DISSERTATIONES
MATHEMATICAE
UNIVERSITATIS
TARTUENSIS
91

VLADIMIR ŠOR

Statistical approach for memory leak detection in Java applications

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

91

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
91

VLADIMIR ŠOR

Statistical approach for memory leak detection in Java applications

UNIVERSITY OF TARTU
PRESS
1632

Institute of Computer Science, Faculty of Mathematics and Computer Science, University of Tartu, Estonia.

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (Ph.D.) in informatics on September 19, 2014, by the Council of the Faculty of Mathematics and Computer Science, University of Tartu.

Supervisor:

Assoc. Prof., Dr.	Satish Narayana Srirama Institute of Computer Science University of Tartu, Tartu, Estonia

Opponents:

Prof. Dr.	Andreas Krall Faculty of Informatics Vienna University of Technology Austria
Dr.	Walter Binder Faculty of Informatics University of Lugano Switzerland

Commencement will take place on November 17, 2014, at 16.15 in J. Liivi 2-403.

The publication of this dissertation was financed by Institute of Computer Science, University of Tartu and by the Estonian Doctoral School in Information and Communication Technology.

European Union
European Social Fund

Investing in your future

ISSN 1024–4212
ISBN 978-9949-32-679-2 (print)
ISBN 978-9949-32-680-8 (pdf)

University of Tartu Press
www.tyk.ee

Contents

Abstract

Managed run-time systems with garbage collection, like Java Virtual Machines, have proved themselves as boosters for developer productivity and have removed the burden of manual memory management by using automatic garbage collection. There used to be two major potential sources of bugs associated with manual memory management: dangling pointers and memory leaks. While garbage collection completely solves the problem of dangling pointers, the problem of memory leaks is solved only partially, as garbage collector cannot reclaim objects which are still referenced while being unused. Such memory leaks pose a major problem for long-running processes with finite amount of heap, as finally the heap will become exhausted and the program will inevitably crash.

Prominent research for the solution against memory leaks in Java applications was mostly conducted before Java Virtual Machines received powerful programming interfaces for monitoring and instrumentation and often rely on modification of the virtual machine and garbage collector.

Current thesis classifies existing approaches for memory leak detection, both automatic and manual, and proposes a novel lightweight approach for automatic memory leak detection, utilizing monitoring capabilities and programming interfaces of modern Java Virtual Machines. Instead of using staleness indicators for particular class instances, the approach considers statistical parameters describing evolution of objects in the heap to find outliers in terms of object lifetimes grouped by their allocation site. Thesis

defines and analyzes such a major indicator as the number of surviving generations for objects created at one allocation site. Thesis demonstrates that this indicator can be effectively used to identify memory leaks.

Shortcomings of using only the number of surviving generations for memory leak are also analyzed. Analysis is based on the data which was acquired from hundreds of different applications, where initial implementation of the method was successfully deployed. As a result of the analysis new supportive statistical metrics are identified and used to further enhance the method using machine learning to decrease the number of both false positives and false negatives.

Case studies of the method were conducted with several real-world and application frameworks and are described in the thesis in order to observe detection quality of the method along with the analysis of the performance overhead which is added by the implementation.

Acknowledgments

Parts of my doctoral studies have been supported by the European Regional Development Fund through the Estonian Center of Excellence (EXCS), the Software Technology and Applications Competence Center (STACC), and by the European Social Fund through the Estonian Doctoral School in Information and Communication Technology (IKTDK).

I thank my supervisor Dr. Satish Srirama, who was persistent and guiding enough for me to be able to finish my thesis. I also would like to thank Prof. Marlon Dumas, discussions with whom have enlightened me about how science is done.

I would like to specially thank Ivo Mägi for giving me the idea for the thesis and the opportunity to work on it while being employed at Webmedia. Thanks to Nikita Salnikov-Tarnovski for helping me solving most complex engineering challenges while developing Plumbr in the early days and discussions with whom were setting the vision and milestones for Plumbr over the years. Big thanks to the whole amazing team of Plumbr.

Thanks to Tarvo Treier and Riivo Kikkas for showing how machine learning works and how data analysis should be done.

Special thanks goes to my family, especially my wife Kairit who was patient and supportive during all the years spent at doctoral studies and especially during last months of intense work. The last but not least thanks go to my grandmother Raissa who was extremely persistent with queries about the progress of the studies.

List of Figures

List of Tables

Chapter 1

Introduction

Managed languages and run-time systems offer several benefits for the developer by abstracting away operating systems and hardware architectures. More high-level services are provided by the virtual machine, thus requiring less low-level code to be written by a developer. Run-time system verifies the code while loading, and executes it in a security sandbox. Deployment cost and complexity is lower, because the same code can be executed on all platforms where run-time system is supported. All this leads to a better productivity for the application developer.

Both managed and un-managed modern programming languages rely on dynamic memory allocation. This allows allocation and de-allocation of memory on demand, not knowing required amounts of memory upfront during compilation. Dynamic allocation occurs in a heap, rather than on the stack during execution or statically during compilation. Memory which is allocated in the heap is accessed through a *reference*. Usually a reference is a pointer to a memory address where allocated memory resides. Heap allocation allows to dynamically choose allocation size, use recursive data structures like maps or trees. Use of references allows returning newly created objects from methods, passing references to other methods, and sharing references between methods.

If memory which is allocated on the stack or statically is freed by the operating system when the application exits, then memory allocated dynamically in the heap has to be de-allocated by the application. This can be performed explicitly by the application or automatically by the run-time system. Explicit de-allocation occurs by calling C's `free` function or C++'s `delete` operator. Run-time systems use garbage collection to reclaim unused memory. Manual de-allocation may result it two types of problems.

First, memory may be freed while there are still references to it. Such a reference is called a *dangling pointer*. If the program tries to access the memory pointed to by the danging pointer the result is unpredictable, as it is unknown what the run-time system will do with de-allocated memory. There are two possible outcomes of this situation: immediate crash or incorrect results. Immediate crash being the best outcome, as incorrect results and probable later crash are much more hard to debug.

Second, the programmer may forget to free an object which is no longer used by the application. This leads to a *memory leak*. While in small applications memory leaks may pose no noticeable problems, in large long-running applications memory leaks may lead to a severe performance degradation or even a crash.

Concurrent programming and shared state further amplify these problems. Liveness of an object becomes a global property, but the decision to deallocate that object still remains local. Solutions proposed over time range from not using heap allocation when possible, to pass and return objects by value, rather than by reference, to use custom allocators to manage a pool of objects [Jone 11].

1.1 Problem statement

Automatic dynamic memory management resolves most of issues created by the manual memory management. *Garbage collection* (GC) prevents dangling pointers, as an object can be reclaimed if there is no reference to it from a reachable object. And also memory leaks in previously defined form cannot occur, as unreachable objects will be removed by the garbage collector. Reclamation decisions are left to the central garbage collector, which has knowledge of all objects on the heap and knows which threads may access them, so it also solves the problem of global liveness and local deallocation.

However, memory leak problem is not solved completely by the garbage collection. Although all objects which are not not accessible for the program are guaranteed to be reclaimed, garbage collection cannot guarantee the absence of space leaks. Objects and data structures, which are reachable, but are growing without limit, or just never accessed again by the program, are not reclaimed and are wasting heap space [Jone 11].

Current thesis focuses on solving the remaining part of memory leak problem for applications running in a Java Virtual Machine (JVM) – a widely used garbage collected run-time environment. According to Tiobe Programming Community Index [Tiob 14], an indicator of the popularity of programming languages, since 2001 by 2014 Java was the most popular programming language for 10 years out of 13. Java progressed not only as a language but most importantly as a cross-platform runtime environment with garbage collection. In recent years we can observe an increase in popularity of languages such as Scala, Clojure, Groovy, etc. which all utilize Java Virtual Machine and its bytecode to create platform-independent applications.

A condition in the JVM when a new object cannot be allocated in the heap, because there is not enough space to accommodate it, ends up by JVM throwing `java.lang.OutOfMemoryError`. There may be two reasons leading to such outcome. There might be plenty of free heap, but the program may try to fit too much data into heap at once because of a programming error, poor choice of algorithms, poor implementation of an algorithm or just the amount of data has grown over time, so that a database query started returning too much data. In such case either changing the algorithm so that it would not load so much data at once or just increasing heap size can help. The good part of this situation is that the source of the problem is immediately visible in the stack trace – method which was trying to allocate too much memory at once is on the top of the call stack. Another reason for such outcome is the memory leak in terms defined earlier – heap filling with objects which are not used anymore, but cannot be collected by the garbage collector because of a forgotten reference.

In case of a memory leak the heap is depleted gradually by unused objects until there is no more space for any random part of the program to continue. This means several bad things. The biggest problems is that `OutOfMemoryError` is thrown in a random piece of code in the program which has nothing to do with the actual source of the problem. The memory leak may grow slowly or be caused by some specific use case, which makes it hard to debug and reproduce anywhere outside of a production environment. When the heap utilization reaches its limits garbage collector has hard times trying to free any memory required for the program, which means that the garbage collector spends most of the applications' CPU time, which seems like application becomes unresponsive, or just hangs.

The problem of detecting memory leaks in Java applications was addressed before, however there's still plenty room for improvement. Probably the most popular way to find sources of memory leaks today is still a manual heap dump investigation. Although sometimes effective and simple, there are many scenarios where it falls short. In large applications, with several gigabytes of normally used heap it may be difficult to separate leaking objects from non-leaking ones. In addition, large heap dumps may consume a lot of resources to be analyzed. Acquisition of the heap dump must be triggered at the right time or several heap dumps are required to see the evolution of the heap contents. Heap dump also misses any temporal information and sources of the leaked objects. All these difficulties are amplified by the human factor, as certain expertise is required to analyze heap dumps.

Specialized, much less adopted, approaches targeted specifically for memory leak detection also have limitations. For example, because of the automatic memory management, reachability graph is so non-deterministic that it makes static analysis on a general level still unfeasible and costly to perform, which limits its application to just a subset of specific scenarios which are known to be causing the leak [Shah 00, Dist 10].

Efficient runtime analysis techniques which account for actual object usage were proposed before efficient JVM tooling for monitoring and bytecode instrumentation made their way to production JVMs. This led to solutions modifying the internals of JVMs and garbage collectors, which are too critical components for such supportive utility functionality.

Current thesis describes the technique for memory leak detection in Java applications which account for temporal information, creation sites of objects and general application behavior while observing statistical metrics of the application which can be obtained using standard monitoring capabilities of modern Java virtual machines without modifying the code of a garbage collector.

1.2 Contributions of the thesis

The main contributions of the current thesis are following.

- Classification of existing memory leak detection approaches is created, separating leak detection methods into online, offline and hybrid methods with following sub-classification. Metrics observed by the approaches and methods used for their quality assessment were compared and summarized. A notion of intrusiveness is introduced and existing methods are compared against this new metric.

- A formal definition of the *genCount* metric for java applications is given, and algorithm for memory leak detection, using *genCount* metric grouped by allocation sites, is described.

- The proposed algorithm was implemented in the leak detection tool called Plumbr. The tool was implemented as a Java agent using standard Java programming interfaces which facilitated ease of deployment and performance of the tool allowed its usage in many production environments. The tool was made available for public use and was used to collect the statistical snapshots of real applications running in real production environments, thus collected statistics are not synthetic and reflect real use of real applications.

- An infrastructure for gathering statistical information about allocation behaviour from thousands of applications was created and deployed. The collected data was used to verify the applicability of initial hypothesis and analyze its shortcomings.

- As a result of the analysis of collected statistical data about Java applications, new metrics were designed and machine learning was applied to improve the detection quality of the initial implementation.

- Plumbr was evaluated on real applications and frameworks and its detection quality was compared with the existing state of the art approach, which was using Java Virtual Machine modifications to measure object staleness and it was shown that statistical approach for memory leak detection performs better, especially considering its low intrusiveness.

1.3 Outline

Chapter 2 discusses state of the art for the research addressed by the thesis. State of the art includes garbage collection and machine learning. Terminology related to automatic memory management and garbage collection is followed by the description of four basic algorithms for garbage collection. Garbage collection algorithms used in the HotSpot Java Virtual Machine along with description of the heap layout are described. Garbage collection algorithms used in competitive virtual machines are described briefly.

Basic machine learning concepts, performance evaluation metrics along with short description of further used classification algorithms are presented. Section covering machine learning describes basic details required to comprehend practical application of classification algorithms required for the thesis.

Chapter 3 continues state of the art chapter by reviewing the preliminary work related to the memory leak detection in Java applications. Existing approaches are classified from the point of view of assessed metrics, performance overhead and intrusiveness. In addition, the methods are classified into online, offline and hybrid groups based on their features. Classification of the existing research outlines areas which are to be addressed and improved by the thesis.

Classification of the memory leak detection approaches is previously published in [vSor 14a].

Chapter 4 introduces statistical approach for an automatic memory leak detection in Java applications. Chapter describes how weak generational hypothesis which works for generational garbage collectors can be used to detect objects which do not conform to the hypothesis. An important concept of number of survived allocations, of *genCount* is presented along with the analysis of its strengths and weaknesses. The approach was implemented in the commercial memory leak detection tool Plumbr and its implementation details are discussed in the chapter including the analysis of the runtime performance overhead using DaCapo benchmarks. Performance of the leak detection is assessed and areas requiring further attention are identified.

Description of the statistical approach and the analysis were previously published in [vSor 11a],[vSor 14b].

Chapter 5 describes how machine learning was used to improve the baseline leak detection quality set by the implementation described in Chapter 4. First, the data sets used for learning and validation are described. It is followed by the description of the design process conducted to identify new attributes which should be used for learning. In addition to the dominating *genCount* attribute, 5 additional statistical attributes are proposed, which are further used for machine learning. Results of the learning with C4.5, PART and Random Forest classifiers are compared with the baseline, showing significant improvement in memory leak detection performance.

Experimental results of C4.5 and PART algorithms were previously published in [vSor 13]

Chapter 6 contains descriptions of 4 case studies conducted to evaluate detection and runtime performance of the initial implementation of the statistical approach for memory leak detection. Case studies include validation of known memory leak from existing open source framework ActiveMQ, real-world eHealth web-application, along with a description of a memory leak in HtmlUnit framework which was found while testing and developing Plumbr. In addition, a comparative case study with an alternative memory leak detection tool, LeakChaser, is described.

Parts of the chapter 6, including case studies of ActiveMQ and the eHealth web application, were previously published in [vSor 11b].

Chapter 7 concludes the findings of the thesis and discusses the future research directions associated with this research.

Chapter 2

State of the Art

2.1 Garbage Collection

Automatic garbage collection and compilation to cross-platform compatible byte code are two most important and distinct features of any Java virtual machine. While cross-platform compatible byte code greatly simplifies deployment of applications across different operating systems and hardware architectures, automatic garbage collections simplifies the life of application developers taking away the burden of manual memory management. Understanding of the main principles of garbage collection is also necessary before addressing the memory leak issue we are trying to fix.

This section gives an overview of the terminology related to garbage collection and reviews most popular collectors used in modern JVMs.

2.1.1 Terminology

The Heap. The pool of memory where dynamic allocation takes place. Section 2.5.3 of the Java Virtual Machine Specification [Lind 13] defines the Java heap as follows:

> **Heap.** The Java Virtual Machine has a heap that is shared among all Java Virtual Machine threads. The heap is the run-time data area from which memory for all class instances and arrays is allocated.
>
> The heap is created on virtual machine start-up. Heap storage for objects is reclaimed by an automatic storage management system (known as a garbage collector); objects are never explicitly deallocated. The Java Virtual Machine assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementor's system requirements. The heap may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary. The memory for the heap does not need to be contiguous.
>
> > *A Java Virtual Machine implementation may provide the programmer or the user control over the initial size of the heap, as well as, if the heap can be dynamically expanded or contracted, control over the maximum and minimum heap size.*
>
> The following exceptional condition is associated with the heap:
>
> - a computation requires more heap than can be made available by the automatic storage management system, the Java Virtual Machine throws an `OutOfMemoryError`.

It must be noted that the JVM process, launched by the operating system, has its own native heap, managed by the operating system. The java heap is allocated within native heap and is managed by the JVM. Also, Java Virtual Machine specification does not prescribe how the heap should be structured, or how garbage collection should be performed. These are implementation details of particular virtual machines and relevant design decisions are driven by the requirements of the garbage collector algorithm.

The mutator and the collector. A garbage-collected program is separated into two semi-independent parts. The *mutator* executes application code (usually in several threads), allocates new objects, and mutates object graph by changing reference fields so that they refer to different destination objects. References may be contained in heap objects or in *roots* – static variables, thread stacks, etc. As a result of such mutations an object may get disconnected from the roots and become unreachable by following any sequence of edges from the roots. The *collector* executes garbage collection code (possibly in several threads) which discovers unreachable objects and reclaims their storage [Jone 11].

The mutator roots. In addition to the heap memory, mutator threads have a direct access to a set of references without traversing object graph. These references are called *roots*. Objects referenced by roots are called *root objects* [Jone 11]. Roots include references and variables on stack, static variables, threads, and Java Native Interface (JNI) references (reference from the native code to a Java object).

Reachability, liveness and staleness. An object is called *reachable* when there exists a path to it from any root. The *liveness* of the object is defined as an object being actively used in addition to being just reachable, i.e., if reachability is a property, which prevents the garbage collector from freeing the object, then liveness of the object shows whether the object is still needed for the application. Liveness can be measured only during runtime and is not available, for example, in a heap dump. Liveness, however, cannot be predicted

Staleness of an object indicates whether an object has not been used for a while. Staleness can be measured as a time since an object was last used/accessed by the application, and the longer an object is not used, the more stale it becomes. Staleness of an object is the best indicator of an unused object; however, it can be expensive to calculate and obtain, as all accesses of an object by the program must be captured. Staleness can be measured only during runtime and the information required for the calculation is not available, for example, in a heap dump.

Strong references and reference objects. A reference from one object to another via direct field or variable reference is called a *strong reference*. An object is strongly reachable if it can be reached from the GC roots via strong references only (without traversing any reference objects). If an object is strongly reachable, it cannot be garbage collected.

Weak/soft reference is a reference from one object to another made by using proxy objects implementing an interface of `java.lang.Reference`. Implementations include `java.lang.ref.WeakReference`, `java.lang.ref .SoftReference`, `java.lang.ref.PhantomReference`. These reference classes are of special meaning to the garbage collector. If an object is reachable only via weak reference object then the object is called *weakly reachable*.

If an object is reachable only via a soft reference then the object is called *softly reachable*. If an object is weakly reachable, then it is eligible for finalization (a special method to be called before the object can be disposed) and garbage collection, and thus will be reclaimed. The reference object will be notified that the object it was referring was collected. If an object is softly reachable, then the garbage collector can choose not to collect the object as soon it becomes softly reachable but it can leave the object on the heap until memory pressure arises. Softly reachable objects are guaranteed to be collected before `java.lang.OutOfMemoryError` will be thrown (see [Orac 13]).

All instances of reference classes may be associated with a *reference queue*. When an object, pointed to by the reference object, is enqueued for garbage collection, the reference object may be notified about this event via the reference queue. If weak and soft references may make an enqueued object strongly reachable again, thus preventing the collection, then phantom references are only notified about objects which are already collected and thus cannot be made strongly reachable again. Phantom references can be used for an alternative implementation of finalization, or for enforcing an order in which objects are collected. These uses are quite advanced and thus phantom references are seldom used in practice.

Dangling references, loitering references and objects. In programming languages with manual memory management a dangling pointer emerges when the object, to which the pointer was pointing is freed, but the pointer itself is not nullified. Thanks to garbage collection dangling pointers do not occur in managed languages; however, the term is often encountered in respective literature.

Unnecessary/loitering reference indicates a situation when the object, to which the reference is pointing, is no longer needed from the application perspective; however, because of the reference, it cannot be reclaimed by the garbage collector. If the reference is removed, the unneeded object will be reclaimed. Loitering references cause memory leaks in runtime environments with garbage collection.

Loitering object is a condition causing leaks in languages with manual memory management. This happens when the reference to the allocated memory is lost, but the memory region has not been deallocated. Loitering objects are not an issue in garbage collected languages, thanks to garbage collection, which detects object with no incoming references and reclaims the occupied memory.

Stop-The-World operation. Some garbage collectors require that the mutator threads are stopped, while the collection is performed. Threads may be stopped at a *safepoint.* There are several reasons for this requirement. For example, collectors may relocate objects and thus existing references must be updated. Stopping the threads simplifies synchronization and allow for some operations to complete faster. Further optimization of particular implementations are targeting to reduce the length of the stop-the-world pause as much as possible. Optimizations include parallelizing collection, if there are multiple CPUs available, or running some phases of the collection concurrently with the application and stopping the application only for compaction. These details will be described in following subsections.

2.1.2 Mark-sweep garbage collection

Mark-sweep garbage collection is one of the first algorithms for garbage collection, described by McCarthy [McCa 60] for the LISP programming language. The algorithm works in two steps. Starting from the roots, it traverses all objects that can be reached by following intermediate references. This phase is called *tracing*. During this traversal, objects are *marked* as reachable. The next step, *sweeping*, removes all unmarked objects from the heap, as they cannot be reached, and thus are unused.

Tracing is an *indirect algorithm*, i.e., it does not detect garbage, rather it detects reachable objects and everything else is considered to be garbage [Jone 11]. A direct algorithm is the *reference counting*, which considers an object to be garbage if it has no incoming references. However, as garbage collectors used in modern Java virtual machines are all tracing collectors, reference counting is not described in further detail.

In the simplest form mark-sweep is fully stop-the-world operation. However, it is possible to reduce the length of the pause, for example, by stopping the threads briefly to scan their stacks and further perform marking concurrently.

Mark-sweep collection does not move objects, thus it is a subject for heap *fragmentation*. Heap fragmentation means that although there might be enough of free space in the heap, none of it is in contiguous block, which is big enough for the requested allocation. Although there are strategies for keeping fragmentation low, they all require overhead during the allocation phase, while the block with correct size has to be found at first. In addition strategies for reducing fragmentation make assumptions about object sizes and do not work well enough in a long-running application which allocates objects of various sizes.

2.1.3 Mark-compact garbage collection

The unfragmented heap allows very fast sequential allocation, because the object may be always allocated next to the used memory. Such allocation technique is called *bump the pointer*, as the allocation only consist of ‘bumping’ the pointer to the end of allocated memory. To take advantage of such fast allocation, the heap must be compacted at some point. Mark-compact collection addresses exactly this issue.

The first phase of mark-compact is marking, the same as in mark-sweep collection. The second phase performs compaction by relocating the objects and updating the references to all live object which have moved. The compaction itself may be performed using different approaches: arbitrary, linearising or sliding. Arbitrary approach relocates objects without regard for their original order or whether they point to one another. Linearising approach relocates objects so that related (referenced) objects are as close as possible. Sliding approach slides objects to the one side of the heap, squeezing out garbage, thereby maintaining original allocation order in the heap [Jone 11]. In any of these approaches old objects tend to accumulate on the bottom of the heap.

The downside of this approach is that in order to compact the heap, several passes over heap must be performed by the collector, which increases the time of the collection.

2.1.4 Copying garbage collection

Further advancement of the mark-compact collector is copying collector, where instead of compacting the whole heap using several passes, the *semispace copying* is performed. The heap is divided into two *semispaces*:

fromspace and *tospace*. Objects are allocated in one semispace only. After marking, collector performs *evacuation* or *scavenging* of all survived objects from one semispace to another and the first semispace is not used until next collection. This allows to perform compaction in one pass over the semispace, compared to multiple passes required by mark-compact algorithm.

Copying collection allows fast allocation and is easier to implement than the mark-compact. It is not subjected to fragmentation compared to the mark-sweep. However, in its simplest form, copying collector requires twice as much memory compared to the mark-sweep or mark-compact algorithms, or with the same amount of heap it will require twice as much collections [Jone 11].

2.1.5 Generational garbage collection

As noted in subsection 2.1.3, long-lived and older objects tend to accumulate on the bottom of the heap. Some compacting collectors avoid compacting these areas, but they have to be visited during the tracing in order to identify reachable objects. In addition to these observations, two generational hypotheses are stated: *weak generational hypothesis* and *strong generational hypothesis*.

The *weak generational hypothesis* states that most newly created objects live for a very short period of time [Lieb 83], or 'die young'. Weak generational hypothesis is supported by multiple research results from object oriented (Smalltalk, Java) and functional (MacLisp, Common Lisp, Haskell, Standard ML/N) programming languages.

The *strong generational hypothesis* states, that even for objects which are not newly created, younger object will have a lower survival rate than older ones. For this hypothesis there is much less evidence and long-lived objects may have much more complex lifetime patterns depending on the application.

Generational collectors use weak generational hypothesis and separate the heap into regions based on the object age, or *generations*, and apply best collection algorithm to each generation separately. Younger generations are collected before old generations and objects that survive long enough are *promoted*, or *tenured* to older generations [Unga 84]. Most generational collectors collect young generation using copying and old generation using mark-sweep or mark-compact.

The documentation for Oracle HotSpot JVM contains figure 2.1 which depicts the average distribution of survived bytes over different garbage collection cycles.

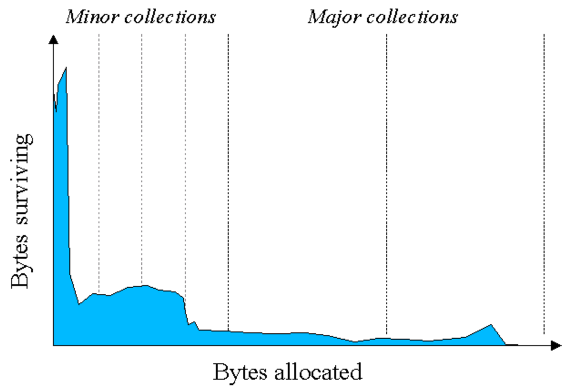


Figure 2.1: Distribution of survived bytes over generations
Distribution of survived bytes over generations, source: [Java]

2.1.6 Garbage collectors in HotSpot JVM

Oracle HotSpot/OpenJDK is the reference implementation of the Java Virtual Machine specification [Lind 13]. It was first released in 1999 by Sun Microsystems. In year 2006 HotSpot JVM was licensed under GPL license, which is now known as OpenJDK and which became official Java 7 reference implementation. In 2010 Oracle corporation acquired Sun Microsystems and since then Sun HotSpot JVM became Oracle HotSpot JVM. In year 2014 version 8 of HotSpot JVM was released.

As the HotSpot JVM, being a reference implementation, is the most widely used, its heap layout and garbage collectors are reviewed in more detail. HotSpot JVMs utilizes generational garbage collection and as of version 5, 6, 7, and 8 have four generations ([Java]) – *young*, *survivor*, *old* and *permanent* (called MetaSpace in version 8). The layout of the heap is shown on Figure 2.2. *Virtual* regions mean that although maximum heap setting may be specified using `-Xmx` parameter, actual sizes of the generations may be smaller, depending on the actual usage. Generations may be resized up to a maximum size at the expense of virtual unallocated space.

The permanent generation keeps objects which should not be collected at all or should be collected very rarely (e.g., on application redeploy in the application server). Class definitions, the string pool, static fields, etc., are kept in the permanent generation. In Java 8 permanent generation, which was held in the heap, is replaced with *MetaSpace*, which is allocated in the native heap, outside of Java heap, and by default is not limited, in contrast with previous implementation.

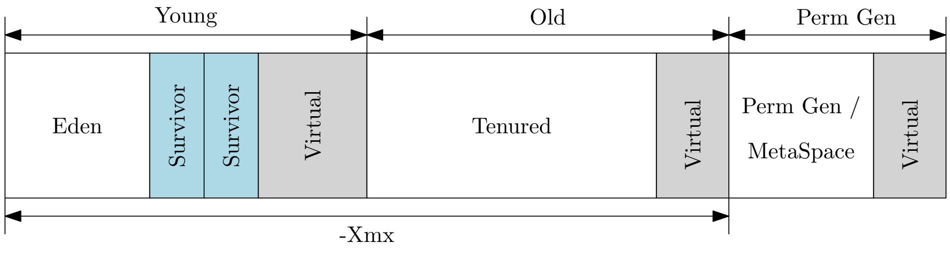


Figure 2.2: Heap layout in HotSpot JVM

All regular objects are created in the young generation. Young collector is a copying collector, which copies survived objects to the survivor generation. Survivor space is a semispace, where survived objects are copied from one semispace to the other during young collection. After surviving a number of young garbage collection cycles, objects are promoted to the old generation. When old generation fills up, the collection of the old generation is triggered. Old generation is performed using mark-compact collection. Garbage collection which occurs only in young generation, is called *minor collection*, and collection of the old generation is called *major collection*.

HotSpot includes several implementations of the garbage collector which can be selected using command line parameters. The *serial* collector is a single-threaded collector, which is best suited for single-processor systems.

The *parallel* collector performs collections in parallel, utilizing multiple processors or cores. Both evacuation and compaction can be performed in parallel (parallel compaction was introduced in Java 5 update 6 and is enabled by default since Java 7 update 4). Parallel collection reduces the duration of the stop-the-world pause and is therefore also called the *throughput* collector.

The *concurrent* collector is a concurrent mark-sweep (CMS) collector which performs part of the marking and sweeping concurrently with the application. The main goal of CMS is to keep enough free memory in tenured space so that promotion (which is still stop-the-world operation) won't fail. As CMS is a non-compacting, fragmentation may occur and due to fragmentation promotion may still fail. Promotion failure will trigger the compaction, which is performed in a single thread.

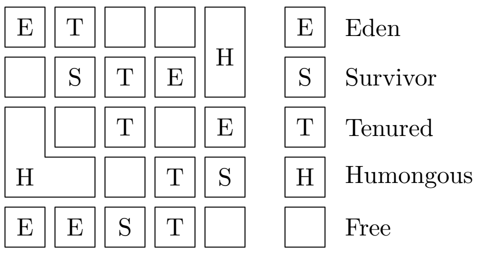


Figure 2.3: Heap layout for Garbage First (G1) collector

The last collector is the *Garbage First (G1)* collector [Detl 04]. It was first introduced in Java 6 as an experimental feature and is official in Java 7. G1 was designed to reduce long pauses for large heaps and solve main CMS problem – fragmentation due to lack of concurrent compaction. Although G1 has the same generations as other collectors, it organizes the heap differently, as shown on Figure 2.3. Instead of having continuous eden and tenured regions, they are divided into smaller fixed-sized regions, which can be collected separately. Humongous regions are intended for allocations which occupy more than 3/4 of the heap region. Size of humongous regions are multiples of the default region size. As is the CMS, so is the G1 a partly concurrent collector. G1 performs the marking concurrently and then evacuates as a stop-the-world operation. Also, like the CMS, G1 can suffer from promotion failure, which triggers the full GC.

2.1.7 Garbage collection in other Java Virtual Machines

IBM J9 is a proprietary Java virtual machine developed by IBM. J9 is mostly distributed as a part of other IBM products, like IBM WebSphere application server, and thus is rarely used on its own outside of these products. J9 support started with Java 5. IBM J9 also uses generational concurrent mark-sweep garbage collection. Young generation is called *nursery* in J9. Unlike HotSpot's separate eden and survivor spaces, nursery is fully semispace and its semispaces are called *allocate* and *survivor* [Gene].

J9 also includes a *balanced* collector, similar to G1 collector in HotSpot. It divides the heap into smaller regions and performs frequently partly concurrent collections of these smaller regions instead of infrequent but long collections of large regions, especially on large heaps.

Oracle jRockit is a proprietary Java virtual machine, initially developed by Appeal Virtual Machines, later acquired by BEA Systems in 2002, which in turn was acquired by Oracle corp. in 2008. After acquisition by Oracle, jRockit started to be integrated into HotSpot.

jRockit have two options for memory layout: single generation or two generations. In single generational layout the heap is collected using either the mostly concurrent or the parallel mark-compact collector. In two generational layout the heap is divided into the nursery and old regions, where after minor collections survived objects are promoted directly into old generation. Two-generational layout may also be collected with either the mostly concurrent or the parallel collector [Tuni].

Jikes RVM is the Research Virtual Machine, designed to provide a platform for experimentation with technologies related to the construction of virtual machines. Jikes RVM is itself implemented in Java. Jikes RVM includes The Memory Management Toolkit (MMTk) for use in research of memory management advances, therefore Jikes RVM contains implementations of all kinds of garbage collectors. However, despite the state-of-the-art research in the virtual machine area, the Jikes' class library is not so state-of-the-art and is not fully compatible with OpenJDK, therefore not all software written in Java may run on Jikes [The 10].

Azul Zing is the proprietary commercial JVM from Azul systems. Its distinctive feature is the Continuously Concurrent Compacting Collector (C4). It solves the main problem of other mark-compact collectors – expensive compaction pause which eventually will occur regardless of the internal optimizations of a collector. The compaction pause is the most expensive part of the collection because, while objects are relocated during compaction, all references pointing to the relocated object must also be updated. The more live objects and references there are in the heap, the longer the compaction pause will take. C4 solves this problem by using hardware read barriers. Since 2005 Azul has provided required features in custom hardware system Vega with custom multi-core processors and specialized kernel. Since 2010 Azul implemented required feature set using modern x86 processors and respective supporting modules in the Linux kernel [Tene 11]. The idea of using read barriers (or *Loaded Value Barrier, LVB*) for garbage collection is to delay updating the references to relocated object only when the respective reference is actually read, instead of updating all references at once during collection.

2.2 Machine Learning

A formal definition of machine learning can be stated as follows [Mitc 97]:

Definition 1. A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

So, to apply machine learning one has to define a set of data to learn from (E), a measure of performance to improve (P) and identify tasks (T) which the learning has to achieve.

Tasks, performed by the machine learning can be separated into common types. These types include among others:

- *Classification*: Assigning a label, or a class, to the data. Probably the most classic example of a classification problem is the detection of spam email.
- *Regression*: Data is assigned a real floating-point value instead of a label. Again, a classic example is prediction of a stock price over time.
- *Clustering*: Data is divided into groups based on similarity. Examples of clustering include market research, sequence analysis in bioinformatics, etc.
- *Association rule learning*: Discovering relations between attributes in the data. Widely used in data mining, for example business rules mining from the large database of facts about the customers.

Classification is the most relevant type of machine learning tasks in the context of current thesis. Leaking objects must be identified to separate a memory leak, thus objects are assigned one of two labels – *leaking* or *not leaking*. Classification task which has only two labels to assign is called *binary classification.*

One may find clustering suited for the task of dividing objects into two clusters of leaking and not leaking objects. However, main difference between clustering and classification is that classification has a predefined set of labels to assign, whereas clustering tries to find any similarities in the data instead of having any predefined groups.

2.2.1 Evaluation of the performance

According to the definition, a measure of performance to improve is also required to apply machine learning. In terms of performance, each result returned by the binary classifier belongs to one of 4 possible outcomes:

1. true positive (TP) – a leaking object was correctly identified as leaking by the classifier,
2. true negative (TN) – a non-leaking object was correctly identified as non-leaking by the classifier,
3. false positive (FP) – a non-leaking object was incorrectly identified as leaking by the classifier,
4. false negative (FN) – a leaking object was incorrectly identified as non-leaking by the classifier.

Such outcomes are usually presented in a form of *confusion matrix* shown in Table 2.1.

	Predicted	
Actual	**Leaking**	**Non-leaking**
Leaking	True Positive (TP)	False Negative (FN)
Non-leaking	False Positive (FP)	True negative (TN)

Table 2.1: Confusion matrix

All further relevant performance metrics are calculated from these four metrics. First relevant derived quality metrics are *precision* (or *sensitivity*) defined as 2.1 and *recall* (or *specificity*) defined as 2.2. In the context of memory leak detection precision indicates fraction of detected leaks which are actually leaks, and recall indicates fraction of leaks detected by the classifier out of all actual leaks.

$$precision = \frac{tp}{tp + fp} \tag{2.1}$$

$$recall = \frac{tp}{tp + fn} \tag{2.2}$$

Last, a combined measure for the classifier performance is required. There are two alternatives to choose from: accuracy and F-measure. Accuracy shows a fraction of classifier decisions that are correct and is defined as $(tp+tn)/(tp+fp+fn+tn)$. The problem with accuracy in the context of memory leak detection is that it also accounts for true negatives. However, the number of leaking objects in an application in normal conditions, i.e., not in a synthetic test, is very small. So, labeling everything as non-leaking would still produce a very high accuracy, which is not desirable.

An alternative to the accuracy is the *F-measure*, which is a weighted harmonic mean of precision and recall. Using harmonic mean instead of arithmetic mean avoids the possibility to get a high combined measure by labelling all allocations as leaking and thus get 100% recall and therefore at least 50% arithmetic mean. When the values of precision and recall differ greatly, harmonic mean is closer to the minimal value, rather than to arithmetic mean. Using weight in the F-measure makes it possible to favor precision or recall in the resulting measure [Mann 08]. In case of memory leaks precision and recall may be treated equally (all memory leaks must be detected – high recall; as small number of false alarms as possible – high precision) thus even weighting will be used. F-measure with even weighting, or balanced, is also called and F_1 score and is defined as:

$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{2.3}$$

Another metric commonly used to evaluate classification algorithms is a ROC (Receiver Operating Characteristics) curve and area under the ROC curve (AUC). ROC curve plots true positive rate (defined as $tp/(tp + fn)$, or recall) against the false positive rate ($fp/(fp+tn)$, or $1-precision$) and the AUC shows the probability that a classifier will rank random positive instance higher than random negative instance. In case of discrete classifiers such ranking can be obtained when probabilities of belonging to one class or another are compared.

Classification algorithms can produce models of different kinds, but on high level they can be divided into two major categories: *black box* and *white box.* While black box models may provide good results, they give little understanding on how data attributes affect the final decision. Examples of black-box models include artificial neural networks, support vector machines, etc. White box models on the other hand can be interpreted, understood and thus implemented and debugged, especially when the number of attributes in not very high. This last feature is important from the practical and engineering standpoint – generated model may be easily implemented in any language, deployed to the end-user and results can be interpreted in case of wrong results. For this reason white box classification algorithms are of primary interest in the context of current thesis. Examples of white box models include decision trees and rule sets.

2.2.2 C4.5 classifier

C4.5 is a general classification algorithm widely used in practice and developed by Ross Quinlan [Quin 93]. It produces a decision tree in a form of a set of *if-then* rules. Each rule in a non-leaf node in the decision tree must contain a test that will divide the training cases. The main question is how to select the best rule to be used in a node? An ideal binary rule would divide all elements in a data set into correct classes. Such an ideal rule usually is hard or impossible to find. C4.5 builds an initial tree and then iteratively globally improves it using heuristic techniques, namely concepts of *information entropy* and *information gain*. These concepts from information theory allow choosing the rules which extract the maximum amount of information from a set of cases, with a constraint that only single attribute may participate in the rule. Improving the decision tree implies dropping redundant rules and optimizing remaining.

2.2.3 PART classifier

Another classifier, PART, generates a decision list based on the repeated generation of partial decision trees in a separate-and-conquer manner [Fran 98]. Separate-and-conquer stands for removing all instances in the data set for which each new rule matches. Partly PART is based on the C4.5, but instead of generating full decision tree and optimizing it (which is a complex and time consuming process), as C4.5 does, PART builds "partial" decision trees which contains branches to undefined subtrees.

2.2.4 Random Forest classifier

Random Forest classifier is introduced by Leo Breiman in [Brei 01] and it belongs to the *ensemble of trees* family of decision tree classifiers. Its formal definition is as follows:

Definition 1. A random forest is a classifier consisting of a collection of tree-structured classifiers $\{h(x, \Theta_k), k = 1, \ldots\}$ where the $\{\Theta_k\}$ are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input x.

This definition is explained by Breiman [Brei 01] as follows:

Random Forests grows many classification trees. To classify a new object from an input vector, put the input vector down each of the trees in the forest. Each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification having the most votes (over all the trees in the forest).

Three important parameters define how the forest is grown: number of trees, number of features and depth of the tree to be generated. Number of trees specifies how many trees should there be in the forest. Trees are grown using binary splitting, where each decision node is split in two children. Number of features defines how many random variables are chosen for any particular tree to start growing.

Random Forest classifier copes well with data sets including large number of attributes and data sets with small number of samples.

2.3 Summary

This chapter described basic terminology related to the dynamic memory management and garbage collection, and discussed the state of the art in garbage collection in Java virtual machines. Conceptual approaches for garbage collection like mark-sweep, mark-compact and generational garbage collection were described. Most widely used heap layouts of modern Java virtual machines were described.

The chapter also introduced basic machine learning concepts and approaches for evaluating the performance of learning algorithms in a volume required for practical application of the machine learning in current thesis. Further used classifiers, C4.5, PART and Random Forest, were introduced.

Chapter 3

Classification of Memory Leak Detection Techniques

An online search for the terms “memory leak java” or “OutOfMemory-Error” finds thousands of blog posts, forum, and mailing list discussions, which means that memory leaks in JVM languages are not just a theoretical problem. Memory leak detection has been studied over the years and several solutions have been proposed. In [vSor 14a] memory leak approaches were reviewed considering their implementation complexity, measured metrics, and intrusiveness. As a result, the classification of memory leak detection from analyzed standpoints is proposed.

The state-of-the-art approaches for memory leak detection can be classified as methods implementing:

1. online detection, further separating methods into
    (a) measuring staleness,
    (b) detecting growth,
2. offline detection, including methods
    (a) analyzing captured stated, e.g., heap dumps,

    (b) using visualization to aid manual leak detection,
    (c) static analysis of the source code.

3. Hybrid methods, combining features from both online and offline approaches.

Following sections will describe state of the art in research according to the defined classification.

3.1 Online methods

Online methods actively monitor and interact with the running virtual machine in order to detect leaking objects. The advantages of online methods are access to run-time information, such as allocation trace, an object's liveness and activity. The main problem of online methods is that they impose overhead on the running application. Some methods use metrics, which are very expensive to measure, thus limiting the applicability of these methods in real production systems. Another limitation is the kind of information, which is used for detection – several online methods rely on features not present in modern production JVMs and require modification of either the garbage collector or some other JVM internals (or both). Such methods are implemented in research JVMs like Jikes RVM which limits their adoption in industrial use, as it is highly unlikely that administrators of a critical system (which probably may only use a JVM, which is certified for use with a particular very expensive hardware or operating system) will deploy such a system in production on a research JVM just to find the memory leak.

Self-healing systems including a fair amount of research are an interesting succession of online leak detection methods. In addition to detecting the leak and reporting it to the user, self-healing systems extend runtime of the system suffering from a memory leak by providing countermeasures, such as swapping out leaked objects, or weaken references that keep leaking objects reachable.

On the conceptual level, online methods can be divided into two groups: staleness detection and growth analysis. In the following subsections these groups will be reviewed in more detail.

3.1.1 Staleness detection

As defined in Section 2.1.1, the staleness of the object is measured as the time since the program last actively used it. The intuition behind this metric is simple: if an object remains long enough in memory and is not used, then it is probably leaked. The main problem with staleness detection is that recording all object accesses without modifying the JVM is extremely expensive, as in addition to each read access there will be also one write access to somehow track the usage. As usage of brute force methods is clearly ineffective for this purpose, researchers try to find ways to implement this complex task more effectively.

Bond et al. [Bond 06] proposes *Bell* (Bit Encoding Leak Location) — encoding and decoding of per-object leak-related information using a single bit. It is a probabilistic encoding, which loses a lot of information, but given sufficiently many objects and a known finite set of allocations (every program has a finite number of lines of code where objects are instantiated), encoded data can be decoded with high confidence. To store allocation sites encoded with Bell for each object, one of four free bits in an object header in Jikes RVM was used, which means that no new memory overhead is introduced.

After implementing proposed bit-encoding in a Jikes RVM, Bond et al. implemented a leak detection approach utilizing Bell in a leak detection tool called *Sleigh*. In addition to one bit of allocation encoding in the object header, Sleigh adds the last-used site using Bell and a two-bit *saturating stale counter*, counting the time since the object was last accessed. The stale counter is implemented as a *logarithmic counter*, i.e., the counter contains a logarithm of the time which passed since the last object access.

Since the counter occupies two bits, it saturates at a value of 3. The base of the logarithm counter is fixed to 4 in [Bond 06].

These counters are added using the remaining 3 free bits in the object header. Sleigh instruments all methods to update the last-used site counter and reset the staleness counter whenever the object is referenced during program execution. Jikes RVM's garbage collector is changed in a way that when it traverses an object graph it increments the staleness counter after every predefined number of collections.

Leak detection is performed by periodically decoding and analyzing highly stale objects. Execution time overhead is 29% but by using the adaptive statistical profiling technique described by Chilimbi in [Haus 04] this overhead is further reduced to 11%.

Bell was later used by Tang et al. in the LeakSurvivor [Tang 08], which is one of the self-healing methods. LeakSurvivor uses Sleigh to detect potential leaks, and then swaps them out and in, if needed. If a previously swapped out potential leaking object is swapped back in, the object is marked as not leaking. LeakSurvivor keeps track of swapped out objects using the *Swap Out Table* (SOT).

LeakSurvivor is implemented as a part of the garbage collector in the Jikes RVM [The 10].

A very similar self-healing system is implemented by Bond and McKinley in the tool named *Melt* described in [Bond 08]. The general approach to swapping objects from the heap to disk and back is the same: to detect object staleness, store stale objects to disk, and activate stale objects when needed. Staleness detection is implemented by changing both the garbage collector (to mark the object during the collection phase as stale) and dynamic compiler (to instrument bytecode to unmark the object on use). The collector also moves the stale object to disk.

If a stale object is referencing an active object, then a compacting or copying garbage collector may move the active object in the heap and the reference in the stale object must be updated to point to the new location of the active object. To mitigate this problem, *stub-scion* pairs are used, where the stale object on the heap is replaced with a *scion* (an object pointing to a stale object) and the *stub* part is swapped out. Scions are special to collectors and cannot be relocated by them. Melt keeps track of swapped out objects using a *scion table*.

Melt is implemented in the garbage collector of Jikes RVM [The 10].

LeakSurvivor and Melt were developed concurrently and the main difference between the two is that Melt guarantees "space and time proportional to in-use memory" [Bond 08]. This means that Melt is able to handle the case when in-memory object references swapped out a stale object and this in-memory object becomes stale on its own. In this case, LeakSurvivor still keeps the reference between the now two stale objects in the heap, whereas Melt is able to swap out that reference as well, thus freeing more heap. In addition, it is claimed that Melt incurs less stale object detection CPU overhead – 6% in Melt vs 21% in LeakSurvivor.

Evolutionary improvement to Melt is described by Bond and McKinley in [Bond 09]. As an improvement from previous work ([Bond 08]), instead of monitoring object staleness, whole data structure (object subgraph) staleness is identified and instead of swapping out the data structure it is reclaimed (*pruned*) altogether. Bell [Bond 06] is used to detect staleness of the data structure.

The leak pruning approach waits for the heap to become exhausted (the threshold is defined externally by specifying *expected memory use*) before predicting and reclaiming possibly dead objects. Reclaiming such a reference is called *reference poisoning*, and when the application accesses a poisoned reference, an error is thrown. This preserves application semantics as if leaking objects have not been reclaimed and then the program would run out of memory anyway (guaranteed by the fact that heap exhaustion triggers pruning).

Leak pruning is implemented in the garbage collector of Jikes RVM [The 10].

Another leak detection technique utilizing object staleness is proposed in [Rays 07] by Rayside and Mendel, where *object ownership profiling* is described. In addition to memory leak detection, the described profiling technique can be used to detect memory usage inefficiencies in the profiled application. Such inefficiencies (or *anti-patterns*) include:

1. *extending a mutable base class* – inheritance of unneeded fields, occupying heap;

2. *failure to release dormant references* – also known as a *leaking listeners* anti-pattern, a reference which was created when an object was active but keeps holding the object after it became inactive;

3. *construction of zombie references* – a references which were created after the object became inactive;

4. *tangled ownership contexts* – expected encapsulated data structure referring to unexpected external structures;

5. *bloated facade* – heavyweight facade object (providing unified access to some subsystem) with lots of internal dependencies; when used not as a singleton, may introduce significant overhead; if a long-lived object, requiring only small subset of functionality retains a reference to such a facade.

Object ownership profiling technique records the unique identifier of every object, its size, time of creation, collection time, source and target of every method call or field access. Collected trace is analyzed by plotting allocated space of reachable and active (actively referenced in the trace) objects over time. An observation which can be made from these plots is whether an object is alive much longer than it is used, i.e., stale, in which case there is some memory inefficiency or leak. In addition, these plots show how much space is occupied by unneeded data.

Collected information is presented to the programmer as an object ownership hierarchy which is annotated with the aforementioned plots showing space occupied by the objects over the time of reachability and liveness. Analysis, which has to be performed by the programmer, consists of correlating object ownership, retained heap size, reachable time and liveness time to infer memory leaks or memory usage inefficiencies.

As the authors admit, instrumenting as much bytecode as possible is extremely expensive, which makes object ownership profiling a very heavyweight method to be used to diagnose complicated memory problems in the development environment.

The last method using staleness as a main indicator for memory leak detection is a method for "precise memory leak detection for Java software using container profiling" described in [Xu 08, Xu 13] by Xu et al. The method's main assumption is the observation that most of the leaks happen via collection classes (or *containers*, e.g., classes belonging to Java Collections API). So, instead of monitoring all objects equally, the approach focuses on monitoring only collection classes for growth and element access times to detect staleness.

Instead of starting with the assumption that there are no leaks in the application and detecting leaks, the method starts by suspecting all containers and during the runtime it rules out non-leaking ones. Containers are identified by code annotations which map *add, get* and *remove* operations to actual implementation. For the default Java collections API this is performed automatically, but corresponding methods in custom collections must be annotated manually by the user.

Next, the application's total memory consumption is monitored to rule out non- leaking collections. If heap usage is growing over a period of time, then the containers' size changes are correlated with the overall heap consumption. If the container is not growing along with the heap, then this container is ruled out.

Leak candidates are ordered by *leak confidence*, which is composed of memory usage *contribution* and staleness *contribution*. Leak confidence is defined as an exponential function of staleness contribution, indicating that staleness is a more important parameter. An important difference in staleness calculation, compared to the one given in Section 2.1.1, is that staleness is computed as the time since the object was placed into, or retrieved from the collection, rather than the time since use of the object.

One aspect, which this approach does not account for, is the origin of the objects in the collection. The allocation site of the object being leaked is valuable information to find and fix the source of the leak. This contrasts with previous approaches, which also tracked allocation sites of leaking objects, whereas this method only monitors containers of leaking objects.

3.1.2 Growth analysis

Online leak detection algorithms based on growth analysis look at various parameters growing over time while the application is working. These various parameters can be the cumulative size of objects of a certain type, object count, or the number of different generations. The current subsection reviews these approaches in more detail.

There are two ways to instrument a running Java program: by performing direct bytecode instrumentation using an agent or using an aspect-oriented programming (AOP, see [Kicz 97]) library. If bytecode instrumentation allows for changing any place in the code and is the most low-level approach available in the JVM, then AOP operates with higher level concepts to select what to instrument – *join points*. Usually join points are either method calls or field references. AOP libraries allow code to be added before, after or around the join points.

In 2007, Chen et al. in [Chen 07] described the usage of AOP for memory leak detection, implementing this approach in the *FindLeaks* tool. FindLeaks analyzes memory leaks by using an AspectJ [Ecli 13] library to collect information about instantiation and references to leaking objects. The intuition behind FindLeaks is simple – track object creation and disposal and if more objects of a certain class are created than freed, then this particular class is leaking. In addition allocation traces of objects are also collected to be presented to the end-user. The end-user must specify the package to monitor, so that only a subset of the application can be tracked. Notification mechanism provided by `WeakReference` class is used to track object destruction.

Leak detection itself operates on the three aforementioned hash tables to calculate ratios between construction and destruction over time to calculate trends and present that information to the end-user in a color-coded form for visual evaluation.

Unfortunately, the publication [Chen 07] does not provide any performance analysis. From the description, the method looks quite heavyweight, both CPU and memory wise, as it maintains a separate object hash table which lists all created objects, a reference hash table with reference related information, and a method hash table to keep track of references pointing in/outside of the package which was enabled for tracking.

In 2006 Jump and McKinley in [Jump 07, Jump 06] described a memory leak detection approach performing size growth analysis. The authors have implemented a dynamic memory leak detection approach for garbage-collected languages in a tool called *Cork*.

The main contribution of the work is the description and construction of the *Type Points-From Graph* (TPFG) data structure, which summarizes dynamic object graph by class. The usual representation of the heap is the *objects point-to graph*, where each node is a single object and each directed edge is an outgoing reference (from the object holding the reference to the object being pointed to). Nodes in TPFG summarize objects by their class while edges contain references between types turned backwards – the outgoing directed edge describes incoming references. Nodes of TPFG are annotated with volumes of live objects for each class. Edges of TPFG are weighted by the volume of nodes to which edges are pointing to.

To fill TPFG with data, the authors have modified Jikes RVM's [The 10] garbage collector in order to update TPFG with size data on every full collection while the collector scans the heap. The authors report very low overhead, both in terms of space and time – less than 1% and 2.3% respectively.

To find memory leaks, TPFG's evolution is analyzed over time in terms of growing types. Cork reports a chain of references between types starting from the growing nodes up to non-growing nodes. Along with a chain of references, Cork also reports type-based allocation sites. A type based allocation site means that for each leaking class all allocation sites are reported, not only allocations which constructed leaking objects.

In 2011 an idea for the statistical approach to memory leak detection, which is the main topic for current thesis, was first presented [vSor 11a, vSor 11b]. The method was refined and further researched in subsequent years ([vSor 13, vSor 14b]) and also implemented in the commercial tool called Plumbr. The method falls in the growth detection category. Although the method looks for objects that are steadily created but not freed over a period of time, the method does not account for object access times, thus it is a growth and not a staleness detection method.

The statistical approach for memory leak detection is the main contribution of the current thesis and thus will be described in detail in following chapters 4 and 5.

A separate group of methods, utilizing growth detection, is composed out of self-healing systems, which monitor the growth of heap usage and then try to swap out or weaken references to specially marked objects.

The first work on self-healing systems involving memory leak recovery for Java applications is described in [Brei 07] by Breitgand et al. and covering the *Panacea* self-healing framework. Leak toleration is only one feature of this general self-healing framework. For Panacea to work, a programmer must annotate the code with annotations defining classes that can be accessed by tools, i.e., (*Healers*) for monitoring, configuration and management tasks during runtime.

A healer intended to fix memory leaks, *ObjectDumpHealer*, can swap out annotated objects to disk when the heap fills above a predefined ratio. Objects that are annotated as `@Dumpable`, are replaced by the proxy object which, if needed, can swap the original object out and then swap in, when accessed. Thus, from the point of view of the leak detection the approach doesn't provide any mechanism to detect leaking objects automatically and swap them out, rather requiring the developer to annotate certain classes suitable for such management.

This solution is therefore intended to survive any memory shortage by swapping out objects rather than throwing the *OutOfMemoryError*. A nice feature of the approach is that it can be implemented in a non-intrusive way, using only bytecode modification.

Evolutionary research in this field is described by Goldstein et al., in [Gold 07], which adds a new healer agent to the aforementioned Panacea framework (see [Brei 07]) – *LoiteringObjectsHealer*. The principle of operation of the `LoiteringObjectsHealer` is very similar to the `ObjectDumpHealer` described by Breitgand. From the description the only noticeable improvement seems to be the ability to notice when proxy objects are garbage collected (using finalizers) so that swapped out objects can also be reclaimed.

3.2 Offline methods

Offline methods are intended to help the programmer analyze what has happened in the program by analyzing either dumps or by gathering some other data to analyze and visualize without affecting the running system. The advantages of such methods are clearly low overhead, especially in the case of heap dump analysis, which can be acquired as a post-mortem artifact of the JVM.

The weak point of offline analysis, and heap dumps in particular, is that runtime information like allocation trace or other runtime behavior is not available offline.

Heap dumps, for example, miss allocation information, thus it is not possible to detect *where* leaked objects were created. Also, heap dumps do not contain temporal information about *when* leaked objects were created. Both of these questions are important to understanding why a leak happens in the first place. Another complication is that by default a heap dump is not created when an `OutOfMemoryError` is thrown, and an additional JVM parameter must be provided in startup scripts. This means that if a leak requires several days to grow big enough and crash the application, then to start working on the heap dump another couple of days are needed to get the dump (which also will include crashing the production environment and affecting end users).

As the heap dump contains all the information stored in the heap by the application, it also may contain sensitive data, which may not be seen by developers, thus making it complicated to find and fix the leak.

Other types of offline analysis include visualization techniques, which visualize data in a way that helps a programmer spot memory usage problems in general and find memory leaks in particular.

This section focuses on both dump analysis and visualization techniques.

3.2.1 Analysis of captured state

Analysis of captured state is widely used in different forms in the work previously reviewed. Capture of state can take different forms. The most popular and standard way to capture the state is to generate a heap dump using standard tools. Unless the leak detection technique involves changing the garbage collector (which traverses the object graph in the heap) heap dump analysis is the easiest way to discover the reference chain from the leaking object to garbage collection roots. Although by default JVM provides means for creating heap dumps, state capture used in the work reviewed take different forms: full heap dump [Maxw 10], partial heap dump [Mitc 03] or custom reference dump. Visualization techniques (Section 3.2.2) also use heap dumps either to apply distinct visualization syntax ([De P 99]) or analyze heap regions for memory problems ([Reis 09]).

The first tool which must be mentioned is Eclipse MAT [The 11] – a production quality open-source tool for heap dump analysis and exploration. Among others, it also contains a function to find dominators and the largest objects, which greatly helps in manual leak analysis.

The next tool specifically focusing on partial heap dump analysis for memory leak detection, is *LeakBot*, described by Mitchell and Sevitsky in [Mitc 03]. The distinctive feature of the approach is that it observes leaks on a data structure level, rather than viewing single objects, and it performs partial dump analysis to monitor structure evolution.

On a high level, LeakBot operates in three steps. First, LeakBot automatically ranks data structures by their likelihood of containing leaks. This is achieved by taking a number of snapshots and finding growing data structures using ranking. Ranking of the candidates is also performed in three steps, where each next step is more expensive to perform, but it operates on a smaller set of candidates. The first step uses eight *binary metrics* divided into structural and temporal metrics, which can rule out many objects using cheap metrics, e.g. observation that an array in Java cannot grow and thus cannot be a leak root. The second ranking step is based on *mixture metrics*, which depend on the particular data structure and include attributes like structure size or reference structure between objects. *Gating functions* combine mixture metrics depending on the context to refine ranking. The third ranking step is called *iterative fixpoint ranking*, which analyzes the remaining candidates from the point of view of domination (see the definition of a dominator below). As a result, the first step prunes the set of candidate structures using object reference graph properties and predefined knowledge of how leaks can occur.

In the second step, LeakBot uses *co-evolving regions* – regions as big as possible, but evolving in a coherent way – to identify suspicious regions within a data structure dominated by a leak root and to characterize their evolution.

In the third step, LeakBot uses results from the first two steps to monitor evolution of the structures by taking and comparing partial heap dumps containing suspected structures. The third step performs evolutionary tracing, which in a loop refines rankings generated by the first two steps.

LeakBot is implemented as a JVMPI agent[1].

Maxwell et al. in [Maxw 10] described how to apply graph mining algorithms to find memory leaks in heap dumps. An important contribution of the work is that the object graph stored in the dump is preprocessed to build a dominator tree of the objects and then this dominator tree is mined for recurring sub-graphs.

In graph theory, a *dominator relationship* is defined as follows: "A node x dominates a node y in a directed graph iff all paths to the node y must pass through the node x" [Maxw 10]. In application to memory leak detection the dominator will be the single object responsible for keeping reachable an entire subgraph of leaking objects, e.g., a collection object holding all leaked objects.

[1]JVMPI is a profiling interface for Java which has been deprecated in favor of JVMTI

To compute the dominator tree an implementation of the Lengauer-Tarjan algorithm [Leng 79] from Boost library [Boos] was used.

3.2.2 Visualization

Visualization approaches are used to help programmers visualize the heap in a way that could help them easily spot memory regions with the most objects having the largest volume.

The first approach to cover is a work from 1999 by DePauw and Sevitsky analyzing the visualization of reference patterns implemented in JInsight[1] ([De P 99]).

The main contribution of the approach is the creation of visual syntax to order graph nodes (representing objects) based on their age and also colorizing them according to age. To reduce the number of elements on the screen, *reference patterns* are extracted and visualized instead of all objects. Reference patterns are intended to eliminate repetitive reference sequences, effectively eliminating redundancy. Generated visual representation allows interactive exploration.

The reason to create such a visual syntax is an observation that memory leaks occur during well-defined operations (e.g., user interaction) which are supposed to release all temporary objects. Thus, if a programmer specified the time boundaries of such operations, then the tool can find objects that were created during that operation but cannot be freed after the operation has been completed, as the reference to the expected temporary object has outlived the operation itself and probably leaked. To specify such time boundaries the programmer has to take two heap snapshots with the JInsight tool – before and after the operation. Snapshots are constructed so that unreachable objects are not dumped. Next, snapshots are compared and the difference – objects created in the meantime and still reachable –

[1]JInsight is a now abandoned profiling and visualization tool developed by IBM research.

is visualized using the visual syntax described earlier. Along with leaked parts, the paths to GC roots are also visualized.

The next attempt to use visualization to aid memory leak detection was described ten years later, in 2009 by Reiss in [Reis 09]. In his work, Reiss applies visualization techniques not only to find memory leaks, but also to detect Java memory problems in general. In addition to memory leaks, three more problems are addressed: inefficient use of memory, churn, and unexpected increases in memory size. Inefficient use of memory is characterized by several levels of intermediate objects between the code using the data and the data itself, especially if the data itself is small, but all intermediate objects occupy more space than the data itself. Churn describes a situation wherein the program is creating a lot of short-lived objects, thus creating a lot of allocation and garbage collection overhead[1]. So, leak detection is only a possible side effect of visualizing the Java heap focused on an object subgraph size.

The idea is to visualize the object ownership graph in a way that is easy to read and grasp by utilizing shapes, coloring, hatching, hue and saturation. The data to achieve this is gathered by periodically traversing the heap, aggregating space used by object hierarchies, and visualizing the object graph as a tree focused on the size of objects.

3.2.3 Static analysis

Static analysis of the source code to detect memory leaks is very popular for languages with manual memory management, like C or C++. In languages with garbage collection, like Java, however, it can be very expensive to statically determine object reachability, particularly in large applications. For this reason, proposed static analysis approaches, which also have a

[1]It must be noted that modern garbage collectors are so efficient that a recommended practice by JVM vendors is to create a lot of short-lived objects, as collection of young heap space is much more efficient than full collections covering old generations, where long-lived objects are finally evacuated.

reported implementation, can handle only certain conditions, like array leaks or objects escaping from loops.

In 2010, Distefano and Filipovic in [Dist 10] described *bi-abductive inference* as an approach for static analysis of the Java code for memory leak detection. Bi-abductive inference is a process of analyzing the data flow in the application in two directions – along the flow of the application and backwards. During first pass forward, structures that are allocated along the flow of the application are identified. During the second pass backwards, it is examined which allocated structures are actually needed for the execution. Next, information from both passes is combined to detect which objects are allocated but unused by the subsequent code. Unfortunately, the paper does not contain any implementation details or case studies of real leaks.

The state of the art in the application of static analysis for memory leak detection is represented by *LeakChecker* by Dacong, Xu et al. [Yan 14]. The cornerstone of the method is the observation that the most severe leaks are caused by events that happen often. Usually the parts of the code that are executed the most are loops. Thus, LeakChecker analyzes each *important* loop in a program and detects the objects that escape one iteration of the loop and never flow back into any later iteration from the memory locations to which they escape [Yan 14]. *Important* loops must be defined by the programmer and may also include code, which, although is executed often, is not represented as a loop structure. Examples of such code may include web request handlers or code, which is invoked by component frameworks.

LeakChecker's average false positive rate is reported to be 49.8%;

3.3 Hybrid methods

Hybrid methods combine both online and offline analysis phases to achieve the best results; however, they may require user interaction to help analyze intermediate results or prepare data for the next steps.

In 2000, Shaham et al. has published a paper about the automatic removal of array memory leaks in Java [Shah 00]. It handles one specific type of leaks: objects, which are no longer used by the application, but are referenced from an array. An array leak is defined as the region of an array, which contains references to objects that are never used. As an example the stack implementation is brought, in which the stack is implemented as an array and the implementation maintains a pointer to the top of the stack as an index within the array. When `pop` operation returns the top of the stack, it returns the object and decrements the stack pointer, but does not clear the reference in the array. This way the array keeps the unneeded reference and prevents the object from being collected.

The approach described performs static analysis of the bytecode of Java classes to detect such array leaks. Next, it sets a flag on a class so that the garbage collector can reclaim unused regions in an array. This last feature makes this otherwise static approach a hybrid approach, because static offline analysis creates an input for the garbage collector to use during runtime.

In 2011 Xu et al. in [Xu 11] described *LeakChaser* – a three-step iterative profiling methodology to find causes of memory leaks in Java applications.

Leak detection is presented as a three-step iterative process. Steps are called *tiers* or *levels*: Tier H (High), M (Medium) and L (Low). The main idea is to split the application into user-level transactions and perform analysis in terms of these transactions. One of the ideas behind this profiling technique is that each level requires a different amount of knowledge about the system from the programmer. On the high level, the tool tries to infer transaction-local and shared objects by itself, requiring almost no knowledge about the system. On the medium level, the programmer has to specify transaction boundaries and shared objects by annotating the code. On the low level, the user can specify per-object lifetime invariants. Moreover, layers, while expecting a different amount of familiarity with the system, allow the programmer to become more familiar with the system while iterating through the levels.

The memory leak detection process using LeakChaser's three tiers is about specifying and asserting which object should escape the previously mentioned user transactions' boundaries (shared objects). The lower the tier, the more precisely it is possible to specify the boundaries down to the assertions that object a should be freed before object b.

LeakChaser was implemented in Jikes RVM by adding following changes to the JVM: The object header was modified to include allocation site information, the previously mentioned assertion checks and management of the *assertion table* were added to garbage collectors. However, as noted in [Xu 11], the described implementation was applicable only to non-generational garbage collectors, because partial heap scans performed by generational collectors may produce both false positives and negatives.

An alternative general-level approach to handle loitering objects instead of swapping them out is to use weak references instead of strong references, which will allow the default garbage collector to reclaim loitering objects only if they are referred by a weak reference. The idea is described by Brian Goetz in [Goet 05], which is cited by several works in the field. The main question of such an approach is how to find references eligible for such substitution?

Quian et al. approaches this question by describing the technique of inferring weak references to fix Java memory leaks in [Qian 12].

The general work flow of the method can be described as follows: to perform analysis on which references can be weakened (described below), then calculate occupied size, rank potential weakening candidates by size, and report the results to the programmer. Based on the report, the programmer can choose whether to use weak references instead of strong references.

Weakening analysis is composed of three steps: During the first step the program's execution trace is recorded. The execution trace contains all reference mutations of all objects' incoming references. This is required for the second step – offline analysis and creation of reference snapshots for each object to detect references which can be weakened. Third step instruments references that can be weakened to collect the object sizes for ranking. References that can be weakened are at first assumed to be all valid for weakening, next they are reviewed using following three rejection rules (cited from [Qian 12]):

1. For a still-in-use object, if there is only one reference to it, then the single reference should not be weakened;
2. For a still-in-use object, if there are multiple references to it, but all these references are instances of the same class field, then the class field should not be weakened;
3. For a still-in-use object, if there are multiple references to it, then at least one of them should not be weakened.

Whether an object is still in use is identified by inspection of the execution trace – whether the reference is used or not.

The approach described can be useful in ranking the contents of the cache to see which cache items are actually used and which are not.

Unfortunately, the publication does not contain a performance analysis of the method, although it would be interesting to see such an analysis, because recording of the mutation trace alone can be very resource-consuming task. It is also not clear whether it is possible to use such a method anywhere except the development environment, because of the performance overhead.

3.4 Comparison of the approaches

Subsection 3.4.1 compares the metrics which are observed by the reviewed methods. Subsection 3.4.2 examines the methods reviewed from the point of view of performance. The performance aspects in focus include both runtime performance and leak detection performance. The main goal of such comparison is to see which methodologies are used across the works to assess performance and whether the results are comparable. Next, Subsection 3.4.3 classifies approaches based on intrusiveness.

3.4.1 Observed metrics

Table 3.1 summarizes and compares discussed state-of-the-art approaches based on metrics which these methods collect, monitor and evaluate in order to detect the leaks. Criteria are elaborated using the following classification.

Generality – defines whether a method is general or limited within some predefined package, or whether it is designed to work only with certain types (arrays or containers) or is applied only to objects, which are previously annotated or otherwise marked by a programmer. Allowing memory leaks to be searched only within a certain package or handle annotated classes imposes most of the constraints in the work reviewed.

Object creation – a metric available only for online methods, indicating that the leak detection method somehow uses an object creation event as a part of metrics used for memory leak detection.

Object destruction – a metric available only for online methods, indicating that the leak detection method somehow uses an object destruction event as a part of metrics used for memory leak detection.

Size – methods using the size of the object (or object subgraph). Size is widely used in several methods. Some methods monitor constant size growth, some relate total memory growth with data structure growth, some use size as a threshold, some use size for ranking leak candidates, and visualization methods use size to distinguish objects visually.

Object access – some methods monitor object access. Accessing an object means that it is not stale, so this property shows that the object is in use and is not a leak. If not implemented inside JVM, this metric is very expensive as for each read access one write operation must be performed.

Time – several approaches are accounting time in some form. However, few of the approaches do it in the same way. A time classifier can be used to drive a staleness counter, record times of creation and destruction of objects (either to record lifetime or just track birth times) or be the source of object lifetime constraints and assertions. Time accounting also varies between methods ranging from wall clock time recording (e.g., time of taking a heap dump) to counting the number of garbage collections passed since the start of the application.

Capture of the state – several methods rely on analysis of the state kept in memory. State may be captured in several ways, ranging from full heap dump to only selected references or data structures. One or another way to capture memory state and analyze it offline is performed by most of the approaches. Memory may be captured either once or periodically to analyze heap evolution offline. *Object swapping* means that some objects are swapped out from the heap to the disk.

Table 3.1: Metrics, observed by memory leak detection techniques

Approach	**Generality**	**Object creation**	**Object destruction**	**Size**	**Object access**	**Time**	**State capture**
Online, Staleness							
Sleigh (Bell) [Bond 06]	General	-	-	-	Last used site	Staleness counter	-
Object ownership profiling [Rays 06, Rays 07]	General	Yes	Yes	Growth	Field access Method call	creation destruction	-
Container profiling [Xu 08, Xu 13]	Containers	-	-	Total memory, Container size			-
LeakSurvivor [Tang 08]	General	-	-	-	-	-	Object swapping
Melt [Bond 08]	General	-	-	-	-	Staleness counter	Object swapping
Leak pruning [Bond 09]	General	-	-	-	-	Staleness counter	-
Online, Growth analysis							
FindLeaks [Chen 07]	Selected packages	Yes	Yes	-	-	-	-
Cork [Jump 06, Jump 07]	General	-	-	Growth	-	-	Heap dump
Statistical approach [vSor 14b, vSor 11a, vSor 11b]	General	Yes	Yes	Threshold	-	As GC generations	Reference dump
Panacea [Brei 07]	Annotated objects	-	-	-	-	-	Object swapping
LoiteringObjectsHealer [Gold 07]	Annotated objects	-	-	-	-	-	Object swapping
Offline, analysis of captured state							
LeakBot [Mitc 03]	General	-	-	-	-	-	Periodical heap dumps
Graph mining on heap dumps [Maxw 10]	General	-	-	-	-	-	Heap dump
Offline, Visualization							
Reference patterns visualization [De P 99]	General	-	-	Size	-	Time of dumps	Periodical heap dumps
Heap visualization [Reis 09]	General	-	-	Size	-	-	Heap dump
Offline, Static analysis							
Bi-abduction [Dist 10]	General	-	-	-	-	-	-
LeakChecker [Yan 14]	Loops	-	-	-	-	-	-
Hybrid							
LeakChaser [Xu 11]	General	Yes	Yes	-	-	Lifetime assertions	-
Array Memory Leaks removal [Shah 00]	Arrays	-	-	-	-	-	-
Weakening inference [Qian 12]	General	-	-	Size	Reference mutations	-	Periodical reference snapshots

3.4.2 Performance evaluation methodologies

An important attribute of any memory leak detection approach is its performance evaluation. Such evaluation should cover leak detection precision, false positive rate and performance overhead in terms of CPU time and memory consumption. Although for offline methods and visualization techniques runtime performance overhead does not apply, leak detection precision still matters (except for visualization techniques, where a human carries out the leak detection).

Reported leaks in open source software are most often used as a detection performance measurement. Such open source software includes Eclipse (several different leaks), Alloy IDE, JDK, ArgoUML, RSSOwl, ActiveMQ, HtmlUnit, Jigsaw, Delaunay, Mckoi and the Drools framework. Leaks in SpecJBB and SpecJVM, which are closed source but are used for benchmarking, are also used to evaluate leak detection.

Performance overhead measurement across different approaches can be summarized as follows: The most popular approaches to measuring performance overhead are to use existing performance benchmarking harnesses – SpecJVM and DaCapo benchmarks. Some approaches measure performance overhead using the same benchmarks which are used for assessing leak detection quality. The interpretation of the measurement results and approaches for experiment setup, however, vary across different authors – number of runs of benchmarks varies and the reported numbers for the benchmarks also vary. Some report best times, while some report average times. To conclude the comparison of the performance measurement approaches it must be noted that it is hard to compare methods objectively based on their performance overhead metrics, as there is no common measurement approach.

Tables 3.2 and 3.3 summarize tools and methods used to measure leak detection performance and performance overhead. In these tables, 'N/A' stands for 'Not Applicable' and '—' stands for 'not described in the paper.'

As measurement of the runtime performance overhead is not applicable to offline methods, static analysis and visualization, the table 3.3 does not include following methods: LeakBot [Mitc 03], Graph mining on heap dumps [Maxw 10], Reference patterns visualization [De P 99], Heap visualization [Reis 09], Array Memory Leaks removal [Shah 00], Bi-Abduction [Dist 10] and LeakChecker [Yan 14].

Table 3.2: Summary of leak detection quality assessment approaches

Method	**Leak detection performance evaluation**
Sleigh (Bell) [Bond 06]	Known leaks in SPEC JBB2000 and Eclipse
Object ownership profiling [Rays 06, Rays 07]	Case studies: known leak in Alloy IDE V3
Container profiling [Xu 08, Xu 13]	Known leak in SpecJBB 2000, JDK bugs #6209673 and #6559589 (leaks in swing and awt)
FindLeaks [Chen 07]	Known leaks in ArgoUML and RSSOwl
Cork [Jump 06, Jump 07]	Leaks in SPECjbb2000, Eclipse 3.1.2, fop, jess (from SpecJVM)
Statistical approach [vSor 14b, vSor 11a, vSor 11b]	Known leaks in ActiveMQ, htmlunit, custom web-application
Panacea [Brei 07]	N/A
Loitering Objects Healer [Gold 07]	N/A
LeakSurvivor [Tang 08]	Eclipse, SPECjbb2000, Jigsaw
Melt [Bond 08]	Time to crash is measured with leaks in EclipseDiff, EclipseCP, JbbMod, ListLeak, SwapLeak, MySQL client, Delaunay, SPECjbb2000, DualLeak, Mckoi. 5 out of 10 leaks are tolerated, until disk space is exhausted.
Leak pruning [Bond 09]	Time to crash is measured with leaks in EclipseDiff, EclipseCP, JbbMod, ListLeak, SwapLeak, MySQL client, Delaunay, SPECjbb2000, DualLeak, Mckoi. 5 out of 10 leaks are tolerated, until disk space is exhausted.
Weakening inference [Qian 12]	—
LeakBot [Mitc 03]	—
Graph mining on heap dumps [Maxw 10]	Synthetic leaks, Existing J2EE application, MVEL parser of Drools framework
Reference patterns visualization [De P 99]	N/A
Heap visualization [Reis 09]	N/A
Array Memory Leaks removal [Shah 00]	—
Bi-Abduction [Dist 10]	—
LeakChaser [Xu 11]	Diff (Eclipse bug #115789), Editor (Eclipse bug #139465), SPECjbb2000, MySQL leak, Mckoi
LeakChecker [Yan 14]	SPECjbb2000, Eclipse Diff, Mckoi, log4j, FindBugs, Derby

Table 3.3: Summary of runtime performance overhead measurement methodologies

Method	**Space overhead**	**Time overhead**	**Measurement methodology**	**Performance benchmarks**
Sleigh (Bell) [Bond 06]	None, uses unused bits in object header in Jikes RVM	Varies depending on configuration	5 benchmark runs, different Sleigh configurations (different features enabled), minimal run time per configuration is reported	SPEC JVM98, DaCapo benchmarks (beta050224), fixed-workload version of SPEC JBB2000
Object ownership profiling [Rays 06, Rays 07]	"Trace collection roughly doubles the memory required to run the program" "The trace analyzer requires between one and two gigabytes of memory for every million objects in the trace"([7] p5),	Our current trace collection implementation slows the target program down by a few orders of magnitude (p5)	—	—
Container profiling [Xu 08, Xu 13]	—	Varies 0.7% .. 313.2%, average 88.5%	Dacapo suite and 3 leak test applications were run once with two different memory settings.	Selection of apps from DaCapo suite and 3 apps used for leak detection analysis.
FindLeaks [Chen 07]	—	—	Future work	Future work
Cork [Jump 06, Jump 07]	0.145% on average and never more than 0.5% of total heap	Average overhead in a generational collector is 11.1% .. 13.2% for scan time; 12.3% .. 14.9% for collector time; 1.9% .. 4.0% for total time.	Several configurations (different heap sizes), each configuration ran 5 times, best run is reported. Geometric mean across recorded measurements of all benchmarks is reported as a general measure. GC-only performance is evaluated separately	DaCapo b.050224, SPECJvm98

Table 3.3 – *Continued from previous page*

Method	**Space overhead**	**Time overhead**	**Measurement methodology**	**Performance benchmarks**
Statistical approach [vSor 14b, vSor 11a, vSor 11b]	—	Up to 35%	Memory usage and application uptime of applications with leaks measured with and without leak detection. One run of each benchmark in 2 configurations.	Active MQ, HtmlUnit, Custom Java EE web application
Panacea [Brei 07]	—	”overhead introduced by PANACEA upon creation of new objects is around 15%, amortized over time” (p10)	Measure performance of the healers, which handle only manually annotated objects. Only average measurements are reported. ”Repeating these sequences for statistically sufficient number of times”	Custom designed benchmarks
Loitering Objects Healer [Gold 07]	—	Up to 300%	Healing time and response time are measured, 3 configurations, averages are reported	—
LeakSurvivor [Tang 08]	Occurs only when leaks need swapping.	23.7% (Sleigh + LeakSurvivor), LeakSurvivor adds 2,5% on top of Sleigh	Applications with leaks (Eclipse, SpecJBB and Jigsaw), needing swap-out, and application without leaks (Dacapo benchmarks) are measured separately. 3 configurations are used: without LeakSuvivor and Sleigh, with Sleigh only, with both Sleigh and LeakSurvivor.	Eclipse, SpecJBB, Jigsaw, Dacapo
Melt [Bond 08]	—	6% on average	Configurations: 1.5x, 2x, 3x, and 5x the minimum heap size for each DaCapo benchmark. /Replay/ methodology is used to get rid of non-determinism introduced by adaptive compilation (analogue of hotspot compiler in Jikes RVM). Median of 5 runs along with deviation is reported. Application time with and without melt, GC times in different configurations and compilation overhead are measured.	DaCapo benchmarks version 2006-10-MR1, a fixed-workload version of SPECjbb2000 called pseudojbb, and SPECjvm98

Table 3.3 – *Continued from previous page*

Method	**Space overhead**	**Time overhead**	**Measurement methodology**	**Performance benchmarks**
Leak pruning [Bond 09]	—	Application overhead - 3%..5%, Compilation overhead - 17%..34%, GC overhead depend on heap and GC configuration.	Replay compilation of Jikes RVM is used to handle hotspot compilation non-determinism,	DaCapo benchmarks version 2006-10-MR1, a fixed- workload version of SPECjbb2000 called pseudojbb, and SPECjvm98
Weakening inference [Qian 12]	—	—	Future work	—
LeakChaser [Xu 11]	below 10%	2.3 times on average for all benchmarks	1 benchmark run without LeakChaser and 1 run with leak chaser, Geomean overhead for all benchmarks is reported	SPECjvm98 and DaCapo

3.4.3 Intrusiveness

In the case of online methods, which monitor execution of the application, in addition to the performance overhead, it is also important how complex the deployment of the method is.

Each of the methods needs a way to gather data on which to base the leak detection. Data required for the leak detection may come from different sources, all of which require different levels of effort to collect or implement collection of such data in an efficient way. Selection of the data collection source dictates how hard it is to deploy one or another leak detection approach.

Although offline methods analyze or visualize either source code or some kind of captured state, they may require some modifications to the configuration (e.g., attach a Java agent) to obtain the required information.

We define the *intrusiveness* of the leak detection methods as a special criterion, which describes how much effort is required in order to use the implementation of the proposed method to find the leak. The following scale of intrusiveness is proposed:

Low. No code modifications are needed; the method can be implemented in a way such that deployment requires either start script modification, or it can be deployed into the running application. Implicit specification of parts of the application to monitor may be required. In case of state analysis or offline analysis, the data can be acquired using bundled tools with low intrusion.

Medium. Changes to the application code or the configuration are required (e.g., adding annotations). Modifications to the application code, even minor ones, require the application to be rebuilt and redeployed. In some environments building and redeploying an application requires a lot of time and an effort of several people, which makes even a tiny change to the source code to become a barrier for the use of a method.

High. The JVM, the garbage collector or the environment requires modification. It is obvious, that modification of such important infrastructural component, as it is a Java virtual machine, is not a trivial task for any kind of application. Of course, once the method will be incorporated in a real production-ready Java virtual machine, like HotSpot or J9, the use of such method becomes trivial. However, getting the implementation of a memory leak detection method into production JVM is even more non-trivial. The problem with methods modifying the JVM or garbage collector to take advantage of 'unused bits' in the object header is that in actual JVMs there is no such concept as 'unused bits' in the object header. For example, in HotSpot JVM 'unused' bits in an object header are used and shared by several subsystems – garbage collector itself (to count generations to decide promotion from survivors to tenured generation), performance optimizations for locking and a place to keep the identity hash code, which all use the same bits in the header, effectively interfering with each other.

Six deployment methods were identified in reviewed memory leak detection approaches and are listed below. Along with the description of the deployment the estimation of the intrusiveness is given based on previous definitions.

1. Garbage collector (GC) modifications. Although the garbage collector seems to be the best tool to integrate leak detection algorithms or at least make available some special counters, it is also one of the most complex parts of a JVM and is crucial for the performance of the whole virtual machine. As explained earlier, very intrusive, though low overhead approach.

2. Java virtual machine (JVM) modifications. Some leak detection approaches require more modifications besides the garbage collector to

be able to track object instantiation-related events or store some additional data in the object header. Very intrusive approach.

3. *Application* modifications to include special annotations or changes for the leak detection method to work. Medium intrusion.

4. A method relying on *aspect-oriented programming (AOP)* to add monitoring and analysis pointcuts to the code can be either low or medium intrusive.

5. A method can be implemented using bytecode modification with the help of an *agent* (either JVMTI or Java agent). Low intrusion, can be added with a single start parameter.

6. Analysis of the *state* captured using standard tools, such as heap dumps. Heap dumps and other sorts of state capture can be obtained from the running application without any modification of the running code, making these low intrusive methods.

Table 3.4 summarizes the intrusiveness of all approaches reviewed in previously defined terms.

3.4.4 Discussion

In previous subsections, memory leak detection methods were reviewed from several different perspectives: observed metrics, performance overhead, evaluation methodologies, and intrusiveness.

The observed metrics largely define the quality of the leak detection. Most relevant metrics, like staleness of objects or size growth patterns, can provide the best leak detection results, so that little additional manual labor might be required to identify the leak. However, the quality of such metrics comes at the expense of performance overhead and intrusiveness.

From the point of view of performance and intrusiveness, the best way would be a formal verification of the source code. Unfortunately, this is very

Table 3.4: Comparison of leak detection intrusiveness

Approach	Year	GC	JVM	App.	Agent	AOP	State	Intrusion
Online, Staleness analysis								
Sleigh (Bell) [Bond 06]	2006	✓	✓	-	-	-	-	High
Object ownership profiling [Rays 06, Rays 07]	2006	-	-	-	✓	-	-	Low
Container profiling [Xu 08, Xu 13]	2008	-	-	✓	✓	-	-	Medium
LeakSurvivor [Tang 08]	2008	✓	✓	-	-	-	-	High
Melt [Bond 08]	2008	✓	✓	-	-	-	-	High
Leak pruning [Bond 09]	2009	✓	✓	-	-	-	-	High
Online, Growth analysis								
FindLeaks [Chen 07]	2007	-	-	-	-	✓	-	Medium
Cork [Jump 07, Jump 06]	2006	✓	-	-	-	-	✓	High
Statistical approach [vSor 14b, vSor 11a, vSor 11b]	2011	-	-	-	✓	-	✓	Low
Panacea [Brei 07]	2007	-	-	✓	✓	-	-	Medium
LoiteringObjectsHealer [Gold 07]	2007	-	-	✓	✓	-	-	Medium
Offline, Analysis of captured state								
LeakBot [Mitc 03]	2003	-	-	-	✓	-	✓	Low
Graph mining on heap dumps [Maxw 10]	2010	-	-	-	-	-	✓	Low
Offline, Visualization								
Reference patterns visualization [De P 99]	1999	-	-	-	-	-	✓	Low
Heap visualization [Reis 09]	2009	-	-	-	✓	-	-	Low
Offline, Static analysis								
Bi-Abductive inference [Dist 10]	2011	-	-	-	-	-	-	Low
LeakChecker [Yan 14]	2014	-	-	-	-	-	-	Low
Hybrid								
Array Memory Leaks removal [Shah 00]	2000	✓	-	-	-	-	-	High
LeakChaser [Xu 11]	2011	✓	✓	✓	-	-	-	High
Weakening inference [Qian 12]	2012	-	-	-	✓	-	✓	Low

hard in practice because of the dynamic nature and non-determinism of the reachability graph at any particular moment introduced by the garbage collector. This can be observed from the static analysis methods, of which the state of the art covers only a very limited set of features.

The next preference from the performance standpoint is the state capture, which is second best to the low-intrusive approaches. It seems that heap dump analysis tools are the most popular for manual leak detection in everyday use. However, state capture presents only a static image of the state, leaving out any temporal or dynamic details, which are important to separate true and false positives. The offline approaches observed provide good information for the developer, but he must infer details of the application behavior manually by inspecting the source code.

Simplicity (in terms of performance overhead and intrusiveness) of analysis of the state capture can be seen in the number of tools that are available for manual heap dump analysis and visualization, starting with specialized tools like Eclipse Memory Analyzer Tool, *jhat* or *jvisualvm* (part of the Oracle JDK and OpenJdk), and other tools provided by the JVM vendors and commercial profilers (e.g., YourKit, JProfiler, etc.).

The only methods that try to perform fully automatic leak detection are online methods, which observe both the state and its dynamic properties. From the quality point of view they are the best, but from the performance overhead and intrusiveness point of view they are the worst.

Therefore, selection of the method to be used should be based on the balance and trade-offs between quality, performance and intrusiveness.

3.5 Summary

This chapter continues the state of the art review from by reviewing existing approaches for memory leak detection in Java applications. The classification of the approaches was proposed, separating them into online, offline and hybrid approaches with further sub-categories. Different aspects of the state of the art methods were compared, including observed metrics, runtime and leak detection performance evaluation methodologies, and implementation and deployment effort, or intrusiveness.

Chapter 4

Statistical Approach for Memory Leak Detection

The current chapter describes the statistical approach for memory leak detection in detail. Section 4.1 covers background information about general concepts needed to understand the method. Subsection 4.2.1 gives a formal definition of the methods along with necessary definitions. Subsection 4.2.2 describes how the method can be implemented in the real world and subsection 4.2.3 describes how detected leaks can be reported in an efficient way. A detailed analysis and case studies will follow in section 4.3 and chapter 6 respectively.

4.1 Background

Before the method can be addressed, a *memory leak* must be defined in order to gain a common understanding. An object that is no longer needed for the application, but cannot be garbage-collected, is said to be leaked. However, a single leaked object is usually not a problem, unless the object occupies a significant amount of memory. Much more dangerous is a situation where objects are leaked constantly, over and over again. This causes the leak to grow until it fills the entire available heap and the JVM runs out of memory. In the context of this thesis a definition of a memory leak is narrowed as a set of leaking objects accumulating over time.

The variety of types of objects in a typical application is limited by the number of classes in the application[1]. The number of types of simultaneously leaking objects is in turn a subset of all classes of the application and usually this subset is rather small. So, to track leaking objects we may group them by their class, to observe common metrics. However, it is obvious that observing objects based only on their class will produce too many false alarms because very common classes, like String or Integer, are used in too many places and are often leaked as well. So, observing further properties on a class level is too coarse. To mitigate this, instead of gathering statistics (which will be described in further detail) on a class level, we gather statistics with the granularity of an *allocation site*. This means that objects of the same class, but created in different places in the code, are treated as different classes. Experiments showed that such approach yields good measures of true positives (real leaks) and excludes a lot of false positives (non-leaks identified to be leaks by the algorithm). Further in the text, we will use the term *allocation* to denote instances of a class created at a particular allocation site.

As was described in chapter 2, the driving force behind generational garbage collection is the weak generational hypothesis, which states that most objects die young, so the heap can be separated into regions, or *generations*, which will hold objects based on their age and each of these regions will be cleaned by a garbage collector of the appropriate type to achieve the best performance. Statistical approach for memory leak detection in Java applications proposes to use the same weak generational hypothesis but for memory leak detection. Following sections describe how.

[1]In the case of dynamic class generation, there is a possibility that an application may have an unlimited number of classes. This scenario is not considered in the thesis, although unbounded number of classes being generated may be seen as a special case of a memory leak in regard to the current definition.

4.2 Statistical approach for memory leak detection

4.2.1 Definitions

Let garbage collections in the application be $GC = (gc_1, gc_2, \ldots, gc_n)$. If an object is created between garbage collections gc_i and gc_{i+1}, where $i \geq 1$, it is said to belong to *generation* g_i. If an object is created before gc_1, it is said to belong to g_0.

Let C be the set of all allocation sites in the application. Then, after any garbage collection $gc_j \in GC$, for each allocation $c \in C$, we can define the function $G : (c, gc_j) \to \{0,1\}^j$, such that for $i \in \{0, \ldots, j\}$,

$$G(c, gc_j)_i = \begin{cases} 1 & \begin{array}{l}\text{if after } gc_j \text{ there exists a live object of} \\ \text{allocation } c \text{ created in generation } g_i\end{array} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Using these definitions, we can capture a snapshot of the distribution of ages of live objects of an allocation at any point in time.

Next, let us define a function which, given the vector $G(c, gc_j)$, will indicate how many different generations have live objects of the allocation c after garbage collection gc_j[1]:

$$genCount(G(c, gc_j)) = \sum_{i=0}^{j} G(c, gc_j)_i \tag{4.2}$$

Let us illustrate these definitions with an example of how instances of classes representing a typical web application in Java, implementations of a *Servlet*, *HttpSession* and *HttpRequest*, behave. In figure 4.1 they are marked respectively *Srv*, *Ses* and *Req*.

[1] *genCount* is first described by Formanek and Sporar as 'generation count' in [Form 06]

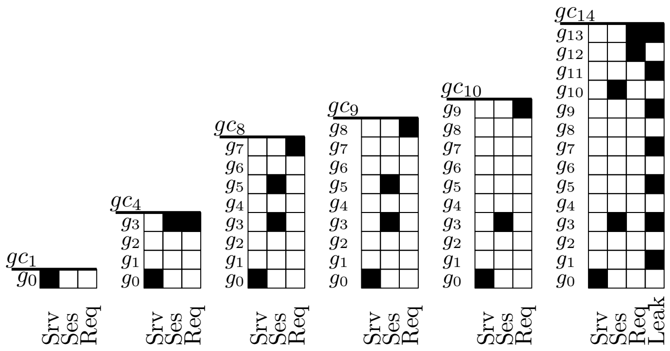


Figure 4.1: Illustration of the *genCount* concept

During initialization of the application, an instance of *Servlet* is created. From Figure 4.1, it can be seen that the instance was created just before the very first garbage collector (gc_1) was run. Thus the instance of *Servlet* belongs to generation g_0. After gc_1 the following values can be calculated:

$$G(Srv, gc_1) = (1),\ genCount(G(Srv, gc_1)) = 1$$
$$G(Ses, gc_1) = (0),\ genCount(G(Ses, gc_1)) = 0$$
$$G(Req, gc_1) = (0),\ genCount(G(Req, gc_1)) = 0$$

After initialization of the application is completed, it can start serving requests. The initialization of the application is finished by garbage collection gc_4 and the first request is served during the gc_4. From Figure 4.1, it can be seen that there are instances of *Session* and *Request* created to serve that first request, and these instances are alive during gc_4. After gc_4 has finished, the following values can be calculated:

$$G(Srv, gc_4) = (1, 0, 0, 0),\ genCount(G(Srv, gc_4)) = 1$$
$$G(Ses, gc_4) = (0, 0, 0, 1),\ genCount(G(Ses, gc_4)) = 1$$
$$G(Req, gc_4) = (0, 0, 0, 1),\ genCount(G(Req, gc_4)) = 1$$

After garbage collection number 8, i.e. gc_8, from Figure 4.1, it can be seen that there are instances of all three classes in the heap, which are reachable, i.e. alive. *Servlet*, which was created in generation g_0 and is expected to be the only instance in the whole application is indeed still alive. Two instances of *Session*, which were created in g_3 and g_5 are reachable after gc_8. A *Request* created in generation g_7 is also reachable after gc_8. So, after gc_8 the following values can be calculated:

$$\begin{aligned}
G(Srv, gc_8) &= (1,0,0,0,0,0,0,0),\ genCount(G(Srv, gc_8)) = 1\\
G(Ses, gc_8) &= (0,0,0,1,0,1,0,0),\ genCount(G(Ses, gc_8)) = 2\\
G(Req, gc_8) &= (0,0,0,0,0,0,0,1),\ genCount(G(Req, gc_8)) = 1
\end{aligned}$$

HTTP requests in a typical web application are usually short-lived objects – they are created when the request comes in and are discarded when the result is rendered to the output stream of a response. HTTP session objects, in contrast, are alive over several requests within the session. Session objects are made unreachable (eligible for collection) after user logs out by specifically signaling the servlet container that the session may be invalidated or when a session timeout occurs, which is configurable using the servlet parameters.

So, after gc_9, the instance of *Servlet* created at g_0 is still alive. Instances of *Session* created at g_3 and g_5 are alive. *Request* from g_7 is now collected, but there is a new *Request* created at g_8, which survived the collection gc_9. So, after gc_9, the following values can be calculated:

$$\begin{aligned}
G(Srv, gc_9) &= (1,0,0,0,0,0,0,0,0),\ genCount(G(Srv, gc_9)) = 1\\
G(Ses, gc_9) &= (0,0,0,1,0,1,0,0,0),\ genCount(G(Ses, gc_9)) = 2\\
G(Req, gc_9) &= (0,0,0,0,0,0,0,0,1),\ genCount(G(Req, gc_9)) = 1
\end{aligned}$$

The *Session* created in g_5 was invalidated due to a timeout or logout action, so it was collected by gc_{10}. However, a *Session* created in g_3 is active and *Requests* are coming in within that session. So, after gc_{10}, the following values can be calculated:

$$\begin{aligned}
G(Srv, gc_{10}) &= (1,0,0,0,0,0,0,0,0,0),\ genCount(G(Srv, gc_{10})) = 1\\
G(Ses, gc_{10}) &= (0,0,0,1,0,0,0,0,0,0),\ genCount(G(Ses, gc_{10})) = 1\\
G(Req, gc_{10}) &= (0,0,0,0,0,0,0,0,0,1),\ genCount(G(Req, gc_{10})) = 1
\end{aligned}$$

Finally, let us fast-forward to after garbage collection gc_{14}. It can be observed that the *Servlet* is still alive and there is still only one instance of it on the heap – the one created in g_0. There is also one active *Session* created back in g_3 and a fresh one created in g_{10}. Instances of *Request* after gc_{14} are alive in two generations – g_{12} and g_{13}. After gc_{14}, the following values can be calculated:

$$\begin{gathered}
G(Srv, gc_{14}) = (1,0,0,0,0,0,0,0,0,0,0,0,0,0), \\
genCount(G(Srv, gc_{14})) = 1 \\
G(Ses, gc_{14}) = (0,0,0,1,0,0,0,0,0,0,1,0,0,0), \\
genCount(G(Ses, gc_{14})) = 2 \\
G(Req, gc_{14}) = (0,0,0,0,0,0,0,0,0,0,0,0,1,1), \\
genCount(G(Req, gc_{14})) = 2
\end{gathered}$$

The intuition behind the statistical method for memory leak detection is as follows. In a healthy application, there are some objects created at the beginning of the application, belong to early generations, and live until the application terminates. These objects are said to be long-lived (like *Servlet* in the previous example). Short-lived objects are created to serve incoming requests, keep a session state, maintain user interaction (like *Session* and *Request* in the previous example), and are kept alive only for several generations. This is also backed up by the weak generational hypothesis. By intuition, a leak is a condition when objects of some class are created regularly at some allocation site, but are not reclaimed by the garbage collector, as they are held by a forgotten reference. Such leaking allocation is illustrated on the Figure 4.1 in the gc_{14} section and is marked as *Leak*.

So, in the case of leaking objects of allocation c, the following holds:

$$\lim_{j \to \infty} genCount(G(c, gc_j)) = \infty \tag{4.3}$$

In the case of well-behaved allocations and taking the weak generational hypothesis into consideration, the following holds:

$$\lim_{j \to \infty} genCount(G(c, gc_j)) \leq M \tag{4.4}$$

Where M is a value dependant to the combination of the application, JVM configuration, usage of the application, the garbage collection algorithm, the environment, etc. The main factor affecting M is how often garbage collection runs – if it runs frequently enough that even short-lived objects survive several collections, then M is larger. Alternatively, if collections run infrequently, then M would be small.

After specifying the initial hypothesis about using *genCount* to detect memory leaks, proof of concept was implemented (described in subsection 4.2.2). Using such proof of concept implementation, we were able to collect statistical data about different applications across different customers. To gather this information, snapshots of G vectors were taken after each full garbage collection in the applications. Based on these snapshots, we were able to analyze real numbers. By the time of the analysis, we had collected $13,851$ snapshots from $1,272$ different customers. These snapshots contained $31,568,128$ allocations in total. The anonymity of the collected data prevented us from distinguishing whether a customer had one or several different applications; however, it was possible to distinguish between different customers. So, to analyze the number of allocations per application, statistical means of the number of allocations within all the applications belonging to one customer were counted. Results for the allocation counts are presented in Table 4.1 and results for *genCount* counts are presented in Table 4.2. While allocation counts show the sizes of the applications we have investigated, *genCount* counts show whether our hypothesis about generally small *genCount* values holds.

Table 4.2 shows that equation 4.4 holds, given the maximum value for *genCount* in examined samples to be $69,335$, whereas the 95th percentile

Min	1
1st Quartile	161
Median	788
Mean	1,691
3rd Quartile	2,124
95th Percentile	6,379
Max	24,427

Table 4.1: Distribution of the number of allocations per application

Min	1
1st Quartile	1
Median	1
Mean	6
3rd Quartile	2
95th Percentile	14
Max	69,335

Table 4.2: Distribution of *genCount* values for allocations

for 31,568,128 observed allocations is 14. Assuming that most of the allocations are not leaking, their *genCount* remains small.

The main challenge of the approach is how to detect the unbounded growth of the *genCount* function for some set of allocations over time, as soon as possible[1], and not to generate too many false positives. The proof of concept solution presented in [vSor 11b] used a simple threshold technique to distinguish between normal allocations and leaking allocations. Algorithm 4.1 expresses the described technique using pseudocode.

In algorithm 4.1 two implied functions are used; the detailed implementation of them is currently not relevant. The first function is *sortByGenCountValue*, which is called on line 7; this function is expected to sort allocations by their *genCount* value in ascending order. The result of such sorting is illustrated by Figure 4.4 in section 4.3.2. The second implied function is the $sublist(C, i, j)$ on line 12, which returns part of the array C, starting at element i until element j.

[1]Detecting the leak as soon as possible is desirable in cases where the existence of the leak is yet unknown. Detecting and reporting the leak as soon as possible will give more time for the operations team to plan for a restart, by paying more attention to heap usage trends, should OutOfMemoryError happen.

Algorithm 4.1: Memory leak detection algorithm

```
1  findLeaks(Allocation[] C) {
2    int[] genCounts
3    foreach Allocation $c \in C$ {
4      genCounts[c] = genCount(c)
5    }
6    //order allocations by their genCount values
7    sortByGenCountValue(genCounts, C)
8    // traverse from large genCounts to small
9    for (int i=genCounts.size(); i>0; i--) {
10     gap = genCounts[i] / genCounts[i-1]
11     if (gap > THRESHOLD) {
12       return sublist(C, i, genCounts.length)
13     }
14   }
15   return []
16 }
```

An important new concept is introduced in algorithm 4.1 on line 10 – *gap*, which is a ratio between the *genCount* values of two neighbors after the allocations are sorted by *genCount*. Operating with ratio rather than difference between neighbors allows for the use of relative values, which allows for the normalization of values across different applications. Selection of the gap threshold must be balanced between reporting too many false positives too early (if the threshold is too small), and reporting too late, thereby leaving no time to perform further analysis and reporting (if the threshold is too large). There are two possibilities for choosing the threshold – constant value and dynamic value specifically tuned for or automatically identified for a particular application. Dynamic selection requires several application runs to determine the threshold, but this is undesirable behavior as we would like to detect the leak before it crashes the application for the first time. Another argument against such a dynamic threshold is that it may become overfitted for some specific usage scenario and fail in a real leak situation. So we chose to use a constant value. It gives good balance in the majority of situations and applies additional pre-checks to verify the leak status in real-world scenarios (e.g., calculate the size of detected leaking objects and whether their size is too small, based on some heuristics, then ignore the leak until it grows bigger). We evaluated different threshold values during testing before releasing Plumbr for general use, and identified that the constant value between 3 and 5 gives good a balance in terms of detection time and false positive rate. It was also confirmed by real cases we analyzed while preparing section 4.3. We will discuss issues with the selection of the threshold in more detail in the analysis of the statistical approach presented in section 4.3.2.

4.2.2 Implementation of the tracking code

Implementation of the automated statistical approach for memory leak detection consists of two JVM agents: the Java agent and the native agent.

The native agent is required because not all required monitoring information is available through Java APIs. Thus, native agent is responsible for gathering low-level information, e.g., garbage collection and object freeing events, which are collected using Java Virtual Machine Tooling Interface (JVMTI, [Sun 06]) callbacks. Responsibilities of the native agent include:

- Marking objects using JVMTI tags (64 bit values, which can be assigned to any object using JVMTI's *SetTag* method). Every tracked object receives a unique identifier, generated by the Java agent. The tag is attached to the object regardless of whether the object was moved by the garbage collector from one generation to another. Tags are stored by the JVM outside of the heap, thereby not impacting the Java heap consumption.

- Listening for JVMTI *ObjectFree* callbacks and notifying the Java agent about objects which were freed by the garbage collector.

- Generating reference dumps (custom data structure, which differs from the HPROF heap dump format which is generated by the JVM). Whereas the heap dump contains all the data in the heap, reference dump contains only minimal reference descriptors (source object identifier, target object identifier, type of the reference, field/array index) needed to find the shortest path from the leaking objects to the GC root. The reference dump also occupies less space because it does not contain the actual data and is optimized to find the shortest paths in the needed direction (from objects to GC roots). Reference dumps are generated on request from the Java agent. The size of the dump is directly dependent on the number of references in the heap.

The native agent provides low-level utility functions for the Java agent, which orchestrates all the work. The Java agent communicates with the native agent using Java Native Interface (JNI, [Lian 99]) method calls.

The Java agent operates with the following flow:

1. Track the allocation of objects using byte code instrumentation – this is implemented by instrumenting allocation sites (places in the code where either the constructor is called, or cloning or serialization is performed). Instrumented code invokes a callback method which registers the object creation event and also the allocation site. Sampling is used to reduce overhead, so not all created objects are tracked. The number of tracked objects is driven by an internal queue, where new objects are placed. The queue is limited in size and when it fills up, it is processed to register new objects in internal structures. New objects are not accepted until the queue is processed. Accumulating objects in the queue uses weak references to maintain the link to the object until the native agent tags it. Such implementation gives an advantage: very short-lived objects can be collected before they become tracked by Plumbr.

2. After each full garbage collection the Java agent analyzes object age distribution using the statistical method for memory leak detection (algorithm 4.1). This analysis is invoked only after full garbage collections, which ensures that tracked statistics do not contain objects, which are located in the old generation, but are already unreachable.

3. If algorithm 4.1 reports a non-empty set of allocations, these allocations are reported as leak candidates.

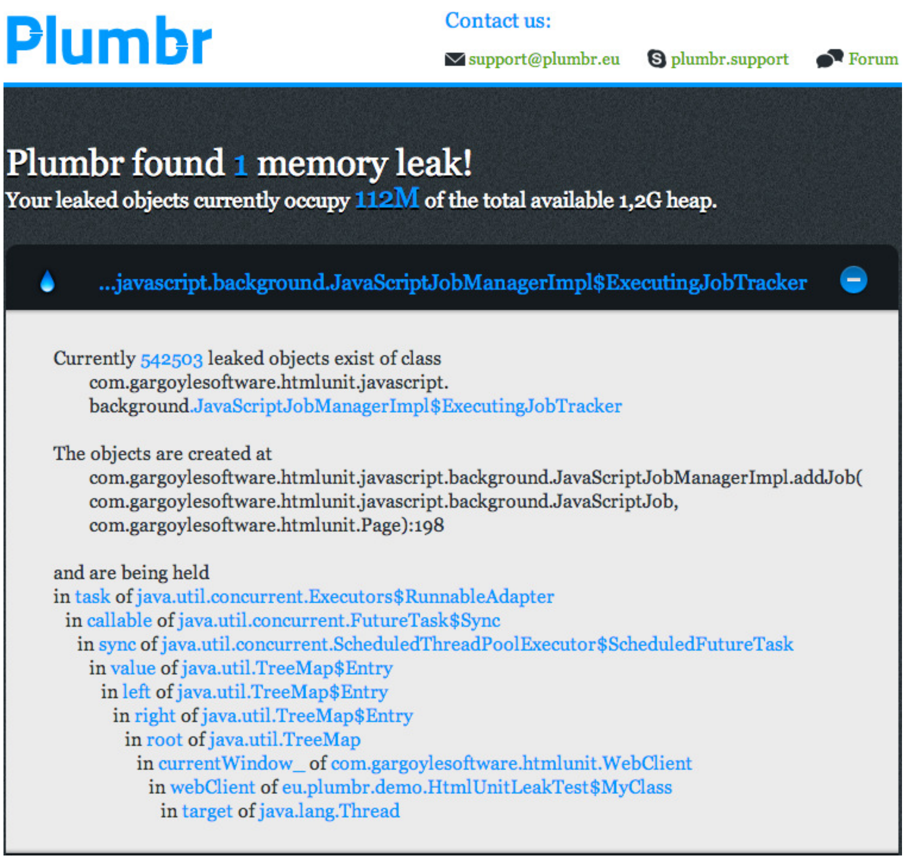
4. If the previous step has identified a set of leak candidates, the reference dump is requested from the native agent and a graph analysis starts discovering paths from leaking objects to garbage collection roots. This analysis is performed in a separate Java process to reduce memory overhead on the host process. It is also important to note that the reference dump is created only after the statistical method has indicated leaking allocations. Discussion of the reference path discovery follows in section 4.2.3.

5. The report is generated. An example leak report is shown in Figure 4.2. The report is generated for the *HtmlUnit* case study presented in detail in section 6.3.

6. At the moment of writing, due to implementation details, after generating first report, Plumbr stops tracking. This is implemented via a quick return from the tracking callback, rather than removing bytecode modifications from the application code.

4.2.3 Resolution of the leaking reference path

After some allocations are reported to be leaking, Plumbr must report the reference chains, which prevent leaking objects from being collected. Dijkstra's algorithm for the shortest path search is used to find the paths



Plumbr found 1 memory leak!
Your leaked objects currently occupy 112M of the total available 1,2G heap.

...javascript.background.JavaScriptJobManagerImpl$ExecutingJobTracker

```
Currently 542503 leaked objects exist of class
    com.gargoylesoftware.htmlunit.javascript.
    background.JavaScriptJobManagerImpl$ExecutingJobTracker

The objects are created at
    com.gargoylesoftware.htmlunit.javascript.background.JavaScriptJobManagerImpl.addJob(
    com.gargoylesoftware.htmlunit.javascript.background.JavaScriptJob,
    com.gargoylesoftware.htmlunit.Page):198

and are being held
in task of java.util.concurrent.Executors$RunnableAdapter
 in callable of java.util.concurrent.FutureTask$Sync
  in sync of java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask
   in value of java.util.TreeMap$Entry
    in left of java.util.TreeMap$Entry
     in right of java.util.TreeMap$Entry
      in root of java.util.TreeMap
       in currentWindow_ of com.gargoylesoftware.htmlunit.WebClient
        in webClient of eu.plumbr.demo.HtmlUnitLeakTest$MyClass
         in target of java.lang.Thread
```

Figure 4.2: Example memory leak report generated by Plumbr

from leaking objects to GC roots. There are some issues which affect the time required to find the path for different kinds of leaks.

As one allocation is responsible for the creation of all the objects which we mark as leaking, the first issue to solve is to select the actual instances from which to perform the search. To keep performance overhead and search time down, we select random objects with an average age (for that particular allocation) as the starting point. Average age is chosen because young objects may still be in active use and referenced by stack variables (the discussion on stack references will follow) and old objects may be some application-wide cached instances. To maintain a balance between precise reporting of all paths and performance overhead, the heuristic of selecting middle-aged instances is chosen.

When the search for the shortest path from the leaking object to the garbage collection root is performed, only certain types of references are followed: fields, static fields and stack local variables. We assume that most of leaks happen via instance fields and static fields (usually, there is a collection object in a field or static field where leaking objects are accumulated). Stack local variables, however, are expected to be very-short lived (they are alive only within a call to a method) and thus, even if a reference path traverses such reference, they are ignored at first. The reason for that is that if there is a leaking collection stored in a field, and at the moment of the reference dump creation, there is a method, which is accessing that collection via stack local variable, then it is desirable to report a path traversing the field, not the local variable (as a local variable is one of the GC roots, it will end the search for the shortest path as soon as encountered). To do that, shortest path search algorithm is trying to reach the GC root via field as the first preference. However, if traversing the reference graph via fields doesn't lead to the root, then stack local variables are followed. However, it takes more time, as all reachable field references must be traversed before stack locals.

Leaks via stack local variables can happen in applications which are working in an infinite loop (see HtmlUnit case study in section 6.3), e.g., polling applications.

Another problem is that objects usually leak in clusters, i.e. there is one composite object, which is actually leaking by being placed in a collection and then forgotten, but all fields of the object are also leaked with the object. If the object graph underlying the leaking object is more complex then even more allocations are reported as leak suspects (which they actually are), but for the final report we must find that one core object that holds all other leaking objects. This is the reason why objects are not reported and analyzed one by one, but are at first gathered and then analyzed as a graph to produce a meaningful report.

To report only a path to such a composite object, merging of reference chains is performed. Figure 4.3 illustrates the concept: let F, E, D and W be detected leaking allocations. Let D be a composite object and one of its fields holds a reference to E, and E in turn references F. The search for the shortest paths will return 4 paths – one for each detected leaking allocation. For allocations E and F it can be seen that their shortest path traverses another leaking allocation, which is located closer to the root D, so paths for allocations E and F are discarded and only allocations D and W are reported.

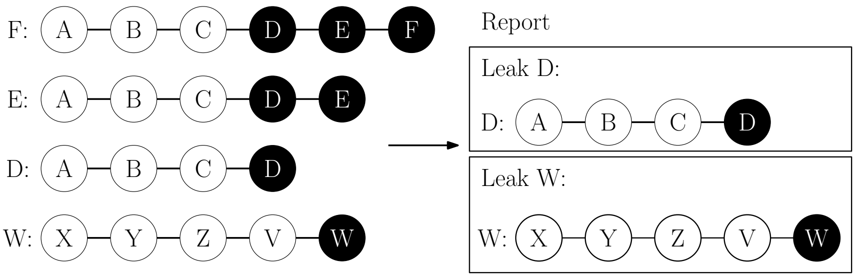


Figure 4.3: Merging of leaking chains

It may happen that an analysis and report generation must be performed in a very memory-limited environment[1] – a memory leak is consuming free heap space on one side and our analysis also requires some heap for the analysis on the other side. This may lead to a situation where JVM runs out of heap before the analysis ends. To mitigate this situation Plumbr stores a *reference dump* file and the accompanying metadata, so that an analysis can also be performed offline. However, some run time information will not be available in the report.

Such a hybrid approach combines the benefits of both online (e.g., profilers which monitor live data and have access to required reflection and

[1]For detailed analysis of embeddable graph manipulation libraries see [vSor 12]

behavior information) and offline (heap dump analysis in a separate environment) memory leak detection approaches. There is no need to wait for the next time leak shows itself, to attempt to repeat the analysis online. Instead, perform the remaining analysis offline, which can be done much faster since such tight memory restrictions do not apply.

4.3 Analysis of the leak detection performance

There are two major perspectives to performance analysis of a tool such as Plumbr – detection quality and runtime performance. Detection quality shows how good is the tool at it's main task – memory leak detection. Runtime performance shows how much performance overhead imposes such a tool for the application being monitored. Current section evaluates both aspects.

4.3.1 Detection performance

Automated memory leak detection is in essence a task of assigning labels 'leaking' and 'not leaking' to allocations, or binary classification of allocations into 'leaking' and 'non-leaking' groups. If at least one allocation is leaking, then there's a leak in the application. Thus, quality metrics applicable to classification algorithms (or classifiers) may be used.

Verified 'right answers' are required to create a confusion table and calculate previously defined metrics for any classification algorithm. In case of memory leak detection the leaking status of the application has to be obtained and verified manually. To manually verify the leaking status, source statistical data which was used by the Plumbr is required. To collect such information Plumbr was modified to collect and serialize the data used by the detection algorithm for offline (outside of a running application) analysis. Such serialized statistical data contained information about live allocations and their G vectors (defined in equation 4.1). Further such serialized data will be called *statistical snapshots*.

Such snapshots were created each time before detection would run, i.e., after each full garbage collection, when G vectors are updated with collected allocations. Tracking code of Plumbr was also modified in a way that it performed the detection based on the same serialized data, so that both runtime and offline analyses would base their decisions on the same data. Statistic snapshots were captured for each application execution separately. In offline snapshots were grouped by the user and the application run (or *session*).

As a first solution, uploading such statistical snapshots to us for an analysis was left to be performed manually by users. However, to simplify this process for users and improve uploading activity for us, automatic uploading of snapshots was implemented. After such automatic uploading solution was deployed, users of Plumbr were effortlessly contributing to the collection of memory usage and object liveness statistics.

Given such statistical snapshots it was possible to start evaluating the performance of the detection algorithm and further develop it. As mentioned previously, to evaluate the detection performance, a set of snapshots had to be selected and leaking statuses had to be verified by hand. 200 sessions were selected for the study. These sessions were selected in a way, trying to include sessions from different users and applications. Selected sessions were studied manually and leaking statuses were assigned at the allocation level. So, if an application contained at least one leaking allocation, the application itself was considered to be leaking. Next, results were compared with leaking resolutions produced by the statistical approach and detection quality assessment metrics were calculated.

Statistical snapshots contain data on an allocation level and snapshots themselves are grouped in sessions. So, the detection performance can be evaluated also on two levels: session level and allocation level. Session level shows whether the leak was detected in the application and allocation level shows how correctly were leaking allocations identified within the application. Confusion matrices for both application and allocation level are shown in tables 4.3 and 4.4.

	Predicted	
Actual	**Leaking**	**Non-leaking**
Leaking	88	21
Non-leaking	46	45

Table 4.3: Application level confusion matrix

	Predicted	
Actual	**Leaking**	**Non-leaking**
Leaking	1128	1177
Non-leaking	2232	442251

Table 4.4: Allocation level confusion matrix

On the application level precision is 0.8, recall is 0.65, the F_1-score is 0.71. Which can be interpreted as following: statistical approach can correctly identify around 71% of leaking applications. This may seem a good result, however at the allocation level precision is 0.49, recall is 0.33 and the F-score is 0.39. This means that although the leak was correctly identified in 71% of applications, fully correct leaking allocations were identified only for 39% of allocations. From the confusion matrix 4.4 can be seen that on allocation level number of false positives (1177) is comparable to number of true positives (1128) and number of false negatives (2232) is almost twice as much as true positives.

These observations lead to the conclusion that although on application level detection performance is relatively good for a single metric like *genCount*, there is plenty of room for improvement in terms of detection performance on allocation level. While successful cases are analyzed in section 6, following subsections analyze kinds of problems contributing to false results to reveal weaknesses which must be addressed to improve the algorithm.

4.3.2 Lack of sufficient gap in the genCount histogram

The most complex issue with algorithm 4.1 presented in section 4 is the choice of the THRESHOLD constant referenced on line 11, which identifies how different the value of *genCount* for the set of leaking allocations in comparison to normal allocations must be. If the threshold is too low, the algorithm will detect too many false positives. If the threshold is too high, there will be too many false negatives and there is a danger that the application will run out of memory before the algorithm detects anything at all, or that late detection will not leave any time or resources for reference path analysis (an issue in *HtmlUnit* case study, section 6.3).

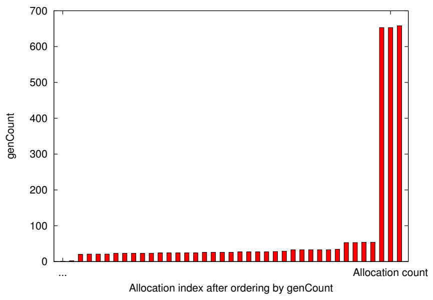


Figure 4.4: Subset of a genCount histogram with the clear gap

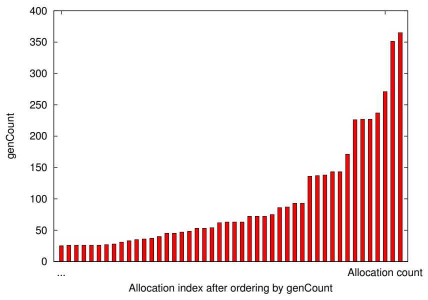


Figure 4.5: Subset of a genCount histogram without the clear gap

Analysis of the results of the algorithm has also shown that there are certain cases without a clear difference between leaking and non-leaking allocations, i.e. in real world applications *genCount* distribution may not have a clear gap exceeding a predefined threshold, thus clearly separating allocations into leaking and not leaking.

Figures 4.4 and 4.5 illustrate the problem. These figures represent parts of the *genCount* histograms from real web applications of different sized code bases. On the X-axis, there are indexes of allocations in a set of allocations ordered by their *genCount* value. The Y-axis shows their actual *genCount* value. For the sake of clarity and readability only allocations with higher *genCount* values are shown; however, most of these allocations usually have a small *genCount*, as shown in section 4.

In the first case (Figure 4.4), the gap can be clearly seen and it is easily detectable using a threshold value. In the second case (figure 4.5) genCount gradually increases over allocations, which means that using the threshold technique is not possible for separating leaking allocations from non-leaking ones. This problem can be particularly evident in large applications with a lot of classes under heavy load, complex user interaction, caches and heavy sessions with varying lifespan. The issue was observed in the *eHealth* web application case study (see section 6.2).

An alternative approach can be taken by losing the whole concept of the gap threshold, presenting the user with a prioritized list of potentially leaking allocation sites, allowing the user to review them, and select the true positive ones. However, shortest path search must be completed and paths must be merged to perform leak analysis. In order to perform path search, leak candidates must be specified. This means that the process of leak detection cannot be automatic and human interaction is required before the final result is produced. This may not be always possible for long running processes, when the leak is detected at a time when there's no human available to perform selection or such selection will take place when the process will be running out of memory and the analysis could not be performed. Another important role of separation of a group of leak candidates in an automated way is to exclude a possibility of human error which could exclude the actual root cause of the leak. So there has to be a way to automatically separate a group of allocations exhibiting similar lifecycles.

4.3.3 Distribution uniformity

In addition to the problem of *genCount* value growth and the existence of a gap between allocations, there is also an issue with distribution uniformity within values contributing to the *genCount* – vector returned by the function $G(c, gc_n)$ defined in equation 4.1 in section 4. To recap, vector $G(c, gc_n)$ shows which generations have live objects from allocation c and it has a length of n (total number of garbage collection generations elapsed).

Consider the example shown in Figure 4.6. This figure shows the distributions of live generations of four allocations within a corresponding vector returned by $G(c, gc_n)$, where $n = 400$, meaning that 400 garbage collection generations have elapsed since the start of the application. A dot on the graph at coordinates (x, y) means that for each allocation x, there is at least one live object created in generation y.

In this example, all four allocations have the same *genCount* of 20, but the values in vector $G(c, gc_n)$ are distributed differently. If we consider that purely looking at the *genCount* parameter (*genCount* of allocation c counts the number of dots on the graph for any given c) is enough, if threshold criteria are met, then all of them should conclude one thing – probability of leaking. However, this figure shows that given the same *genCount* it also should be considered how the values contributing to the *genCount* are distributed throughout time.

In the case of allocation 1, all of its live objects are concentrated within the first 50 generations, which means that these objects were created during system start-up and their number will most likely remain stable, as they probably belong to the application's base classes, meaning that equation 4.3 does not hold even if the *genCount* threshold requirement is met.

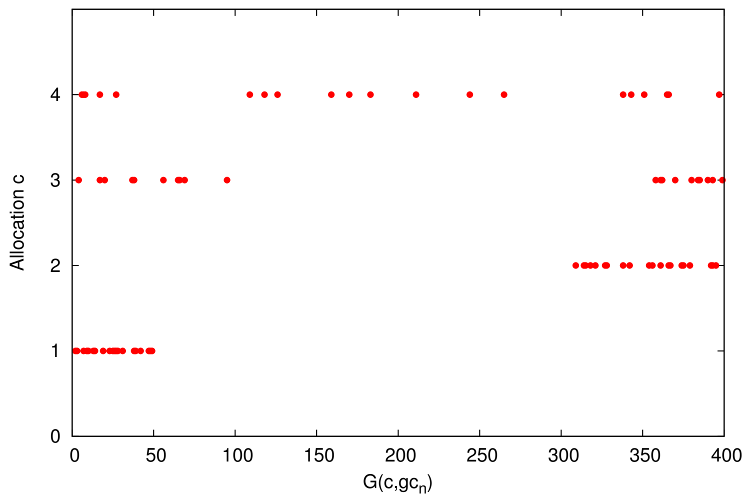


Figure 4.6: Distribution of values within $G(c, gc_n)$ with equal *genCount* Synthetic example of the distribution of values within $G(c, gc_n)$ with equal *genCount*

In the case of allocation 2, all of its live objects are concentrated within the last 100 generations, which means that these can be objects serving incoming requests and which remain alive over several requests. For example, HTTP session objects in a web application will remain reachable between requests and can survive enough garbage collections to reach the *genCount* threshold compared to other application classes. However, after the HTTP session timeout will be reached relevant objects will become eligible for garbage collection. Again, equation 4.3 does not hold and looking purely at the *genCount* threshold is not enough to mark the allocation as leaking.

Allocation 3 represents a hybrid case between cases 1 and 2 where some live instances were created during start-up and some live instances are used for serving incoming requests. Thus, allocation 3 is displaying properties of both allocations 1 and 2 and the *genCount* value of allocation 3 is limited by some constant M as defined in equation 4.4. So, even if this constant

M is larger by a threshold compared to other allocations, it is still not a leak.

Allocation 4, even visually, can be perceived as being a leak as its instances are constantly created since the start of the application and some of its instances have remained alive during the entire run-time. Which, given that a threshold condition is met, can be a good enough reason to mark a class as leaking, as we can assume that this pattern will continue.

To further improve the statistical method a separate indicative property is needed to handle the distribution problem.

4.3.4 Factory methods

Because instances of a class can be created in many different places in the code, the notion of allocation, as described in section 4, is used. Although the allocation site indicates the line of code where the instance of a class is created, sometimes it is not enough to identify the source of the leak, as more context (or deeper stack trace, as the allocation is the top of the call stack) is needed. One common example of where one allocation site is not enough are *factory methods*. Methods implementing the factory design pattern produce objects which are used in many places within the application. Efficient tracking of the full instantiation context is more difficult, as it introduces a greater run time overhead to track full stack traces instead of a single allocation site.

One possible solution to this problem is to apply some sort of context-sensitive *points-to* analysis, e.g. *object-sensitivity* [Mila 05], as a preprocessing step. As a result of this preprocessing, it would be possible to distinguish between allocations based on the callers of the factory method, rather than the factory method itself.

4.3.5 Application uptime and load

When analyzing the distribution of generations of live objects and their uniformity, two other important aspects arose – application uptime and its load. As the goal of the statistical method is to provide high probability memory leak detection as soon as possible, which contradicts with the definition of a memory leak (equation 4.3), which in term expects unlimited growth over time, it means that we should wait longer to see if the leak really is growing without limits. Given a uniformly distributed vector of generations, such as allocation 4 in Figure 4.6, and knowing that the application was running for a day, we may say with high probability that instances of allocation 4 are quite probably leaking, as the heap contains reachable objects which were created during a day, which is probably longer than any HTTP session timeout. Given the same distribution and knowing that the application was running for less than a minute, we should probably wait a little longer to see if objects will be garbage collected. Unfortunately, the estimation of *long* or *short* uptime duration is very relative and depends on application usage, meaning that we cannot provide a rule of thumb to detect whether an application was working *long enough*.

The load of the application is also a relevant topic of the uptime issue. It emerged when comparing real world applications with synthetic tests, which are often created to test leak detection algorithms. While server-side applications tend to work continuously for a long period of time, meaning the leak has time to grow and we have time to detect it, then synthetic tests work as a fast burst of only leaking memory and will exhaust the heap too quickly for the statistical method to detect anything.

4.3.6 Lazy application caches

There can be two types of caches in the applications – eager caches and lazy caches. An eager cache will be loaded with data once and it will be alive for a certain period of time. From *genCount*'s point of view, an eager cache will have lot of objects within one or several close generations, such as allocations 1 and 2 in Figure 4.6. So, eager caches are not erroneously identified as leaks.

Lazy caches, on the contrary, are filled with data when needed, so in some cases such caches may look like leaks. Well-behaved caches must somehow be restricted from occupying the entire heap. If this restriction is not in place, then the cache can cause $OutOfMemoryError$ and a subsequent crash of the JVM, which makes the cache ill-behaved. However, some lazy caches may be smart enough to monitor memory usage and unload unused objects only when heap usage exceeds a certain threshold. Some special handling must be added to the automated statistical approach to distinguish such caches from memory leaks, which is a research topic all its own.

4.4 Analysis of the runtime performance

To measure the overhead of Plumbr, DaCapo Benchmark suite version 9.12 was used [Blac 06]. Benchmarks were executed on Oracle HotSpot JVM 1.7.0_09 running on a 2.70 GHz Intel^{TM}Core^{TM}i7 CPU with 8 GB of RAM, SSD drive and MacOS X 10.8 operating system. DaCapo benchmark was set up to perform 30 iterations of all benchmarks.

To record memory usage during the execution, the same technique was used as during the case studies. The execution time of benchmarks was also recorded. Memory usage graphs cover all 30 sequential executions of benchmarks (to get a bigger picture which would account for warm-up), whereas execution time was analyzed separately from all runs.

Measurements of execution times are summarized in Table 4.5, showing the minimum, maximum, mean and standard deviation of execution times with and without Plumbr. The total mean overhead across all benchmarks' mean run times is 41%. The negative overhead percentage for maximum *avrora* benchmark execution times can be explained by the first iterations when the warm-up was not yet completed and this is the period which produces the longest runs in the whole suite. Large deviations across runs are caused by the overall short execution times; even a slight change in conditions results in a relatively large deviation. This may be attributed to be a problem of the DaCapo benchmark suite, which outputs its results as execution times instead of a compound score, like the SpecJVM benchmark suite. Considering complex environment setup with Plumbr being attached, and that numerous components need to be initialized first, such short benchmarks may not be the best way to evaluate the performance of such a complex system.

To illustrate memory overhead, we show the top 5 longest-running benchmarks: *eclipse* (Figure 4.7), *tradesoap* (Figure 4.11), *tomcat* (Figure 4.9), *tradebeans* (Figure 4.10) and *sunflow* (Figure 4.8).

From the results, we can calculate the mean execution time overhead over all benchmarks: 41%. Although, actual values for particular tests fluctuate from 2% on *avrora* and *luindex* up to 127% on *eclipse.* Inspection of GC logs showed that collectors' throughput in case of eclipse benchmark is almost twice as big with Plumbr as without and accumulated pause times is nearly ten times bigger. This indicates that eclipse allocates large

	Min			Mean			Max			Std dev	
Benchmark	**Cl.**	**Pl.**	**OH**	**Cl.**	**Pl.**	**OH**	**Cl.**	**Pl.**	**OH**	**Cl.**	**Pl.**
avrora	3 828	3 939	3%	3 984	4 058	2%	5 799	5 532	-5%	389	296
batik	1 099	1 193	9%	1 350	1 526	13%	6 253	8 275	32%	939	1 286
eclipse	19 838	46 490	134%	21 680	49 117	127%	42 905	77 455	81%	4 179	5 599
fop	225	387	72%	416	683	64%	2 938	4 191	43%	512	701
h2	4 242	5 136	21%	4 691	5 772	23%	6 929	7 429	7%	459	446
jython	1 662	2 640	59%	2 211	3 356	52%	11 424	15 549	36%	1 809	2 370
luindex	524	527	1%	650	664	2%	1 422	2 396	68%	197	352
lusearch	1 257	2 479	97%	1 482	2 748	85%	5 632	6 156	9%	789	651
pmd	1 811	2 283	26%	2 068	2 767	34%	6 172	9 580	55%	823	1 488
sunflow	3 560	6 231	75%	3 798	6 418	69%	4 481	8 445	88%	139	398
tomcat	2 156	2 491	16%	2 439	2 971	22%	5 151	7 158	39%	581	854
tradebeans	5 013	7 081	41%	5 173	7 283	41%	7 084	10 724	51%	364	657
tradesoap	9 550	11 887	24%	15 991	16 583	4%	21 096	39 776	89%	2 152	4 859
xalan	1 380	1 868	35%	1 652	2 307	40%	6 165	10 508	70%	885	1 572

Table 4.5: DaCapo benchmark results in milliseconds without (Cl) and with Plumbr (Pl.) attached

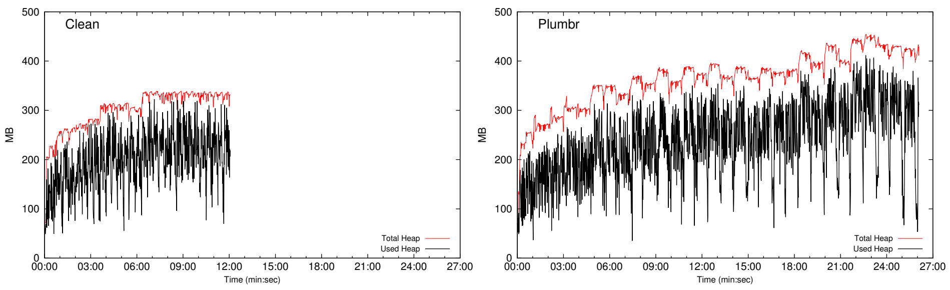


Figure 4.7: Memory usage for eclipse benchmark

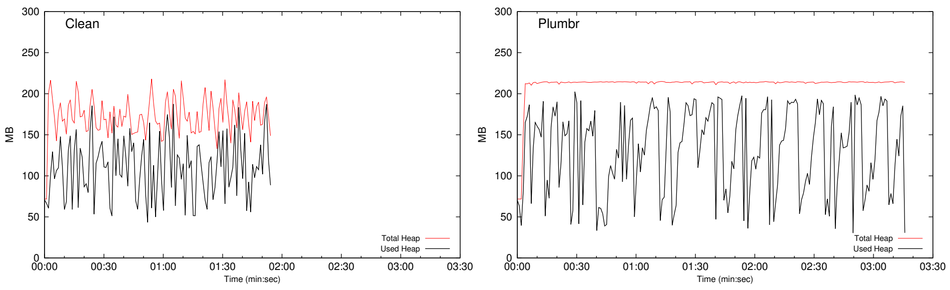


Figure 4.8: Memory usage for sunflow benchmark

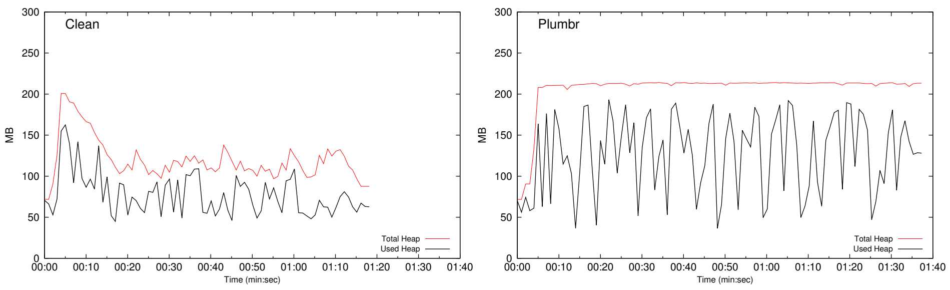


Figure 4.9: Memory usage for tomcat benchmark

amounts of objects, which lead to high tracking overhead. From this, it can be concluded that actual performance overhead of Plumbr is very much dependent on the application and which subsystem (CPU-intensive, high object creation ratio, I/O, etc.) the application is stressing.

The same set of benchmarks was also executed on different machines – on the Amazon EC2 cloud instance and on a PC with a traditional hard

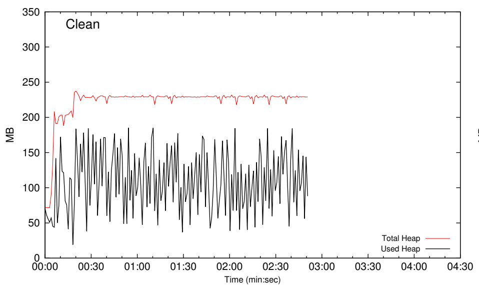


Figure 4.10: Memory usage for tradebeans benchmark

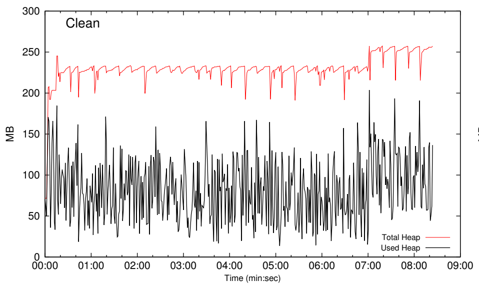


Figure 4.11: Memory usage for tradesoap benchmark

drive. It was observed that because of slower disk access performance, execution time overhead was lower than the average 41% shown previously – as disk I/O operations take more time from the total execution time, the CPU part decreases in total time, so Plumbr overhead is less perceivable.

4.5 Summary

This chapter described the statistical approach for the memory leak detection in Java applications, the most important contribution of the thesis. The notion of *genCount* along with formal definition was presented. The proof of concept algorithm, utilizing introduced concept, along with its implementation details were described. An implementation of the described algorithm called Plumbr was presented. Plumbr was used by actual end users to collect the data for the validation of the concept. Based on the collected statistics it was concluded that the hypothesis about using $genCount$ as a major indicator for memory leak detection was valid. The analysis of the data along with the evaluation of the detection performance and the runtime performance were also presented in the current chapter.

Chapter 5

Improving Statistical Approach using Machine Learning

As analysis in section 4.3.1 revealed, using single metric like *genCount* and a simple algorithm utilizing a constant threshold value, detects presence of the leak in the application with F_1 score of 0.71. However, fully correct leaking allocations are identified with F_1 score of only 0.39. Also, it was shown that additional attributes could improve statistical memory leak detection. However, when using one or two attributes it is easy to analyze them by hand, but when using multiple attributes and considering their combinations, manual analysis becomes unfeasible. Machine learning was chosen to help introduce additional parameters for the leak detection.

As definition in section 2.2 suggests, to apply machine learning one has to define a set of data to learn from (E), a measure of performance to improve (P) and identify tasks (T) which the learning has to achieve. In the case of memory leak detection the task to perform T may be formulated as: identify which allocations in the application are leaking. The source of experience E is composed of statistical snapshots as described in section

4.3.1. The measure of performance to improve P will be the F_1 score, as defined in section 2.2.

5.1 Setting the baseline

To use a wider range of attributes supervised machine learning was used. To apply supervised learning, a set of examples of java allocations and their leakage statuses is required. Possible leakage statuses of a class are discrete: *leaking* and *not leaking*. If the outcome of the supervised learning is discrete then the function that associates the attributes' values and the leakage status is called a classifier and in that case supervised learning algorithms are also called *classification algorithms*.

Before setting off to feature design and application of any concrete learning algorithms a baseline and a data set must be defined in order to have clear understanding of what is the starting point and what is desired to be achieved.

As described in section 4.3.1, source data for the leak detection per application is a set of statistical snapshots, where each snapshot in the set is acquired after a full garbage collection. One snapshot contains a set of all allocations $c \in C$ in the application and a vector $G(c, gc_j)$, as defined in section 4.2.1. By the beginning of the learning process there was a set of statistical snapshots from 10 894 application runs (sessions), collected from 974 different users, who were using Plumbr. Out of these, 200 sessions were hand-picked, manually inspected, and actual leaking statuses were assigned on an allocation level. Sessions were chosen in a way that exactly half of those 200 sessions contained allocations with the leakage status *leaking*. Snapshots from these 200 sessions will be referred as *full data set*.

Applications were separated into two sets for training and for testing of classification algorithms in a proportion of 2/3 for training and 1/3 for testing. The training set contained 134 applications in total, 67 applications out of those contained leaking allocations and 67 applications consisted only of not leaking allocations. The rest 66 applications were selected into testing data set. Training set consisted of 287 776 allocations and 0.5% of them were leaking. Testing set consisted of approximately 159 037 allocations and 0.7% of them were leaking.

To fix the baseline before proceeding to testing classifiers, the statistical approach was applied to the testing data set, which will be used for validation of the classifiers. As testing set constitutes 1/3 of the full data set and fraction of leaking allocations is so small, then using the numbers calculated in section 4.3.1 from the full data set is not feasible as they differ from the results obtained from the testing data set.

	Predicted	
Actual	**Leaking**	**Non-leaking**
Leaking	511	367
Non-leaking	577	157 582

Table 5.1: Baseline confusion matrix

Confusion matrix for the baseline is shown in Table 5.1. Baseline precision is 0.582, recall is 0.469 and the F_1-score is 0.51.

5.2 Design of attributes

Attribute identification is the first task in the implementation of machine learning. Attributes of an allocation, which can be used for learning, can be any features which can be used to characterize the allocation, e.g., name of the class, number of live objects, *genCount* value, number of all generations and ratios of combinations of other attributes. Further on, a specification of an attribute and its value is called a *feature*.

To identify learning attributes statistical snapshots accompanied with leakage statuses on an allocation level were used. To determine learning attributes, 44 applications from the full data set were chosen so that they would be as different as possible, based on their allocation and object age distribution. Difference between applications was derived from visual evaluation of plotted distribution of live generations of allocations over time. From previous analysis it was known, that 11 of these 44 selected applications were given incorrect diagnosis by the statistical approach.

Two human experts were employed, who were able to verify whether the leakage status of the class is leaking or not leaking and correlate their decision with the decision of the default approach. Human troubleshooting experts were using professionally similar memory leak detection approach based on the *genCount* value, however, they also used 'gut feeling' which was based on yet unidentified attributes. The main goal at this stage was to determine which additional attributes they were using without formal specification. To give their resolutions, experts were allowed to use only the data available in the collected snapshots (also in visualized form). During reevaluation of these 44 applications, the decision process of experts was tracked by letting them document their decision process to identify such 'gut feeling' attributes.

Evaluation process was done in 2 iterations. During the first iteration experts worked independently. After the first iteration of determination, approximately 10% of expert resolutions were conflicting. In the second iteration experts worked together with the goal to resolve conflicts. After the second iteration the final common determination of each allocation was achieved. The result of the final determination concluded that 20 applications from 44 contained allocations with the leakage status *leaking*. The total number of allocations in all applications was approximately 130 000 and 0.3% of them were leaking based on expert resolution.

When making their decisions and resolving conflicts, experts were creating a log, which was further analyzed and additional features were determined. Some of these features were usable for human experts, but currently not measurable. One example of such feature is a warm up period of an application. During the warm up period the first initialization process is taking place, a lot of classes are loaded and instantiated and no class can be considered leaking until warm up is finished. As it is not yet possible to detect automatically when warm up period of the application is over, such attribute is omitted. Analysis of the decision logs discovered six attributes to consider.

1. Gap between two neighbour $genCount$ values of allocations after sorting the allocations by their $difGen$ value. The same attribute used in the baseline classifier, as described in Listing 4.1.

2. $genCount$ – major indicator, as described in section 4.

3. Overall uptime of an application – as outlined in section 4.3.5, ignoring the uptime of an application was causing false positive results for the baseline detection approach. For the attribute value uptime is measured in minutes.

4. Uniformity of $genCount$ distribution – analysis in section 4.3.3 has shown that distribution uniformity is affecting leak detection quality. To mitigate the problem, the uniformity of $genCount$ distribution is defined as a standard deviation of distances between neighbor non-zero elements in $G(c, gc_n)$ vector. The intuition behind the definition tells that if leaking objects are created regularly, then the distances between generations of live objects are more equal, than distances between alive generations of allocations which are created in bursts over the lifetime of the application. For illustration of the concept see Figure 4.6 in section 4.3.3. So, the standard deviation of the distances between non-zero elements in $G(c, gc_j)$ vector should indicate the uniformity in our case – the smaller the deviation, the more equal are distances between elements, the more uniformly are distributed

the live generations in time. To normalize the feature across different applications the value of uniformity is divided by the length of the $G(c, gc_n)$ vector.

5. Ratio between live objects of an allocation and its $genCount$ indicates how many live objects are there in application per one live generation on average. This ratio can give insight to how many instances of an allocation may leak at a time. If the ratio is small then it means that there is a small amount of instances per live generation, which in combination with high $genCount$, which is uniformly distributed, may indicate that there are some instances of an allocation which are created regularly over time, small number of them is surviving, which looks like a memory leak.

6. Ratio between number of an allocations with the same $genCount$ gap and number of all classes (*leakingAllocationRatio*) can give insight to how large part of all allocations within the application is exhibiting similar liveness pattern.

Ratios, instead of absolute values, are used to normalize input data and improve feature comparison for classes from different applications.

The log of expert's decision process was using discrete qualifiers as *low* and *high* or *small* and *big* for all of the attributes. After the features were extracted from the data set two experiments were conducted. One used raw features for the learning. For the second experiment continuous values of all attributes were manually discretized using thresholds, which were based on experts 'gut feeling', into three groups: *small*, *medium* and *large.*

Such manual discretization resulted in a training set which contained many allocations with the same feature values. After removing allocations with duplicate features there were only 235 allocations with unique combination of features left, which makes only 0.8% of all allocations in the selected 134 applications. Because such drastic difference could skew the result of training, chosen classification algorithms were trained and tested twice with discretized training data set. First time algorithms were trained with all allocations from discretized training data set. For the second training duplicates were removed from the training data set. Testing data set was not filtered in both cases to make the results comparable.

To sum up, six attributes were defined and training data set was used in three configurations: continuous values, discrete values and discrete values with duplicates filtered out.

5.3 Experimental results and analysis

Three classification algorithms were used for experiments: C4.5 [Quin 93], PART [Fran 98], and Random Forest [Brei 01].

These algorithms were chosen because they are known to perform well over a wide range of learning tasks, thus being general enough. On the other hand, C4.5 produces a decision tree, PART produces the a rule list, and Random Forest produces an ensemble of trees, thus using different approaches. For all three algorithms implementations from Weka toolkit [Hall 09] were used (C4.5 is called J48 in Weka).

On the continuous training set C4.5 used 5 attributes (all, except *leaking AllocationRatio*) and generated a tree of size 121 with the number of leaves of 61. On the discrete training set C4.5 used 5 attributes (all, except *leakingAllocationRatio*) and generated a tree of size 35 with 18 leaves. On the discrete training set without duplicates C4.5 used only two attributes (*genCount* and *genCount* uniformity) and generated the tree of size 5 with 3 leaves. The confusion table with calculated metrics can be see in the summary table 5.2.

On the continuous training set PART used 5 attributes and produced 36 rules. On the discrete training set PART used 5 attributes and produced 16 rules. On the discrete training set without duplicates PART used 3 attributes, with 5 rules. The confusion table with calculated metrics can be see in the summary table 5.2.

Random Forest was tried with different number of trees (10, 30, 50) and different number of features (from 1 to 6). Results shown in the summary table 5.2 are obtained with the combination of 30 trees and 6 features, which gave the best F_1 score.

In all cases classification algorithms were evaluated using allocations from the testing group. Baseline results, evaluated with the testing group, are also shown in the summary table 5.2.

Algorithm	**Group**	**TP**	**FP**	**FN**	**TN**	**Prec**	**Rec**	F_1	**AUC**
Baseline		511	367	577	157582	0.582	0.469	0.519	0.789
C4.5	cont.	540	338	113	158046	0.827	0.615	0.705	0.996
C4.5	disc.	745	133	252	157907	0.747	0.849	0.795	0.996
C4.5	no dups.	788	90	454	157705	0.634	0.897	0.743	0.962
PART	cont.	529	349	793	157366	0.400	0.603	0.481	0.648
PART	disc.	745	133	247	157912	0.751	0.849	0.797	0.996
PART	no dups.	509	369	81	158078	0.863	0.580	0.693	0.958
RandomForest	cont.	762	116	354	157805	0.683	0.868	0.764	0.948
RandomForest	disc.	745	133	253	157906	0.746	0.849	0.794	0.958
RandomForest	no dups.	620	258	3811	154348	0.140	0.706	0.234	0.957

Table 5.2: Summary of machine learning experiments (group *cont* means training data set with continuous values, *disc* means manually discretized training data set, *no dups.* means manually discretized training data set with allocations containing only unique features)

As training was performed on an unbalanced dataset (0.5% positives and 99.5% negatives), there was the risk that classifier can be tempted to score all samples as negative as this will yield a high score. However, from confusion matrix it can be seen that such risk has not realized. Moreover, trained algorithms show smaller number of false negatives than the baseline.

Notably, training on manually discretized data performed best considering F_1 score. And almost equally so for all three classifiers. With such training data set C4.5 and RandomForest performed identically – difference is in 1 false negative, which is negligible. PART managed to produce 5 false negatives less and thus becomes a winner with a F_1 score of 0.797.

It can be seen that using only *genCount* attribute alone, as in baseline statistical approach, is enough to predict leaks with a score of 0.51, which means that this is the most dominant attribute. Adding 5 more parameters resulted in 27% better leak prediction performance. However, from the confusion table it can be seen that the overall improvement came from the decreasing number of false positives, produced by simple threshold method.

5.4 Summary

In this section it was described how the machine learning was used to improve the detection quality of the statistical approach for memory leak detection in Java applications. In order to achieve improvements, five additional learning features were identified based on the analysis of the baseline method. Three machine learning algorithms, C4.5, PART and Random Forest were used on the dataset obtained from real applications which were using Plumbr – the implementation of the statistical approach for memory leak detection.

Before applying machine learning the baseline performance metrics, like F_1-score and ROC AUC were measured. Same metrics for the machine learning approach showed 27% increase in F_1-score, at the expense of reduced number of false positives and false negatives.

Chapter 6

Case Studies

As Plumbr uses run time information to generate the leak report, it is crucial that the leak is detected, analyzed and the report generated before the application runs out of memory. How long it will take detect the leak (in case there is a leak in the application) depends on the threshold in algorithm 4.1. We conducted several case studies to investigate the behavior of the systems containing a memory leak.

For these experiments, we had to find known memory leak bugs to see if our tool could find the right reason for the leak. For that, we searched the issue tracker of the Apache Software Foundation [Apac 11a] for memory leak bugs in Java-based projects. Among these, we searched for ones that had the test case attached as a unit test, to be able to reproduce bugs. In addition to these open source projects, we also looked for known bugs in business web applications with closed source code.

Current work does not include any false positive case studies, instead, sections 4.3.2, 4.3.3, 4.3.4, 4.3.5 and 4.3.6 analyze the reasons for possible false positive results. The reason for such handling is that in the case of actual memory leaks, the amount of free heap is degrading, thus affecting the performance of the application. Therefore, if performance is degrading and at some point the JVM crashes it would be interesting to see whether

Plumbr would be able to generate the report before the crash, and what kind of overhead would be given by Plumbr. In the case of false positives, the only impact is that wrong leaks are reported, but as the application does not have a leak, there is no risk of failure or memory pressure even if the report is generated and it does contain wrongly identified leaks. In terms of the execution overhead in non-leaking applications, refer to section 4.4 for the performance analysis.

Every case study followed the same process:

1. Collect a baseline against which to evaluate our agents' efficiency and overhead. For this purpose, every application under the study was run without Plumbr agents. The following baseline numbers were marked:

   (a) Time to crash with `OutOfMemoryError`

   (b) Memory usage (both heap and native) during the course of the run

2. Run the application until the crash, with troubleshooting agents attached. In addition to the baseline metrics, we were interested in the following information:

   (a) Whether the agent will find the leak

   (b) Whether there will be any false positive alerts

   (c) How fast will the agent be able to spot the leak and produce a report

   (d) Whether the agent will influence the time to the crash (compared to the baseline metric)

   (e) How large a memory overhead is caused by the agent (compared to the baseline metric)

Experiments were conducted in Amazon EC2 cloud [Amaz 14], using Standard Large instance running 64bit Ubuntu Linux and Oracle Java HotSpot 1.6.0_24 64-Bit Server VM (build 19.1-b02, mixed mode). At the time of conducting experiments, the standard large instance had the following characteristics: 7.5 GB memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB instance storage, 64-bit platform.

To log memory consumption a small Java agent was used which measured memory usage every second using the standard $java.lang.Runtime.$ $totalMemory()$ and $java.lang.Runtime.freeMemory()$ methods and logged the values to the file.

In total, 6 different experiments were conducted with bug reports from 4 different projects from Apache Software Foundation (Velocity, Xalan, Derby and ActiveMQ), among which ActiveMQ [Acti 11] was chosen for the presentation in this thesis. ActiveMQ was chosen as it is a popular message broker which can be used as a standalone application in a cloud environment, in contrast to other projects which are either utility frameworks (Velocity and Xalan) or are not widely used (Derby). In addition to the above, case studies with HtmlUnit from SourceForge [Memo 11] and the closed source eHealth web-application are also presented in the following subsections.

ActiveMQ and the eHealth application case studies were also described in [vSor 11b], the HtmlUnit case study was added specifically for this publication.

6.1 Case study: Apache ActiveMQ

Apache ActiveMQ [Apac 14] is an open source message broker which fully implements the Java Message Service 1.1 (JMS) specification. It provides many features such as clustering, multiple message stores, and the ability to use any database as a JMS persistence provider, besides the VM, cache, and journal persistence required by large enterprise installations.

The subject of this case study was a bug reported to Apache ActiveMQ project's issue tracker [Acti 11]. It was reported before we conducted case studies and we reused the test case attached to the ticket. A modified version of the description from the bug report follows.

Two parties, a server (called a broker) and a client participate in communications using the ActiveMQ infrastructure. The broker's job is to listen for incoming connections, accept them and respond to events occurring during this connection lifetime, as per protocol. The client opens a connection to the broker, and then, in violation of ActiveMQ protocol, just drops it. This can take place either due to buggy client code or due to network problems. The result of such a dropped connection is that the instance of object, representing the client who has initiated this connection, remains in the broker's memory. If many clients repeatedly create new connections to the same broker and at some point in the future just drop this connection without closing it, as required by ActiveMQ protocol, it can result in many objects accumulating on the broker's side. As the broker cannot distinguish between a dropped client and the client who merely is inactive for some time, these objects are held by the broker until the broker's memory is depleted, which results in OutOfMemoryError and the broker's crash.

In order to reproduce the reported bug in our environment, a sample client for ActiveMQ was written. This client spawned a number of threads, each opening a new connection to the broker and then abruptly closing this connection. This process was run repeatedly, simulating network problems over time. ActiveMQ broker was run with default settings, as provided in default start-up scripts with the distribution. As a result, in both cases (with and without an agent), the broker was run with Java heap size fixed to 64MB. It must be noted that our sample client was specifically targeted for this bug and was therefore only exploiting the leak. To bring this scenario closer to a real-world situation a pause of 2 seconds was introduced between requests.

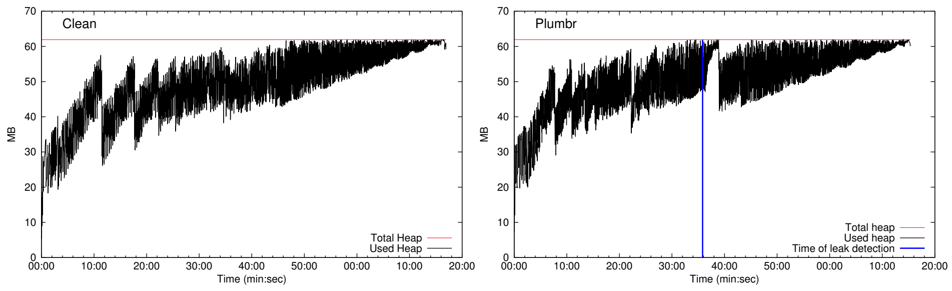


Figure 6.1: Case study: Apache ActiveMQ memory usage
Apache ActiveMQ memory usage without (Clean) and with Plumbr

Following the description in section 6, the baseline was recorded in the first step:

(a) The broker runs for 1 hour and 17 minutes before the crash

(b) Memory usage is depicted in Figure 6.1 (Clean).

After that, Plumbr was attached to the broker and the same sample client was run. The results were as follows:

(a) The agent was able to successfully find the correct source of leaking objects.

(b) No false positives were reported.

(c) The leak was detected 36 minutes after the start of the application.

(d) The broker ran for 1 hour and 15 minutes before crashing.

(e) Memory usage is depicted in Figure 6.1 (Plumbr).

A comparison of memory usage graphs (Figure 6.1) shows that troubleshooting agents created no significant memory overhead. However, a slight increase in the frequency of full garbage collector runs can be seen (large drops in heap usage indicate full garbage collection).

In conclusion, it should be mentioned, that such a leak can easily occur in distributed environments, especially in such dynamic ones as a cloud. In a distributed environment, network outages, lags, congestion as well as client machine crashes are all very likely scenarios. In addition, such a memory leak can be easily exploited against a service provider who uses ActiveMQ to create Denial of Service (DoS) type attack (to cause service outages simply by opening connections and then dropping them). In the context of an elastic cloud computing environment with dynamic resource allocation, the provider could be attacked, causing them to allocate new instances to serve more and more non-existing clients, in turn causing financial damage by forcing them to pay for allocated resources.

6.2 Case study: eHealth web application

This case study focuses on a large enterprise application ($\approx$ 1 million lines of code) with a web front-end – the eHealth web portal, including patient record handling and many other healthcare related functions. The application is actively developed and supported by a dedicated team of developers. At one point in time, the team started receiving reports that the application crashes in the production environment every few days with `java.lang.OutOfMemoryError`. As the team was unable to fix the leak within a reasonable period of time, it was estimated that allocating increased development resources to the problem is more expensive than simply having the client's operations team restart the application servers every other night in order to prevent a crash. It must be noted that such a solution is not uncommon in other similar projects, as regularly restarting

Java servers is often a lot cheaper and simpler, than finding the leak with existing tools and expertise.

As the case study was conducted when Plumbr was still in a very early research phase, we were unable to attach the agents to the production servers due to the client's concerns about the agents' stability. However, we were allowed to use a test environment to verify that Plumbr would be able to find the leak. As the exact sequence of actions leading to the leak was unknown, the leak first had to be found manually. Manual heap dump analysis was used to do that. After the leak was identified manually, a test case was generated to simulate the needed activities in the test environment using Apache JMeter [Apac 11b] script.

The next step in this case study was to attach our agent to the application server in the test environment, run the recorded JMeter stress-test, and determine whether our agent is able to find the problem.

In both cases, with and without the agent, the application was run with a Java heap size of 512MB.

When memory usage was first recorded, it was noted that without Plumbr, memory consumption over time grows even faster than with Plumbr (Figure 6.2). Apparently, memory consumption is closely related to the amount of requests performed by the application. So to get the full picture, the number of performed requests is also added to Figure 6.2 (to fit both memory consumption and the number of requests on one scale, the number of requests must be multiplied by 100).

Following the description in section 6, the baseline was recorded as a first step:

(a) The application ran for approximately 2 hours and 30 minutes before crashing

(b) Memory usage as well as the number of requests sent is depicted in Figure 6.2 (Clean)

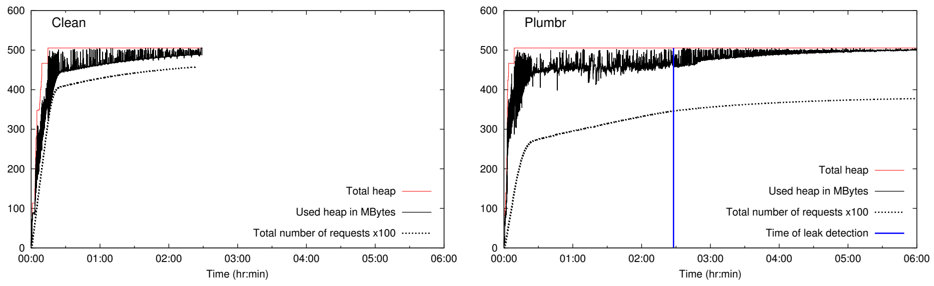


Figure 6.2: Case study: eHealth web application memory usage eHealth web application memory usage without (Clean) and with Plumbr

After that, the agent was attached to the application server and the same stress-test was run. Results were as follows:

(a) The agent was able to successfully find two sources of leaking objects (the application actually had two leaks, one of which was not discovered by manual analysis)

(b) No false positives were reported

(c) The leak was spotted after 2h 30min after the application's start

(d) The application ran for approximately 6h before crashing

(e) Memory usage and the number of requests are depicted in Figure 6.2 (Plumbr).

We can draw very interesting results from these figures.

First of all, let us look at the number of requests the application was able to serve and how this number changed over time. With both of these, figures we see a rapid degradation of the application's performance approximately 30 minutes into the stress test. This degradation can be easily explained by taking the amount of the free memory available to the application into account. As it decreases, the garbage collector starts to run more and more often and takes more and more time. Which, in turn, leaves less opportunity for the application to serve incoming requests. Hence, we see performance degradation and, as a result, a slower increase in memory used. So, in the case of a memory leak situation, not only does the overall performance of the application decrease, but the speed of the leak decreases significantly over time as well. This also explains the fact that the application crashes much later compared to the start of the slowdown.

Second, comparing the first 30 minutes of running the stress test with and without the agent, we can estimate the overhead of running an application with our agent to be approximately 35%, which is almost in sync with the 41% average overhead observed in DaCapo benchmarks (see section 4.4).

Third, we have one counter-intuitive result of the application running for 6 hours with the agents, vs. 2 hours and 30 minutes without them. This can be attributed to the fact that due to agents' performance overhead, fewer requests were served during this time, and, as a result, memory leaks were consuming memory at a much slower pace, so the application managed to stay alive for a much longer time, although most of the time was occupied by garbage collection, not real work.

From Figure 6.2 we also can see a very interesting thing about memory leaks in Java applications – when the amount of free heap reaches its limit, application will continue to run and the garbage collector will attempt to free memory, but the performance of the application is dramatically decreased. For example, in this case, clients would be experiencing long response times and timeout errors for 2 hours before the application actually crashes.

6.3 Case study: HtmlUnit

HtmlUnit is an open source project hosted on SourceForge [Html 11]. It is a "GUI-Less browser for Java programs". Most often it is used as an embedded browser in unit testing frameworks, to automatically navigate on web pages without user intervention. As the cloud is increasingly used for automated and distributed load and functional testing of web-applications, HtmlUnit is used to perform navigation and browsing for such applications.

The memory leak we studied is discussed on the HtmlUnit user's mailing list [Memo 11]. This leak was found, solved and reported by us while we were testing Plumbr.

The essence of the leak is as follows. It happens only when an HTML page has a special condition – it must have a JavaScript job, which is executing periodically without reloading the whole page. When the HtmlUnit opens such an HTML file, the embedded browser creates Java objects representing such periodic jobs each time this job must be executed. Then this Java object, representing this JavaScript job in the HtmlUnit run time environment, is stored in some internal storage. Due to a bug in the JavaScript engine in HtmlUnit this object is not released from storage even after the corresponding JavaScript job has completed its execution. As a result, if the HTML page remains open long enough, an `OutOfMemory` condition can occur and HtmlUnit crashes.

In order to reproduce this problem, we created a sample HTML page with a JavaScript code which updates the page with some random text. The Java representation of this page is held in HtmlUnit's `WebPage` object, which in turn is held in HtmlUnit's `WebClient` object, which is the originator of HTTP call for the page. The leak itself happens in `JavaScriptJobManager-Impl`, which is held inside `WebClient`. Figure 4.2 in section 4.2.2 displays the final report for the HtmlUnit case study.

Following the description in section 6, the baseline was recorded as a first step:

(a) It took 28 minutes for the crash to happen;

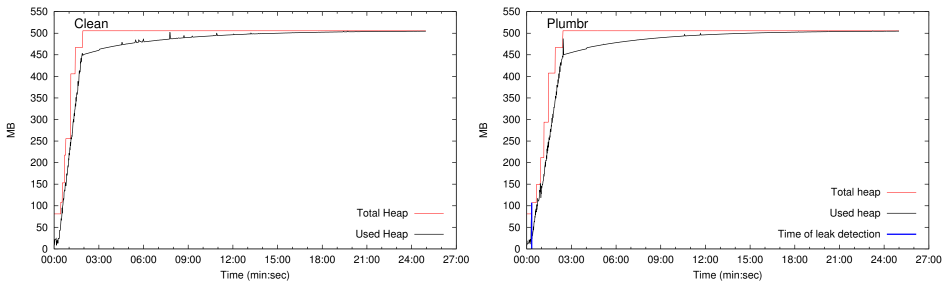


Figure 6.3: Case study: HtmlUnit memory usage
HtmlUnit memory usage without (Clean) and with Plumbr

(b) Memory usage without the agents is depicted in Figure 6.3 (Clean) (note that after 25 minutes of run time memory usage was constantly near total available heap and we cut the graph on minute 25 for the sake of readability of the figure).

After that, troubleshooting agents were attached to the HtmlUnit test case and the same HTML page was opened in a loop. The results were as follows:

(a) Agent was able to successfully find the correct source of leaking objects

(b) No false positives were reported

(c) The leak was spotted 20 seconds after the start of the application

(d) The page was held active for 26 minutes before the crash

(e) Memory usage is depicted in Figure 6.3 (Plumbr).

In this case, thanks to rapid leak detection, the agent was able to perform the required analysis very quickly and disable its monitoring part early in the test run. Thus, no overhead can be discovered from the graph. The reason for such rapid detection is that this test case is synthetic in the sense that it is exploiting one particular leak in a highly focused way, and it is not doing anything else.

However, during some runs of this test case with agents, albeit with different memory settings, the expected memory leak was spotted and reported; however, the full analysis, which is needed to pinpoint the path from garbage collection roots to leaking objects, was unable to finish due to the limited memory settings. This is an example of the situation described in section 4.2.2 when leaks, having their roots in local variables, can require considerably more time to be analyzed. In such situations the ability to perform the full analysis offline, using metadata collected and saved by agents during the first collection phase, right after the memory leak was detected, becomes valuable.

6.4 Comparison with the LeakChaser

To compare leak detection quality with competitive solutions a special case study was conducted. First, the competitors had to be chosen. As the statistical approach for memory leak detection is a general purpose online approach, according to the classification in chapter 3, a comparison to another general purpose online or hybrid approach would make sense. When choosing a hybrid approach, it should use also online analysis. As all online general purpose leak detection approaches were at least five years old at the moment of the comparison and Java virtual machines have made a lot of progress, the *LeakChaser* [Xu 11] was selected for comparison. As the publication contains comparison to the *Sleigh* [Bond 06], transitive conclusions may be drawn. Moreover, Sleigh and its bit-encoding (Bell) is used in other leak detection tools as well, so comparing to the LeakChaser is a reasonable choice. While it considers leaks based on the allocation sites, it makes the LeakChecker more similar to the statistical approach.

When comparing memory leak detection approaches by their *intrusiveness*, approaches requiring changes in the virtual machine were classified as highly intrusive. Such classification becomes apparent when one shall try to apply such an approach outside of laboratory environment. LeakChaser was implemented in a Jikes RVM 3.1.0 and is available as a patch to the Jikes RVM source code. As the patch can be applied only to a correct version of the source, such correct version has to be obtained and built or it has to be re-implemented and adapted to the more modern version of the virtual machine. Both approaches are very time and labour consuming. Once the seemingly easier approach of building the old version of the JVM is chosen, one must consider that modern environments have also developed and some tool chains needed to build the JVM are also changed. But once the changes are applied and all issues resolved (which may take several days), the implementation of the approach can be used. It must be noted, though, that described challenges are engineering problems, and not the problems with the scientific part of the approach.

Listing 6.1: Leaking class

```java
public static class Garbage {
  int[] x = new int[1000];
  String y;
  Date[] d = new Date[2];

  public Garbage(int i) {
    x[0] = i;
    y = new String(String.valueOf(x[0]));
  }
}
```

As reported by the manuals of Jikes RVM, it does not fully support default class libraries. So, for example a *tradesoap* benchmark from the suite of DaCapo benchmarks does not work with Jikes RVM 3.0.1. For this reason, before trying real leaks, one simple synthetic test was tried first.

The test leaks instances of the class `Garbage`, which in turn contains three fields, see listing 6.1.

Passing in an argument which is actually not used and evaluating internal values are needed to *fool* the optimizing JIT (Just In Time) compiler not to apply optimizations and lose leaking allocations altogether.

Listing 6.2: Leaking code

```
public class GarbageProducer {
  static List<Garbage> randomLeakingGarbage =
            new LinkedList<Garbage>();
  static List<Garbage> anotherRandomLeakingGarbage =
            new ArrayList<Garbage>();
  void doSomeWork(Garbage o) {
    randomLeakingGarbage.add(o);
  }
  void doMoreWork(Garbage o) {
    anotherRandomLeakingGarbage.add(o);
  }
  public GarbageProducer(int iterations, Random r) {
    Transaction workTransaction =
      LeakChaser.createTransaction(
          Transaction.MODE_INFERRING, 90);
    LeakChaser.registerTransactionObject(
        workTransaction, this);
    LeakChaser.startTransaction(workTransaction);

    for (int i = 0; i < iterations; i++) {
      Garbage o = new Garbage(i);
      Garbage o2 = new Garbage(i);
      String s = o.getY();
      if (i \% (r.nextInt(iterations) + 1) == 0
          || s.length() == i) {
        doSomeWork(o);
        doMoreWork(o2);
    } }
    LeakChaser.endTransaction(workTransaction);
  } }
```

Listing 6.2 shows the class `GarbageProducer`, which creates instances of `Garbage` and randomly leaks them via static fields `randomLeakingGarbage` (line 12) and `anotherRandomLeakingGarbage` (line 14). Instances of `Gar-bage` are leaked randomly and most of the instances do not survive. This is required to create some work for the garbage collector and model the situation where only a fraction of all created objects is leaking and also guarantee that this synthetic test case will not run out of memory after several first garbage collections. From the formal point of view, when one instance of `Garbage` class is leaked, four objects are leaked in total – `Garbage` itself and its three fields: `int[] x`, `String y`, and `Date[] d`. When all allocation points in the current example are distinguished, then there are five leaking allocations: in listing 6.1 lines 2,3,4 and in listing 6.2 lines 31, 32.

Code on lines 23. . . 28 and 39 is required for the LeakChaser to specify *transactions* during the *Tier H* analysis (see section 3.3 for the description of the LeakChaser algorithm), when precise source of the leak is not known and the code is being explored for the first time. The paper claims that to mark the coarse-grained transaction boundaries on the high level of the approach no knowledge is required about the system. However, marking such transaction boundaries across the large application is highly unfeasible because of the amount of work. So, to find the leak using LeakChaser in a large multi-module application, at least an indication of the module responsible for the leak is required, meaning that at least some knowledge about the source of the problem is required. In contrast, proposed statistical approach implies no prerequisite knowledge about the source of leak, as the whole application is monitored for potential violators of the weak generational hypothesis.

After running the synthetic test application with LeakChaser, it produced the report containing all five leaking allocations. Part of the full report is show in listing 6.3 with line numbers fixed to match the source code in listings 6.1 and 6.2. Part of the report in the listing shows the leak details for instances of the `Garbage` class created at the line 32.

Listing 6.3: Part of the LeakChaser output, line numbers adjusted

```
 1  Transaction specified at:
 2  Class: leakchaser.GarbageProducer
 3  Method: <init>(ILjava/util/Random;)V
 4  Line Num: 31
 5  Violating objects created at:
 6  Class: leakchaser.GarbageProducer
 7  Method: <init>(ILjava/util/Random;)V
 8  Line Num: 32
 9  Violation type:
10  objects shared among transactions are stale
11  Frequency: 13
12  #Object reference paths: 1
13  Path 1, frequency 13, root obj referenced in method:  No info
14  --> Object type:Ljava/util/ArrayList;; Creation site: 14@leakchaser.GarbageProducer: <clinit
        >()V
15  --> Object type:[Ljava/lang/Object;;
```

While the report is correct, the depth of the path, displayed on lines 14 and 15, is by default limited to 5 elements. In case of field `anotherLeakingGarbage` which is an instance of `LinkedList`, whole chain of references constists of references between `LinkedList$Entry` elements. Another issue to note is how the holding reference is identified on line 14. While in this simple synthetic test the container is initialized and assigned to a field on the same line, in more complex applications initialization of the holding container may occur in a site unrelated the class containing the field the leaking container is held in.

In addition to this simple test case, mostly used to verify that Jikes RVM of the correct version is compiled and LeakChaser works, one real-world case study was also conducted. One of the tests used to analyze performance of Plumbr was used with LeakChaser – HtmlUnit. As was described in section 6.3, the memory leak of HtmlUnit was buried inside of the frameworks' code. To keep the test case close to the real life, the transaction boundaries for the LeakChaser were specified only in the user code which was calling the HtmlUnit framework.

For this use case Plumbr reported just one allocation requiring attention, as shown in image 4.2. As a result of a test execution, 47 leaking allocations were reported by LeakChaser, and none of the reported allocations were in fact relevant to the real memory leak. Based on its output it would be very hard to narrow down the scope for further diagnosis of the memory leak.

As a conclusion it must be noted that while targeted for developers, LeakChaser may be a good tool to diagnose a memory leak, given an insight where to narrow the focus.

6.5 Summary

This chapter described the case studies that were performed to evaluate detection performance along with runtime overhead of the Plumbr, implementation of the statistical approach for memory leak detection. Three case studies explained how existing leaks in ActiveMQ framework and one large web-application were verified with Plumbr and one new leak in HtmlUnit framework was detected during development and experimentation.

Last case study was conducted to compare proposed approach with an existing state of the art solution for memory leak detection. Most novel approach was selected among contenders – LeakChaser. Finding and fixing the leak consists of two distinct phases: detection and diagnosis. While the detection says what is leaking, diagnosis must find out how to fix it. As the statistical approach showed, while the detection can be automated in such a way that the developer is presented with the root cause of the problem, the diagnosis of the problem still has to be performed by the developer. It cannot be automatically prescribed what are the correct invariants for the program and what are the correct lifetime boundaries for particular objects. Approaches like LeakChaser can take over and help the developer where Plumbr leaves.

Chapter 7

Conclusions and Future Research Directions

Memory leak detection has been studied for the past 18 years, ever since Java was released. The solutions proposed range from low-overhead, low-intrusive, manual offline methods to high-overhead, high-intrusive, high-precision automatic online methods. Some of the proposed high-precision automatic online approaches require modifications of the JVM to work. If we look at the JVMs used in the industry – Oracle HotSpot/OpenJDK, Oracle jRockit, IBM v9, Azul Zing – none of them include GC-modifying approaches developed by scientists.

Solutions that rely on modification of the JVM probably will not ever reach real production JVMs, as all detection mechanisms provide only a certain probability that some objects are no longer used, and this functionality is inherently a tool's responsibility rather than part of JVM.

While early JVM implementation lacked support for third-party tooling to provide hooks into the JVM and garbage collection needed for online memory leak detection, modern JVMs provide powerful APIs to support monitoring and instrumentation combined with extremely easy installation of the tools. Java agents and native agents using JVMTI API can be used to implement powerful low-level and low-intrusive tooling which can use benefits promised by the online methods described in scientific publications. The latest publications also show that usage of JVM agents is also being adopted by the scientists instead of integrating tooling directly into the JVM or garbage collectors.

It should also be noted that there is no common measurement methodology across different approaches, which makes the comparison of leak detection quality complicated.

Author of the thesis believes that design goals such as general applicability, low intrusiveness, and reasonable overhead are really important for creating an algorithm which could be implemented in a standalone tool that can be used without modifying the application, JVM, or environment and yet produce readable reports for a programmer who is not a performance tuning professional but a regular programmer who just encountered an `OutOfMemoryError` in the production environment and just wants to get it fixed and continue with development, rather than spend days debugging.

Current thesis proposes to use the statistical metric, expressing the number of different generations of objects grouped by allocation sites, called $genCount$, which in combination with weak generational hypothesis allows to abstract such expensive metric as object staleness – an important attribute indicating a memory leak. The initial hypothesis stated that the unbounded growth of the $genCount$ metric of an allocation site indicates the source of the memory leak. Once the leak candidates are identified, additional analysis is performed to identify a chain of references holding onto leaked objects and merge reference chains for the leaking object subtree. To validate the hypothesis, the approach was implemented in a tool called Plumbr, which was made available for public use. The tool was also used to collect usage and feedback data. Although the analysis of false positives revealed several scenarios where a detection algorithm could benefit from enhancements, the $genCount$ parameter alone correctly identified the leak in 70% of the applications with an F_1-score of 0.71. However, on the allocation site level performance of the approach had an F_1-score of 0.39.

Further analysis identified the reasons for false positive detection results, as no clear separation of the leaking *genCount*, uniformity of *genCount* distribution, limited allocation context, impact of application load and uptime. Machine learning was used to further enhance the approach. Five additional metrics were specified and C4.5, PART and RandomForest classifiers were used to analyze the data set acquired from Plumbr users, consisting of carefully selected 200 different applications. Application of classifiers to identified new metrics resulted in an F_1-score of 0.797 on an allocation site level.

Apart from these, analysis of the use cases revealed several interesting observations about the behavior of the application itself when a memory leak is in progress. When the amount of Java heap used by the application begins approaching the limit set for the JVM, most of the time is spent in garbage collection, as can be concluded from garbage collection logs. Because of that, the performance of the application is reduced dramatically. As was shown in section 6.2, such performance degradation starts long before the application crashes with `OutOfMemoryError` or the GC overhead limit set by JVM ergonomics is reached [Java]. Which means that "Time to crash" cannot really be used as a reliable measure of application performance and that of performance tuning results. Only direct measures of request service time or, in the case of cloud applications and/or applications with a more distributed request-serving pipeline, time spent in GC as a ratio of total CPU time, will be an adequate assessment of an application performance.

Following section summarizes contributions of the thesis and outlines future research directions.

7.1 Future research

Modern complex and distributed applications require new approaches for performance management and monitoring. Monitoring data must not only be gathered and visualized, but rather interpreted. Current thesis showed that modern tools for data analysis like machine learning can be successfully applied to make sense of the data acquired from application monitoring. It allows to build application-aware monitoring tools which account for application specific behaviour automatically, with little specification required from a human. Such approach makes it possible to perform complex troubleshooting tasks without heavy-weight debugging and profiling tools, but rather measuring right metrics and making correct conclusions out of such data, possibly giving instructions for solving the problem.

Future research include, but is not limited to, discovering new types of performance problems, which may be analyzed in a similar way by applying machine learning for the detection and diagnosis. Statistical data about object lifetimes and allocation behaviors, gathered from Plumbr users, allows conducting further research to investigate lifetime patterns of long-lived objects.

In the scope of memory leak detection, still unaddressed is the problem of factory methods and lack of context which produces most of the false positives. Efficient methods for collection and encoding of stack traces are required to solve the problem. Engineering research topics also include performance optimizations for the current implementation which would further reduce the runtime overhead, that can be very noticeable in applications exhibiting high object allocation rates.

Feasibility of the application of the approach on other platforms, like Android or .NET must also be studied. In addition, interpretation of the leak detection results in the context of other dynamic programming languages executing on the JVM, like Groovy, Scala, Clojure, etc. is yet to be analyzed.

Acronyms

ACM	Association for Computer Machinery
AOP	Aspect Oriented Programming
API	Application Programming Interface
AUC	Area Under the Curve
CMS	Concurrent Mark and Sweep
CPU	Central Processing Unit
FN	False Negative
FP	False Positive
GB	Gigabyte
GC	Garbage Collection
GPL	General Public License
GUI	Graphical User Interface
HPROF	Heap Profiler
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IBM	International Business Machines
IDE	Integrated Development Environment
JDK	Java Development Kit
JIT	Just In Time Compiler
JMS	Java Messaging Service
JNI	Java Native Interface
JVM	Java Virtual Machine
JVMPI	Java Virtual Machine Profiling Interface
JVMTI	Java Virtual Machine Tooling Interface
LVB	Loaded Value Barrier
MAT	Memory Analyzer Tool
RAM	Random Access Memory
ROC	Receiver Operating Characteristic
RVM	Research Virtual Machine

SOT	Swap Out Table
SSD	Solid State Drive
TN	True Negative
TP	True Positive
VM	Virtual Machine

List of Publications

1. Šor, V., Srirama, S. N.: A Statistical Approach For Identifying Memory Leaks In Cloud Applications, First International Conference on Cloud Computing and Services Science (CLOSER 2011), May 7-9, 2011, pp. 623–628. SciTePress. ISBN: 978-989-8425-52-2, [vSor 11a].

   The author contributed the idea, the description, the architecture, the implementation and performed the benchmarking of the approach.

2. Šor, V., Salnikov-Tarnovski, N., Srirama, S. N.: Automated Statistical Approach for Memory Leak Detection: Case Studies (OTM 2011), Oct. 17-21, Springer Verlag, 2011, (Lecture Notes in Computer Science; 7045, Part II), pp. 635–642. ISBN: 978-3-642-25105-4, ISSN: 0302-9743, DOI: 10.1007/978-3-642-25106-1, [vSor 11b].

   The author contributed the methodology and the results for the ActiveMQ case study and the implementation of the tool.

3. Šor, V., Srirama, S. N.: Evaluation of embeddable graph manipulation libraries in memory constrained environments, 2012 ACM Research in Applied Computation Symposium (RACS 2012), Oct. 23-26 2011, pp. 269–275, ACM ISBN 978-1-4503-1492-3, [vSor 12].

   The author contributed the the methodology of evaluation and the evaluation of graph libraries used in comparison.

4. Šor, V., Srirama, S. N., Treier, T.: Improving statistical approach for memory leak detection using machine learning, 29th IEEE International Conference on Software Maintenance (ICSM 2013), Sep. 22-28 2013, pp. 544–547, IEEE, [vSor 13].

   The author contributed to the design of learning features, design, implementation and deployment of the infrastructure for collection of statistical snapshots from applications and to the extraction of the learning data from collected statistical snapshots.

5. Šor, V., Srirama, S. N., Salnikov-Tarnovski, N.: Memory leak detection in Plumbr, Journal of: Software: Practice and Experience (2014), DOI: 10.1002/spe.2275, [vSor 14b].

   The author contributed to the implementation of the Plumbr tool, formalized the mathematical definitions used to describe main principles of the statistical approach, analyzed the detection and runtime performance, listed the deficiencies of the method and ran the benchmarks.

6. Šor, V., Srirama, S. N.: Memory Leak Detection in Java: Taxonomy and Classification of Approaches, The Journal of Systems & Software (2014), pp. 139–151, DOI: 10.1016/j.jss.2014.06.005, [vSor 14a].

   The author contributed the taxonomy and the classification along with the analysis of memory leak detection approaches.

Bibliography

[Acti 11] **Active MQ issue tracker, ticket 3021**. 2011. `https://issues.apache.org/jira/browse/AMQ-3021`, [URL last visited on 11th June 2014]. 113, 114

[Amaz 14] AMAZON INC. **Elastic Compute Cloud (EC2)**. 2014. `http://aws.amazon.com/ec2/`, [URL last visited on 11th June 2014]. 113

[Apac 11a] APACHE SOFTWARE FOUNDATION. **Apache Software Foundation issue tracker**. 2011. `https://issues.apache.org/bugzilla/`, [URL last visited on 11th June 2014]. 111

[Apac 11b] APACHE SOFTWARE FOUNDATION. **JMeter**. 2011. `http://jakarta.apache.org/jmeter/`, [URL last visited on 11th June 2014]. 117

[Apac 14] APACHE SOFTWARE FOUNDATION. **ActiveMQ**. 2014. `http://activemq.apache.org/`, [URL last visited on 11th June 2014]. 113

[Blac 06] STEPHEN M. BLACKBURN, ROBIN GARNER, CHRIS HOFFMANN, ASJAD M. KHANG, KATHRYN S. MCKINLEY, ROTEM BENTZUR, AMER DIWAN, DANIEL FEINBERG, DANIEL FRAMPTON, SAMUEL Z. GUYER, MARTIN HIRZEL, ANTONY HOSKING, MARIA JUMP, HAN LEE, J. ELIOT B. MOSS, AASHISH PHANSALKAR, DARKO STEFANOVIĆ, THOMAS VANDRUNEN, DANIEL VON DINCKLAGE, AND BEN WIEDERMANN. **The DaCapo benchmarks: java benchmarking development and analysis**. In: *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pp. 169–190, ACM, New York, NY, USA, 2006. 97

[Bond 06] Michael D. Bond and Kathryn S. McKinley. **Bell: Bit-Encoding Online Memory Leak Detection**. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25*, pp. 61–72, 2006. 42, 43, 44, 62, 64, 65, 71, 122

[Bond 08] Michael D. Bond and Kathryn S. McKinley. **Tolerating memory leaks**. In: *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pp. 109–126, 2008. 43, 44, 62, 64, 66, 71

[Bond 09] Michael D. Bond and Kathryn S. McKinley. **Leak pruning**. In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pp. 277–288, 2009. 44, 62, 64, 67, 71

[Boos] **Boost C++ Libraries**. `http://www.boost.org/`, [URL last visited on 11th June 2014]. 54

[Brei 01] Leo Breiman. **Random Forests**. *Machine Learning*, Vol. 45, No. 1, pp. 5–32, 2001. 38, 108

[Brei 07] David Breitgand, Maayan Goldstein, Ealan Henis, Onn Shehory, and Yaron Weinsberg. **PANACEA Towards a Self-healing Development Framework**. In: *Integrated Network Management, IM 2007. 10th IFIP/IEEE International Symposium on Integrated Network Management, Munich, Germany, 21-25 May 2007*, pp. 169–178, 2007. 50, 62, 64, 66, 71

[Chen 07] Kung Chen and Ju-Bing Chen. **Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs**. In: *31st Annual International Computer Software and Applications Conference (COMPSAC 2007), 24-27 July 2007, Beijing, China*, pp. 23–28, 2007. 48, 62, 64, 65, 71

[De P 99] Wim De Pauw and Gary Sevitsky. **Visualizing Reference Patterns for Solving Memory Leaks in Java**. In: Rachid Guerraoui, editor, *ECOOP' 99 — Object-Oriented Programming*, pp. 668–

668, Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48743-3_6. 52, 54, 62, 64, 71

[Detl 04] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. **Garbage-first Garbage Collection**. In: *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, pp. 37–48, ACM, New York, NY, USA, 2004. 31

[Dist 10] Dino Distefano and Ivana Filipović. **Memory Leaks Detection in Java by Bi-abductive Inference**. In: David S. Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pp. 278–292, Springer Berlin Heidelberg, 2010. 17, 56, 62, 64, 71

[Ecli 13] Eclipse foundation. **AspectJ Home page**. 2013. `http://eclipse.org/aspectj/`, [URL last visited on 11th June 2014]. 48

[Form 06] Ian Formanek and Gregg Sporar. **Dynamic Bytecode Instrumentation**. *Dr. Dobbs Journal*, Vol. Online, 2006. `http://www.drdobbs.com/tools/184406433`, [URL last visited on 11th June 2014]. 75

[Fran 98] Eibe Frank and Ian H. Witten. **Generating Accurate Rule Sets Without Global Optimization.** In: *Proceedings of the Fifteenth International Conference on Machine Learning (ICML 1998), Madison, Wisconsin, USA, July 24-27, 1998*, pp. 144–151, Morgan Kaufmann, 1998. 38, 108

[Gene] *Generational Concurrent Garbage Collector.* IBM Corporation. `http://www-01.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/com.ibm.java.doc.diagnostics.60/diag/understanding/mm_gc_generational.html?lang=en`, [URL last visited on 11th June 2014]. 32

[Goet 05] Brian Goetz. **Java theory and practice: Plugging memory leaks with weak references**. Online, november 2005. `http://www.`

`ibm.com/developerworks/java/library/j-jtp11225/`, [URL last visited on 11th June 2014]. 58

[Gold 07] Maayan Goldstein, Onn Shehory, and Yaron Weinsberg. **Can self-healing software cope with loitering?** In: *Fourth International Workshop on Software Quality Assurance, SOQUA 2007, in conjunction with the 6th ESEC/FSE joint meeting, Dubrovnik, Croatia, September 3-4, 2007*, pp. 1–8, ACM, New York, NY, USA, 2007. 50, 62, 64, 66, 71

[Hall 09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. **The WEKA data mining software: an update**. *SIGKDD Explorations*, Vol. 11, No. 1, pp. 10–18, 2009. 108

[Haus 04] Matthias Hauswirth and Trishul M. Chilimbi. **Low-overhead memory leak detection using adaptive statistical profiling**. In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pp. 156–164, 2004. 43

[Html 11] HtmlUnit. **HtmlUnit homepage**. may 2011. `http://htmlunit.sourceforge.net/`, [URL last visited on 11th June 2014]. 119

[Java] *Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning.* Oracle corp., http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html. `http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html`, [URL last visited on 11th June 2014]. 29, 30, 130

[Jone 11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 1st Ed., 2011. 14, 15, 23, 26, 27, 28

[Jump 06] M. Jump and K. S. McKinley. **Cork: Dynamic Memory Leak Detection for Java, Technical Report TR-06-07**. Tech. Rep., The University of Texas at Austin, January 2006. 48, 62, 64, 65, 71

[Jump 07] Maria Jump and Kathryn S. McKinley. **Cork: dynamic memory leak detection for garbage-collected languages**. In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles* *of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pp. 31–38, ACM, 2007. 48, 62, 64, 65, 71

[Kicz 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. **Aspect-oriented programming**. *ECOOP'97—Object-Oriented Programming*, pp. 220–242, 1997. 47

[Leng 79] Thomas Lengauer and Robert Endre Tarjan. **A fast algorithm for finding dominators in a flowgraph**. *ACM Trans. Program. Lang. Syst.*, Vol. 1, No. 1, pp. 121–141, Jan. 1979. 54

[Lian 99] Shen Liang. *The Java^{TM}Native Interface Programmer's Guide and Specification.* Sun Microsystems Inc., 1999. 83

[Lieb 83] Henry Lieberman and Carl Hewitt. **A real-time garbage collector based on the lifetimes of objects**. *Commun. ACM*, Vol. 26, No. 6, pp. 419–429, June 1983. 28

[Lind 13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition.* Addison-Wesley Professional, 1st Ed., 2013. 21, 29

[Mann 08] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval.* Vol. 1, Cambridge university press Cambridge, 2008. 36

[Maxw 10] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. **Diagnosing Memory Leaks using Graph Mining on Heap Dumps**. In: *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, pp. 115–124, 2010. 52, 53, 62, 64, 71

[McCa 60] John McCarthy. **Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I**. *Commun. ACM*, Vol. 3, No. 4, pp. 184–195, Apr. 1960. 26

[Memo 11] **Memory leak discussion on HtmlUnit user mailing list**. Online, June 2011. `http://old.nabble.com/memory-usage-problems-tt28907672.html#post28907786`, [URL last visited on 11th June 2014]. 113, 120

[Mila 05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. **Parameterized Object Sensitivity for Points-to Analysis for Java**. *ACM Trans. Softw. Eng. Methodol.*, Vol. 14, No. 1, pp. 1–41, Jan. 2005. 95

[Mitc 03] Nick Mitchell and Gary Sevitsky. **LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications**. In: *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, pp. 351–377, Springer, 2003. 52, 62, 64, 71

[Mitc 97] Thomas M. Mitchell. *Machine Learning.* McGraw-Hill, Inc., New York, NY, USA, 1 Ed., 1997. 34

[Orac 13] Oracle Corp. **java.lang.ref package summary**. 2013. `http://docs.oracle.com/javase/1.5.0/docs/api/java/\lang/ref/package-summary.html`, [URL last visited on 11th June 2014]. 24

[Qian 12] Ju Qian and Xiaoyu Zhou. **Inferring Weak References for Fixing Java Memory Leaks**. In: *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, pp. 571–574, IEEE Computer Society, 2012. 58, 59, 62, 64, 67, 71

[Quin 93] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993. 37, 108

[Rays 06] Derek Rayside, Lucy Mendel, and Daniel Jackson. **A dynamic analysis for revealing object ownership and sharing**. In: *Proceedings of the 2006 international workshop on Dynamic systems analysis*, pp. 57–64, ACM, New York, NY, USA, 2006. 62, 64, 65, 71

[Rays 07] Derek Rayside and Lucy Mendel. **Object ownership profiling: a technique for finding and fixing memory leaks**. In: *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pp. 194–203, ACM, 2007. 45, 62, 64, 65, 71

[Reis 09] Steven P. Reiss. **Visualizing the Java Heap to Detect Memory Problems**. In: *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2009, September 25, 2009, Edmonton, Alberta, Canada*, pp. 73–80, 2009. 52, 55, 62, 64, 71

[Shah 00] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. **Automatic Removal of Array Memory Leaks in Java**. In: David A. Watt, editor, *Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, Arch 25 - April 2, 2000, Proceedings*, pp. 50–66, Springer Berlin Heidelberg, 2000. 17, 57, 62, 64, 71

[Sun 06] Sun Microsystems Inc. **JVM^{TM} Tool Interface**. 2006. `http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html`, [URL last visited on 11th June 2014]. 82

[Tang 08] Yan Tang, Qi Gao, and Feng Qin. **LeakSurvivor: towards safely tolerating memory leaks for garbage-collected languages**. In: *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*, pp. 307–320, USENIX Association, 2008. 43, 62, 64, 66, 71

[Tene 11] Gil Tene, Balaji Iyengar, and Michael Wolf. **C4: the continuously concurrent compacting collector**. In: *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, pp. 79–88, 2011. 33

[The 10] The Jikes RVM Project. **Home page**. 2010. `http://jikesrvm.org/`, [URL last visited on 11th June 2014]. 33, 43, 44, 45, 49

[The 11] The Eclipse Foundation. **The Eclipse Memory Analyzer**. 2011. `http://www.eclipse.org/mat/`, [URL last visited on 11th June 2014]. 52

[Tiob 14] Tiobe Software BV. **TIOBE Programming Community Index**. 2014. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, [URL last visited on 11th June 2014]. 15

[Tuni] *Tuning the Memory Management System.* Oracle Corporation. `http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/memman.html#wp1087125`, [URL last visited on 11th June 2014]. 33

[Unga 84] David Ungar. **Generation Scavenging: A non-disruptive high performance storage reclamation algorithm**. *SIGSOFT Softw. Eng. Notes*, Vol. 9, No. 3, pp. 157–167, Apr. 1984. 29

[vSor 11a] Vladimir Šor and Satish Narayana Srirama. **A Statistical Approach For Identifying Memory Leaks In Cloud Applications**. In: *CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science, Noordwijkerhout, Netherlands, 7-9 May, 2011*, pp. 623–628, SciTePress, 2011. 20, 49, 62, 64, 66, 71, 134

[vSor 11b] Vladimir Šor, Satish Narayana Srirama, and Nikita Salnikov-Tarnovski. **Automated Statistical Approach for Memory Leak Detection: Case Studies**. In: Robert Meersman, Tharam Dillon, Pilar Herrero, Akhil Kumar, Manfred Reichert, Li Qing, Beng-Chin Ooi, Ernesto Damiani, DouglasC. Schmidt, Jules White, Manfred Hauswirth, Pascal Hitzler, and Mukesh Mohania, editors, *On the Move to Meaningful Internet Systems: OTM 2011 - Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2011, Hersonissos, Crete, Greece, October 17-21, 2011, Proceedings, Part II*, pp. 635–642, Springer Verlag, 2011. 20, 49, 62, 64, 66, 71, 80, 113, 134

[vSor 12] Vladimir Šor and Satish Narayana Srirama. **Evaluation of embeddable graph manipulation libraries in memory constrained environments**. In: *Research in Applied Computation Symposium, RACS '12, San Antonio, TX, USA, October 23-26, 2012*, pp. 269–275, ACM, New York, NY, USA, 2012. `http://doi.acm.org/10.1145/2401603.2401663`. 87, 134

[vSor 13] Vladimir Šor, Tarvo Treier, and Satish Narayana Srirama. **Improving statistical approach for memory leak detection using machine learning**. In: *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pp. 544–547, IEEE, 2013. 20, 49, 135 [vSor 14a] Vladimir Šor and Satish Narayana Srirama. **Memory Leak Detection in Java: Taxonomy and Classification of Approaches**. *Journal of Systems and Software*, Vol. DOI: 10.1016/j.jss.2014.06.005, pp. 139–151, 2014. 19, 40, 135

[vSor 14b] Vladimir Šor, Satish Narayana Srirama, and Nikita Salnikov-Tarnovski. **Memory leak detection in Plumbr**. *Software: Practice and Experience*, Vol. DOI:10.1002/spe.2275, 2014. 20, 49, 62, 64, 66, 71, 135

[Xu 08] Guoqing (Harry) Xu and Atanas Rountev. **Precise memory leak detection for java software using container profiling**. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pp. 151–160, 2008. 46, 62, 64, 65, 71

[Xu 11] Guoqing (Harry) Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. **LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks**. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011. 57, 58, 62, 64, 67, 71, 122

[Xu 13] Guoqing (Harry) Xu and Atanas Rountev. **Precise Memory Leak Detection for Java Software Using Container Profiling**. *ACM Trans. Softw. Eng. Methodol.*, Vol. 22, No. 3, pp. 17:1–17:28, July 2013. 46, 62, 64, 65, 71

[Yan 14] Dacong Yan, Guoqing (Harry) Xu, Shengqian Yang, and Atanas Rountev. **LeakChecker: Practical Static Memory Leak Detection for Managed Languages**. In: *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14, Orlando, FL, USA, February 15-19, 2014*, p. 87, 2014. 56, 62, 64, 71

Statistiline lähenemine mälulekete tuvastamiseks Java rakendustes

Kokkuvõte

Kaasaegsed hallatud käitusaja keskkonnad (ingl. *managed runtime environment*) ja programmeerimiskeeled lihtsustavad rakenduste loomist ning haldamist. Klassikaliste programmeerimiskeeltega võrreldes on hallatud käitlusaja keskkondades programmeerijal vaja vähem tegeleda madalatasemelisteste probleemidega. Käitusaja keskkond jooksutab koodi eraldi turvatud keskkonnas (ingl. *sandbox*) ning võimaldab käivitada üks kord kompileeritud programmi erinevatel platvormidel ja operatsioonisüsteemidel. Kõige levinumaks näiteks säärase keele ja keskkonna kohta on Java.

Klassikalistes programmeerimiskeeltes peab programmeerija ise hoolitsema operatsioonisüsteemilt tarvilikku mälu koguse küsimise (e. allokeerimise) ning hilisema vabastamise (e. deallokeerimise) eest. Mälu allokeerimise päringu tulemusena antakse rakendusele tagasi viide allokeeritud mälule. Allokeerimise/deallokeerimise operatsioonide väärkasutamine on suure hulga programmeerimisvigade põhjustajateks. Üheks sellise vea

näiteks on mäluleke ehk olukord, kus rakendus küll allokeerib mälu, kuid ei deallokeeri seda pärast kasutamist. Mälulekkel on kaks erinevat tekkepõhjust: tarbetu mälu hoidmine ja kaotatud viide:

- Tarbetu mälu hoidmise puhul on viide allokeeritud mälule lihtsalt unustatud konkreetsesse muutujasse. Selle viite unustamise tõttu on mälu rakendusele kättesaamatu, kuigi reaalselt antud mälu enam ei kasutata.

- Kaotatud viite puhul kirjutatakse viidet hoidev muutuja üle enne mälu deallokeerimist. Selles olukorras on eraldatud mälu kadunud kuni rakenduse tööaja lõpuni, mil operatsiooni süsteem vabastab kogu rakenduse poolt allokeeritud mälu. Kui sellisel moel lekib mälu piisavalt palju, võib kogu olemasolev mälu otsa saada. Mälu otsasaamisel võib seiskuda nii rakendus kui halvimal juhul ka operatsioonisüsteem.

Üheks tähtsaks hallatud käitusaja keskkonna ülesandeks on automaatne mäluhaldus. Automaatne mäluhaldus hoolitseb mälu allokeerimise ja deallokeerimise eest. Kui mälu allokeerimiseks on piisav, kui programmis luuakse uus objekt, siis deallokeerimise eest hoolitseb eraldi moodul – prügikoristaja (ingl. *garbage collector*). Prügikoristaja kustutab mälust kõik objektid millele ei viita ükski teine objekt. Seega elimineeritakse üks eelmainitud mälulekke põhjustest – kaduma läinud viide.

Teine võimalik mälulekke allikas (tarbetu mälu hoidmine) on endiselt relevantne. Probleemi teeb eriti oluliseks fakt, et Javale antav mälu on rangelt piiratud, mistõttu mäluleke on üks väheseid programmeerimisvigu, mis võib hävitada kogu Java käitusaja keskkonna protsessi koos kõikide seal töötavate programmidega. Mäluleke on selles aspektis eriline – tänapäevaste raamistike kõrge tõrkekindlus tagab muude tekkida võivate erindite osas Java käitlusaja keskkonna töö jätkamise. Siit järeldub, et probleem on eriti kriitiline rakendustes, mis peaksid ööpäevaringselt tõrgeteta toimima.

Vaatamata sellele, et mälulekete probleemi on uuritud Java keele ning selle käitluskeskkonna tekkimise algusaegadest saadik, pole sellele siiani pakutud lihtsat tuvastus- ning diagnostikalahendust. Käesolev väitekiri uurib mälulekete problemaatikat Javas ning pakub välja mälulekkeid tuvastava ning diagnoosiva algoritmi. Lahenduse loomisel on olulise aspektina silmas peetud jõudluse ülekulu – pakutud lähenemist peab saama kasutada ka toodangukeskkondades (ingl. *production environments*).

Kuna sisseehitatud prügikoristajaga Javas tähendab mäluleke tarbetut mälu hoidmist, siis heaks indikaatoriks otsustamaks, kas objekt on kasutuses või mitte, on objekti viimane kasutusaeg. Objekti viimase kasutuse aega mõõdavad ning kasutavad indikaatorina mitmed teised uurimistööd. Selle meetrika põhiliseks puudujäägiks on selle hind jõudluse mõttes – selleks et teada, millal objekti on viimati kasutatud, peab iga kasutuskorra (nii kirjutamise kui ka lugemise) puhul uuendama seda kasutust jälgivaid lippe või muutujaid. Nii toimides kaasneb iga lugemisoperatsiooniga ka ühe mäluaadressi kirjutamine, mis ainult tarkvaraliste vahenditega on jõudluse mõttes liiga kallis. Jõudlusprobleemid on ka põhjuseks, miks sääraste meetodite rakendatavus on piiratud arenduskeskkondadega ja tehisnäidetega.

Käesolev väitekiri pakub alternatiivset lähenemisviisi objektide mittekasutuse hindamiseks. Töö põhimõte baseerub ühel kaasaegsete prügikoristajate optimiseerimiste alustalaks oleval tähelepanekul. Nimelt elab enamus objekte prügikoristusega käitusaja keskkonnas väga lühikest aega. Väitekirja põhihüpoteesiks on idee, et lekkivaid objekte saab statistiliste meetoditega eristada mittelekkivatest, kui vaadelda objektide populatsiooni eluiga erinevate gruppide lõikes. Selliseks vaatluseks piisab, kui salvestada objektide loomise aeg, grupeerida objektid klassi ning loomise koha järgi ning pärast prügikoristust vaadelda, kuidas jagunevad elus olevate objektide vanused.

Pakutud lähenemine on oluliselt odavama hinnaga jõudluse mõttes – põhieeliseks on, et objekti kohta on vaja salvestada infot ainult selle loomise hetkel. Hilisema kasutuse jooksul on episoodiliselt tarvis analüüsida eraldiseisvat andmestruktuuri, muutmata rakendust ennast. Väitekirja uurimistöö tulemusi on rakendatud mälulekete tuvastamise tööriista Plumbr arendamisel.

Pärast sissejuhatavaid peatükke 1 ja 2 on käesoleva väitekirja 3. peatükis vaadeldud siiani pakutud lahendusi ning on pakutud välja ka mälulekete tuvastamise meetodite klassifikatsioon. 4. peatükis on kirjeldatud statistiline meetod mälulekete tuvastamiseks ning põhimõõdik, mille abil on võimalik eristada lekkivaid objekte mittelekkivatest. Lisaks on analüüsitud ka kirjeldatud põhimõõdiku puudujääke. 5. peatükis on kirjeldatud kuidas masinõppe abil said defineeritud lisamõõdikud, mis aitasid pakutud mälulekete tuvastamise meetodit täpsemaks teha. Testandmeid masinõppe tarbeks on kogutud Plumbri abil päris rakendustest ning toodangkeskkondadest. 6. peatükk kirjeldab juhtumianalüüse, mille abil on hinnatud lahenduse mõju vaadledavate rakenduste jõudlusele. Lisaks on teostatud ka võrdlev juhtumianalüüs ühe olemasoleva mälulekete tuvastamise lahendusega.

Curriculum Vitae

Name:	Vladimir Šor
Date of Birth:	14.09.1979
Citizenship:	Estonian

Education:

2007 – 2014	University of Tartu, Faculty of Mathematics and Computer Science, doctoral studies, specialty: computer science.
2001 – 2003	University of Tartu, Faculty of Mathematics and Computer Science, master studies, specialty: computer science.
1997 – 2001	University of Tartu, Faculty of Mathematics and Computer Science, bachelor studies, specialty: computer science.

Work experience:

2011 –	Plumbr OÜ, CTO
2001 – 2011	AS Webmedia, programmer, systems architect

Elulookirjeldus

Nimi:	Vladimir Šor
Sünnikuupäev:	14.09.1979
Kodakonsus:	Eesti

Haridus:

2007 – 2014	Tartu Ülikool, Matemaatika-informaatikateaduskond, doktoriõpe, eriala: informaatika.
2001 – 2003	Tartu Ülikool, Matemaatika-informaatikateaduskond, magistriõpe, eriala: informaatika.
1997 – 2001	Tartu Ülikool, Matemaatika-informaatikateaduskond, bakalaureuseõpe, eriala: informaatika.

Teenistuskäik:

2011 –	Plumbr OÜ, CTO
2001 – 2011	AS Webmedia, programmeerija, süsteemiarhitekt