calling the `GetModuleMetaData` function of the `ICorProfilerInfo` interface, where `REFIID riid` is the wanted pointer. This function needs as an input parameter the `moduleId` of the module in which the class is contained.

```
HRESULT GetModuleMetaData(ModuleID moduleId,
     DWORD dwOpenFlags, REFIID riid,
     IUnknown** ppOut);
```

The moduleIds and the typedef tokens are collected with the `GetClassIDInfo` method of the `ICorProfilerInfo` interface. The inspected input parameter of this method is the `classId` of the considered class.

```
HRESULT GetClassIDInfo(ClassID classId,
    ModuleID* pModuleId,
    mdTypeDef* pTypeDefToken);
```

With this information and the references form the `objectRefIds[]` parameter of the `ObjectReferences` method, it is possible to send all information for the extended heap dump to the server. Because it is impossible for the .Net profiling API to acquire all loaded classes, the extended heap dump must store the classIDs sent to the server in a hash map. Each class is only sent once. The .Net extended heap dump has a drawback: It is impossible to collect static references because only field references are reported by the `ObjectReferences` method.

The .Net simple heap dump is similar to the extended dump. The implementation only differs at one point: By the extended heap dump sends all data to the server. The simple heap dump stores the data in a hash set and sends the collected data to the server if all data is collected. The `RuntimeResumeFinished` callback can be used to check if the heap dump is finished.

```
HRESULT RuntimeResumeFinished()
```

The heap dump is part of a garbage collector run. The gc suspends the virtual machine before the run starts and resumes it if the run is finished. The collected data from the simple heap dump is sent to the server in the `RuntimeResumeFinished` callback.

More about the .Net Profiling API can be found at [10].

Appendix C

Implementation

The practical part of the master thesis was the implementation and integration of the selected algorithms. The implemented classes and the structure between them are discussed in Appendix C. The heap dump algorithms are presented in Chap. 3 and the analysis algorithms are presented in Chap. 4. The first section in this chapter discusses the algorithms to create heap dumps and the second section discusses the heap dump analysis algorithms.

C.1 Heap Dump Algorithms

Fig. C.1 shows the structure of the classes necessary for the heap dump algorithms.

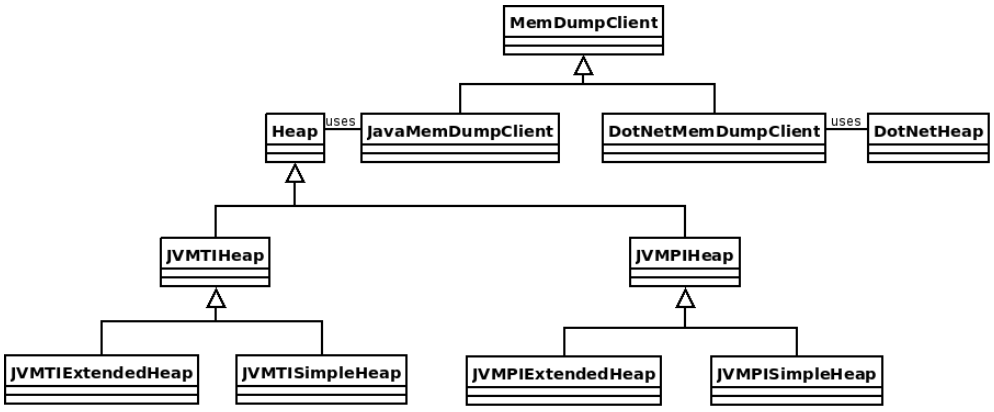

Figure C.1: Heap Dump Overview Class Diagram

The class diagram of `MemDumpClient` can be seen in Fig. C.2. This class manages the communication between the dynaTrace server and the dynaTrace agent. There are only minor differences between `JavaMemDumpClient` and `DotNetMemDumpClient`.

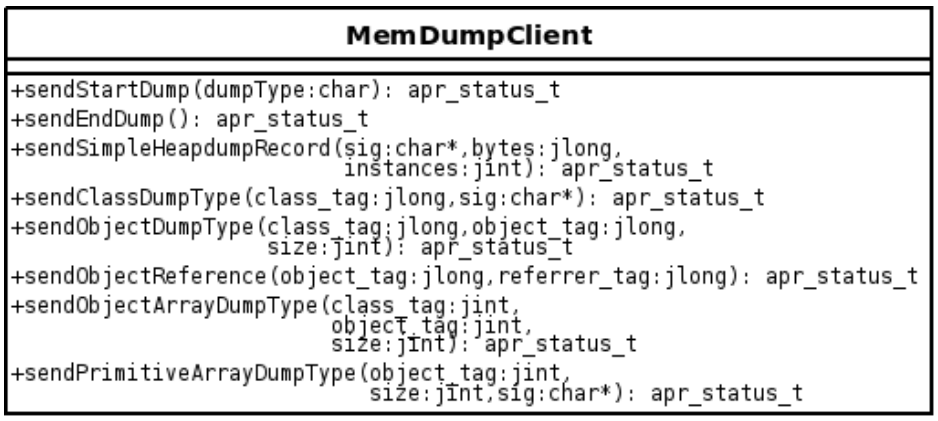

Figure C.2: MemDumpClient Class

The `Heap` class contains methods to manage the heap dump channel and to send the *HeapDumpStart* and the *HeapDumpEnd* event. The `JVMPIHeap` and `JVMTIHeap` classes are implemented to manage the `JVMPI_Interface` or the `jvmtiEnv` environment pointer. All of the following heap dump classes contain a `HeapDump()` method. Calling this method is required to initiate a heap dump.

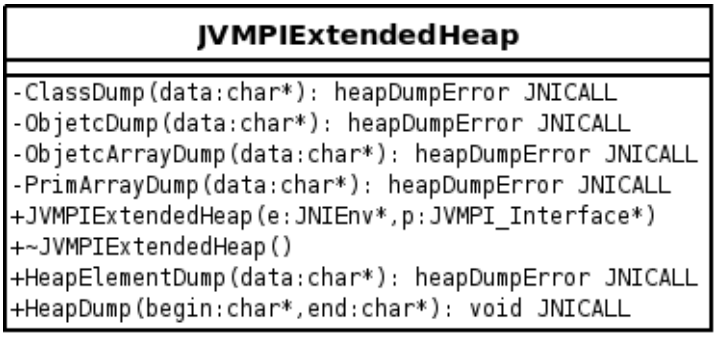

Figure C.3: JVMPIExtendedHeap Class

Fig. C.3 shows the class diagram for the extended heap dump implemented using JVMPI. The method `HeapElementDump()` is called for each `JVMPI_EVENT_OBJECT_DUMP` response. The method parses the record type field and calls `ClassDump()` for class records, `ObjectDump()` for object records, `ObjectArrayDump()` for object array records, and `PrimArrayDump()` for primitive array records. The methods parses the records and

collects information required for the class, objects or refference records. For details about the algorithm see Chap. 3.

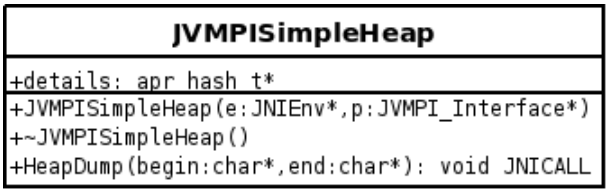

Figure C.4: JVMPISimpleHeap Class

The class diagram of the JVMPI simple heap dump can be seen in Fig. C.4. The algorithm is presented in Chap. 3.

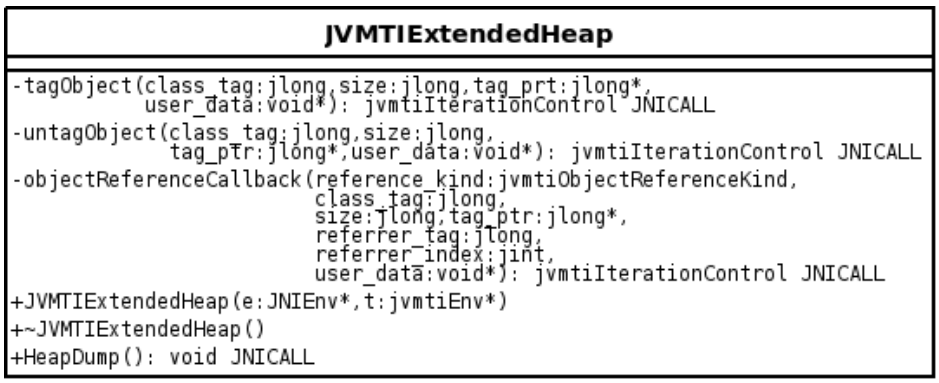

Figure C.5: JVMTIExtendedHeap Class

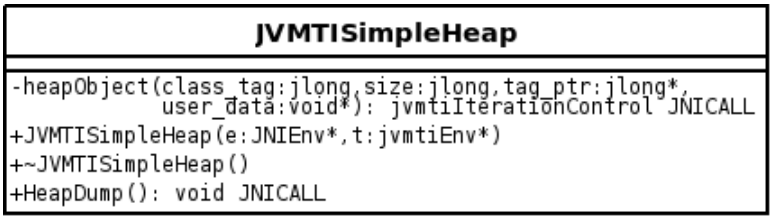

Figure C.6: JVMTISimpleHeap Class

Fig. C.5 and Fig. C.6 present the class diagram for the simple and the extended heap dump implemented with JVMTI. Other methods and callbacks are explained in Chap. 3.

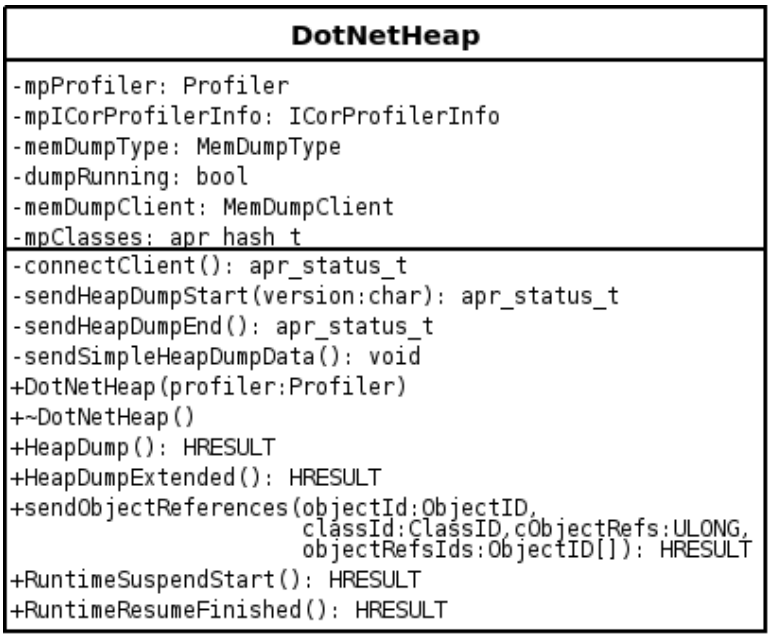

Figure C.7: DotNetHeap Class

`DotNetHeap` implements the simple and the extended heap dump algorithm for .NET. The class diagram is presented in Fig. C.7. The method `connectClient()` is used to establish the heap dump channel between agent and server. The methods `sendHeapDumpStart()` and `sendHeapDumpEnd()` are used to send *HeapDumpStart* and *HeapDumpEnd* events to the server. The .NET heap dump algorithms depends on a garbage collector run. To determine the start and the end of the gc run the CLR callbacks `RuntimeSuspendStart()` and `RuntimeResumeFinished()` are used. Another callback used is the `sendObjectReferences()`. For each object in the CLR this method is called. The callback is used to collect information for the simple and the extended heap dump. If enough information for the simple heap dump is collected `sendSimpleHeapDumpData()` method transfers it to the server. Detailed information about the algorithm can be found in Chap. 3.

C.2 Reference Graph Analysis Algorithms

This section discusses the heap dump generator algorithm, the delta debugger and the actual heap dump analysis algorithms.

C.2.1 Heap Dump Generators

Four different hep dump generators were implemented in its own classes. All the heap dump generators are JUnit tests, extending the `AbstractGenerator` class.

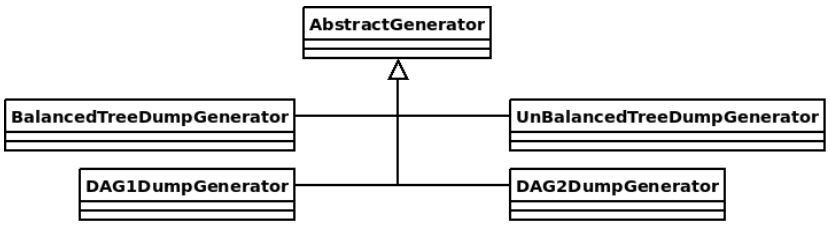

Figure C.8: Heap Dump Reference Graph Generators

The `AbstractGenerator` class contains all methods necessary for the JUnit test. `setUp()` constructs the `memDumpWriter` object and `tearDown()` destroys it. The `testGenerator()` method is the JUnit test method. The main part of the `testGenerator()` method is the call of the abstract `generate()` method.

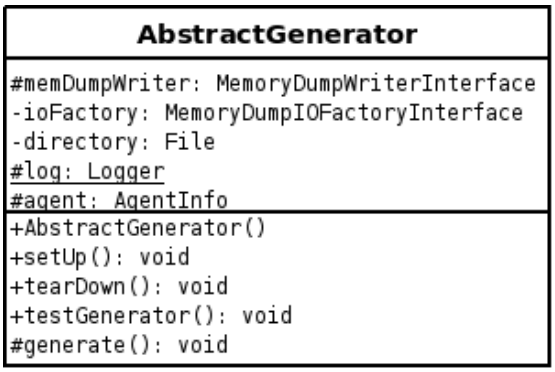

Figure C.9: AbstractGenerator Class

All extended classes must implement the `testGenerator()` method. All heap dump generator classes use the `memDumpWriter` object to create the classes, objects and references of the heap dump.

C.2.2 Delta Debugger

The delta debugger was implemented to find defects in the heap dump analysis algorithms, by reducing the references in a given heap dump to a minimum. The `ddmin` algorithm from [17] was implemented as JUnit tests.

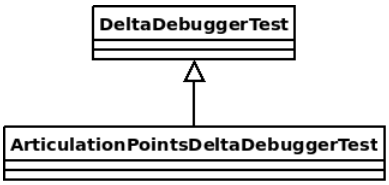

Figure C.10: Reference Graph Delta Debugger

The `ddmin` algorithm was implemented in the `DeltaDebuggerTest` class. The `ddmin` algorithm produces reduced heap dumps from the original heap dump. For each of the reduced dumps it is tested whether the defect still occurs, by calling `Test()`. The method is abstract and must be implemented by the class that implements the debugger for a specified defect. The `ArticulationPointsDeltaDebuggerTest` class was implemented to find the defect in the dynamic size algorithm that uses articulation points.

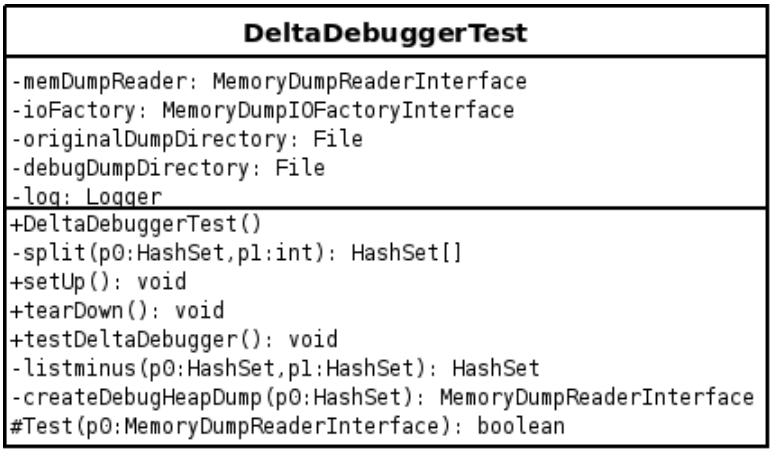

Figure C.11: DeltaDebuggerTest Class

The methods `split()` and `listminus` belongs to the `ddmin` algorithm and can be found in [17]. `createDebugHeapDump()` produces reduced heap dumps from the original one.

C.2.3 Analysis Algorithms

Analysis algorithms are implemented as JUnit tests. To run them some helper classes are necessary. Fig. C.12 show the `AbstractTest` class with all helper classes.

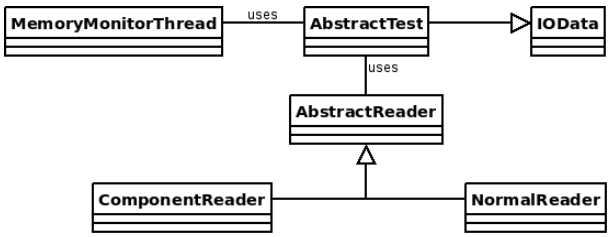

Figure C.12: Reference Graph Analysis Algorithms Helper Classes

The `AbstractTest` class is the super class for all heap dump analysis algorithms implemented. The class implements all methods, necessary for the JUnit test and manages the `memDumpReader` interface to the heap dump. The analysis algorithms implement the `test()` method.

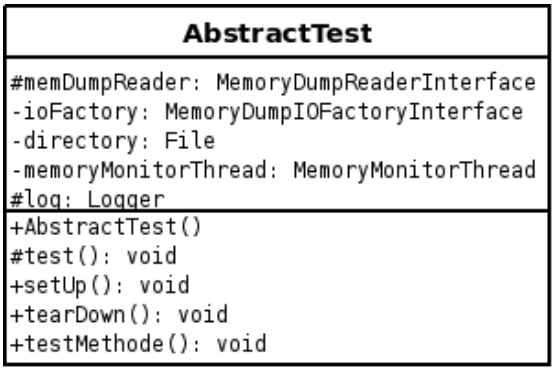

Figure C.13: AbstractTest Class

To monitor memory consumption of the analysis algorithm `MemoryMonitorThread` is used. The thread determines the free-, the used- and the maximum memory consumption of the analysis algorithm every five seconds. The values collected are stored in a comma separated values file. The thread controlled by `AbstractTest`.

The `IOData` class is the super class of the `AbstractTest` class. It extends the analysis algorithms with two additional methods to read and to store integer arrays from files.

This is necessary because most of the analysis algorithms stores their results in integer arrays.

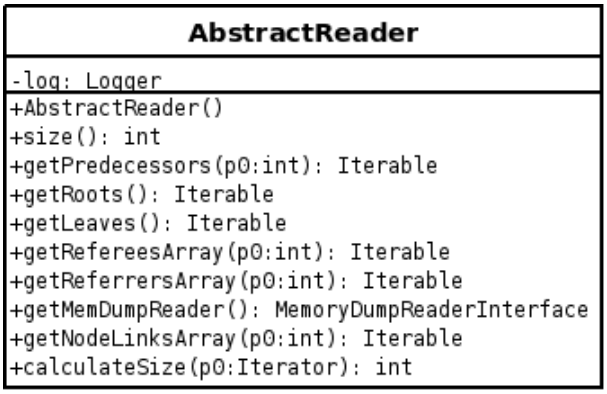

Figure C.14: AbstractReader Class

All analysis algorithms use the `NormalReader` or the `ComponentReader` class. The classes implement the `AbstractReader` interface, see Fig. C.14. `size()` is used to get the number of nodes in the graph. `NormalReder` returns the number of objects and classes, `ComponentReader` the number of SCCs. The iterators return a `java.lang.Iterable` object. The iterable objects store references to a `java.util.Iterator` object. Several iterator classes were implemented. The `next()` functions return the ID of the next node, independent from whether the node is a class or an object. The `ArrayIterator` class is necessary for `getRoots()` and `getLeaves()` functions. The iterator object iterates over the roots or leaves array. The `getRefereesArray()` and `getReferrersArray()` use the `NodeIterator` of the `NormalReder` or `ComponentReader`. The `getNodeLinksArray()` is used in the articulation points algorithm. It returns an object of the type `NodeLinkIterator`. The iterator object iterates over all incoming and outgoing references of a given object. This is necessary because the articulation points algorithm works on undirected graphs only.

All heap dump analysis algorithms extend `AbstractTest`. Fig. C.15 shows all implemented analysis algorithms and the corresponding JUnit tests. The analysis algorithms can be used by more than one JUnit test.

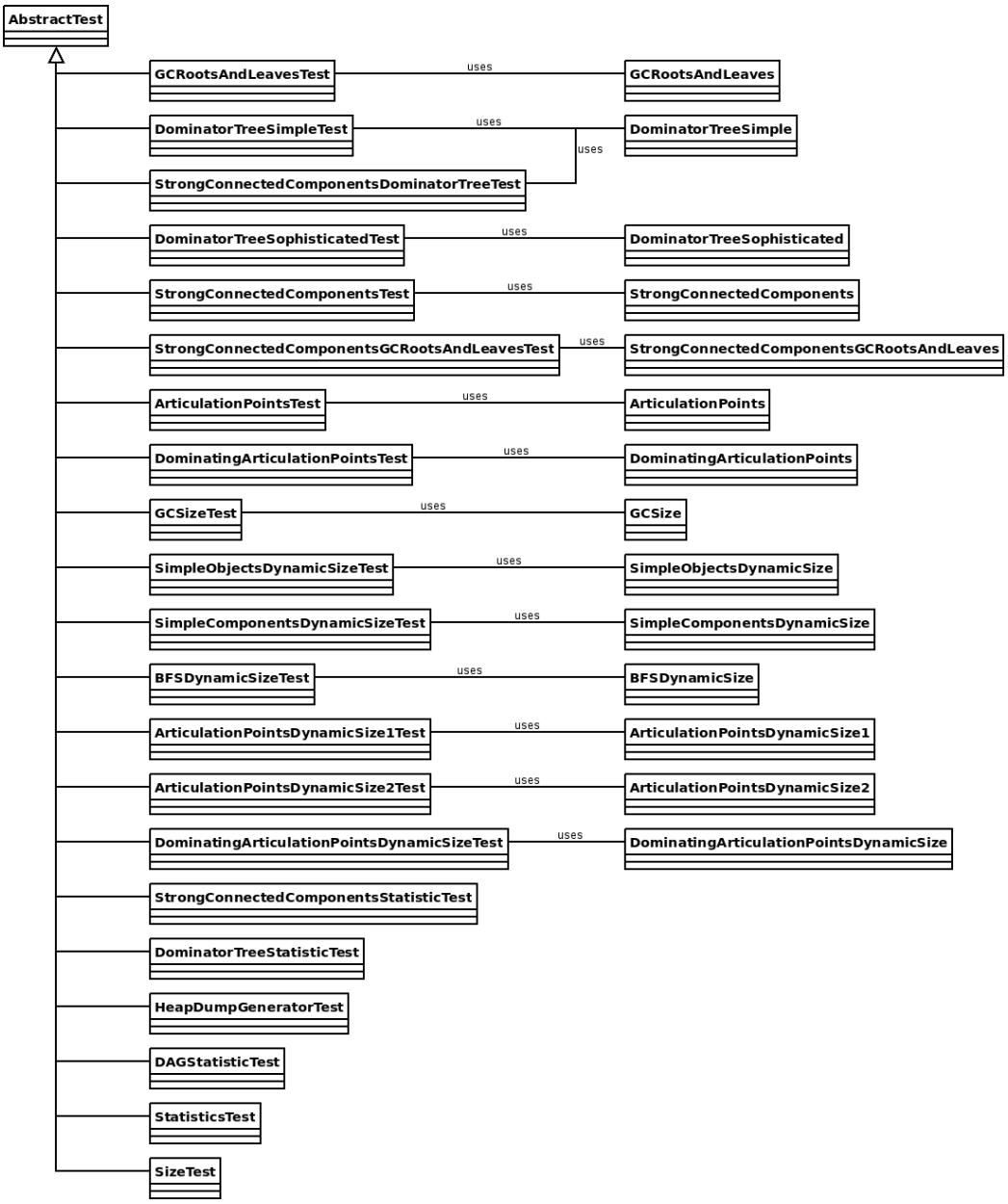

Figure C.15: Reference Graph Analysis Algorithms Overview

The JUnit test loads the requirements from the files in the heap dump directory, constructs the analysis algorithm instance, runs the algorithm and stores the results to the file system. An exception to this are the statistic and the size tests. They do not use any of the implemented analysis algorithms. The statistic test reads the needed data from the files in the heap dump directory and writes the results to the log file. The `SizeTest` reads all computed sizes from the filesystem and compares the results.

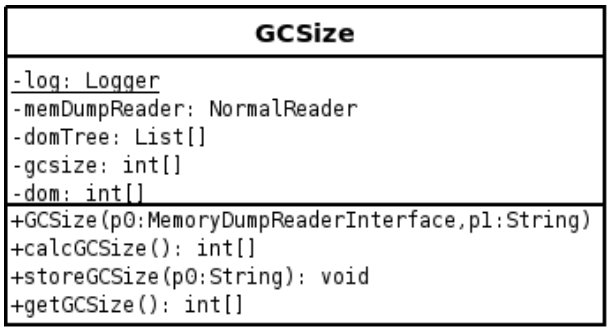

Figure C.16: GCSize Class

Fig. C.16 shows the class diagram of the GCSize algorithm. The calculated dominators are loaded to the `dom` array. The value at each array position represents the dominator of the node. This is not useful for the depth first traversing of the dominator tree. To traverse the dominator tree the `domList` array is computed. For each node the list of all direct dominated nodes is calculated and stored in the array at the position of the index of the ID of the node. To optimize the algorithm `domList` array can be computed only once by storing it into the heap dump directory. The `gcsize` array stores all the size values computed.

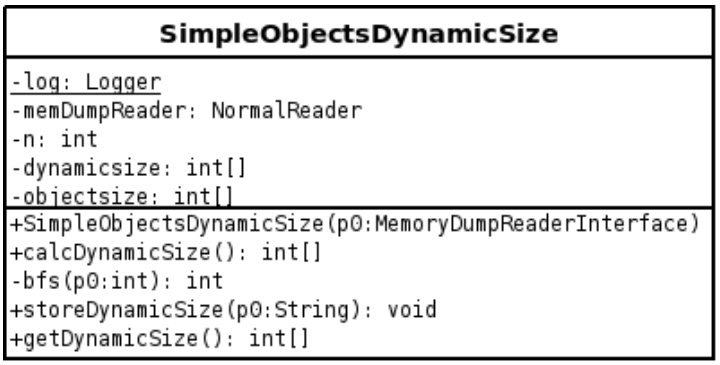

Figure C.17: SimpleObjectsDynamicSize Class

The class diagram of the `SimpleObjectsDynamicSize` class is displayed in Fig. C.17. The array `dynamicsize` stores the dynamic sizes calculated and the array `objectsize` contains the static size of the nodes. The array `objectsize` is used to reduce the IO of the algorithm. To calculate dynamic size for each node the graph is traversed in breadth-first-order, because the breadth-first algorithm is implemented iteratively. All other dynamic size algorithm classes are similar to the `SimpleObjectsDynamicSize` class.

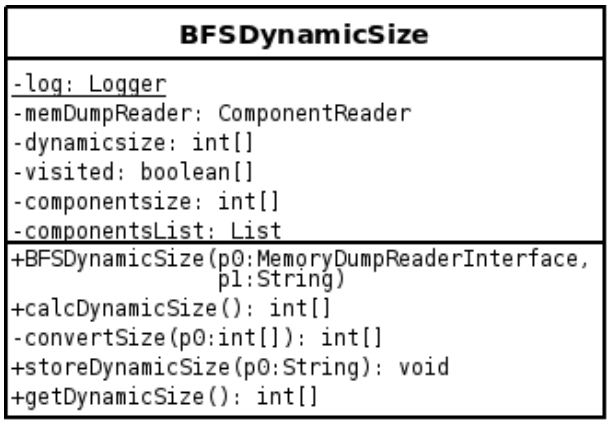

Figure C.18: BFSDynamicSize Class

Fig. C.18 shows the class diagram of the implemented `BFSDynamicSize` algorithm. For more about this algorithm see Chap. 4. The `BFSDynamicSize` implements, like all classes connected to the SCC graph, `convertSize()`. It computes the dynamic size values of the heap dump reference graph nodes from the dynamic size values of the SCC nodes.

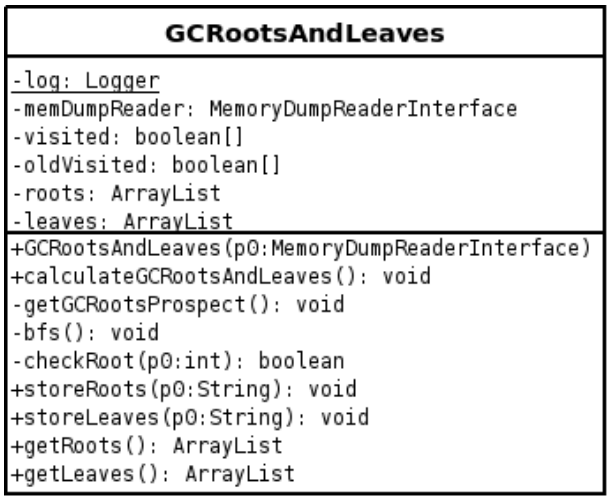

Figure C.19: GCRootsAndLeaves Class

The class diagram of `GCRootsAndLeaves` is shown in Fig. C.19. `getRootsProspect()` adds all nodes without referrer nodes to the `roots ArrayList`. `bfs()` is called for each of the nodes in the `roots` list. It marks all reachable nodes in the array `visited`. Afterwards, for each node not marked, `checkRoot()` is called. It checks if the node is part of a cycle. If so, it adds the node with the lowest ID in the cycle to the `roots` list.

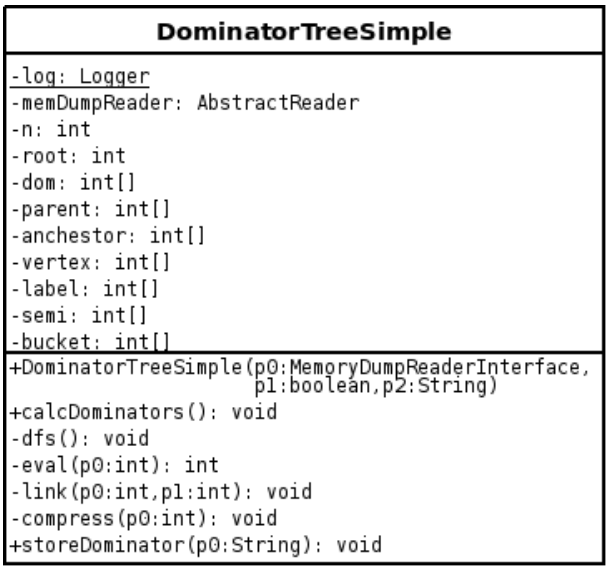

Figure C.20: DominatorTreeSimple Class

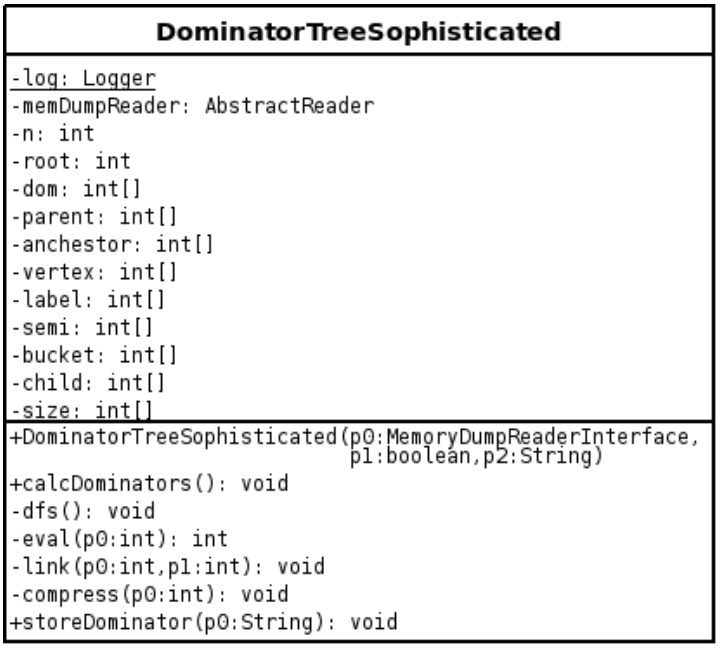

Figure C.21: DominatorTreeSophisticated Class

The class diagrams of `DominatorTreeSimple` and `DominatorTreeSophisticated` can be seen in Fig. C.20 and C.21. Detailed information about the fields and methods can be found in [5].

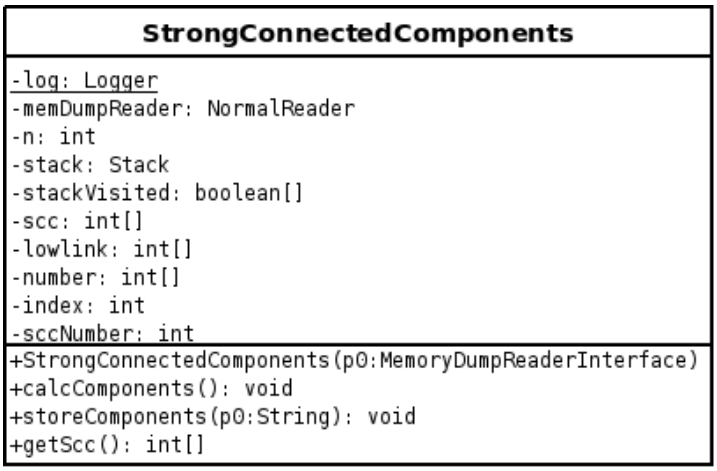

Figure C.22: StrongConnectedComponents Class

Fig. C.22 shows the class diagram of the `StrongConnectedComponents` algorithm. Detailed information about the fields and methods can be found in [16].

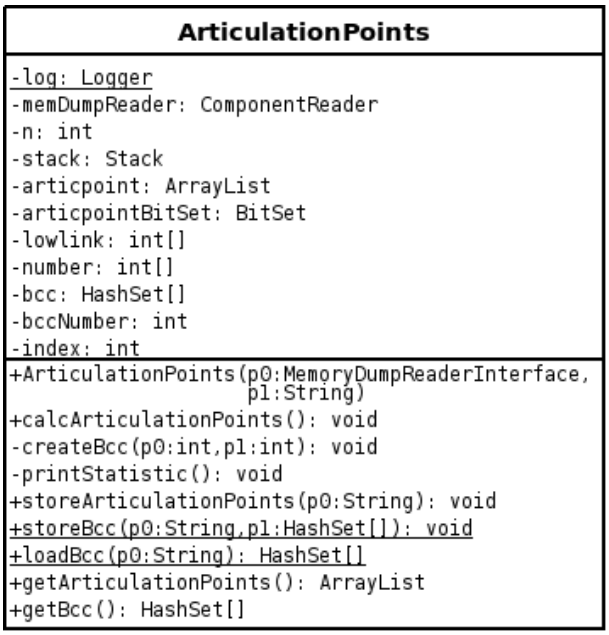

Figure C.23: ArticulationPoints Class

The class diagram of `ArticulationPoints` can be seen in Fig. C.23. Detailed information about the fields and methods can be found in [16].

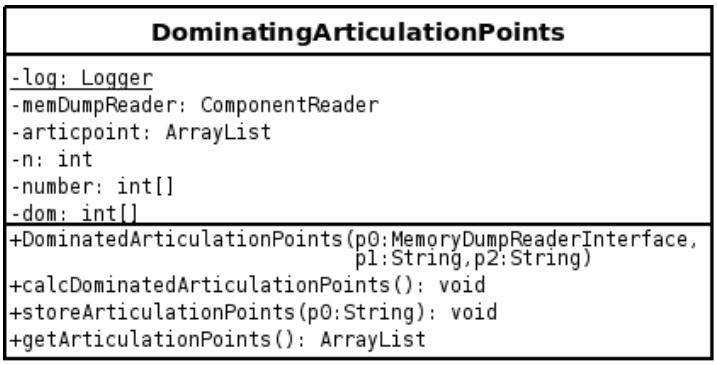

Figure C.24: DominatingArticulationPoints Class

Fig. C.24 shows the class diagram of the `DominatingArticulationPoints` algorithm. The dominators computed for the SCC nodes are loaded to the `dom` field. The algorithm is explained in Chap. 4.

Bibliography

[1] dynaTrace software GmbH. http://www.dynatrace.com/.

[2] Google. google-perftools fast, multi-threaded malloc() and nifty performance analysis tools. http://code.google.com/p/google-perftools/.

[3] Jonathan L. Gross and Jay Yellen. *Graph Theory and Its Applications, Second Edition (Discrete Mathematics and Its Applications).* Chapman & Hall/CRC, 2005.

[4] Steven Haines. *Pro Java EE 5 Performance Management and Optimization.* Apress, 2006.

[5] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[6] Eclipse Memory Analyzer (MAT). http://www.eclipse.org/mat/.

[7] Brendan D. McKay. nauty. http://cs.anu.edu.au/ bdm/nauty/.

[8] Microsoft. Clr profiler. http://msdn.microsoft.com/en-us/library/ms979205.aspx.

[9] Sun Microsyystems. Memory management in the java hotspot virtual machine. http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf.

[10] Microsoft Developer Network (MSDN). Profiling (unmanaged api reference). http://msdn.microsoft.com/en-us/library/ms404386.aspx.

[11] Sun Developer Network. Jvmpi java virtual machine profiler interface. http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html.

[12] Sun Developer Network. Jvmti tool interface. http://java.sun.com/j2se/1.5.0/-docs/guide/jvmti/jvmti.html.

[13] Kelly O'Hair. Hprof: A heap/cpu profiling tool in j2se 5.0. http://java.sun.com/-developer/technicalArticles/Programming/HPROF.html.

[14] Jeffrey Richter. Garbage collection: Automatic memory management in the microsoft .net framework. http://msdn.microsoft.com/msdnmag/issues/1100/gci/.

[15] Jack Shirazi. *Java Performance Tuning.* O'REILLY, 2003.

[16] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[17] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, October 2005.

Lebenslauf

Stefan Riha

Adresse

Hauptstraße 50\2\21
4040 Linz Österreich
Email: stefan.riha@liwest.at

Persönliche Daten

Geburtsdatum: 1981-05-07
Geburtsort: Steyr, Österreich
Staatsbürgerschaft: Österreich

Ausbildung

09/1988–07/1992	Volksschule Steyr Ennsleite
09/1992–07/1996	Hauptschule Steyr Münichholz
09/1996–07/2001	HTL Steyr Abteilung für Maschinenbau
25.6.2001	Matura HTL Steyr
10/2001–10/2002	Zivildienst Alten und Pflegeheim Steyr
10/2002–11/2009	Studium der Informatik an der JKU Linz

Dienstverhältnisse

07/2001–09/2001	Technischer Zeichner GFM Steyr
03/2004–12/2004	Programmierer bei BM-IT
08/2005–05/2009	Programmierer bei dynaTrace software

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am October 28, 2009

Stefan Riha