Generating Realistic Trace Files for Memory Management Simulators by Instrumenting IBM's J9 Java Virtual Machine

by

Johannes Ilisei

Bachelor of Engineering in Software Engineering,
Esslingen University of Applied Sciences, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Kenneth B. Kent, PhD, Faculty of Computer Science
	Gerhard Dueck, PhD, Faculty of Computer Science
Examining Board:	Eric Aubanel, PhD, Faculty of Computer Science, Chair
	David Bremner, PhD, Faculty of Computer Science
	Richard Tervo, PhD, Electrical and Computer Engineering

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

May, 2017

©Johannes Ilisei, 2017

Abstract

High-level programming languages like Java, C#, or Python rely on memory management systems that allocate and free objects automatically. A Java Virtual Machine (JVM) is responsible to execute compiled Java code. Several JVM implementations are available that include ongoing improvements throughout many years with reductions in execution time and memory footprint as well as the addition of new features. JVM implementations are large-sized projects that consist of many files, classes, and functions. Changing or extending the code can be a difficult and time consuming task. Therefore, simulators that reproduce desired JVM operations are available. They can be used to implement and test new features in little time. As with the Java Virtual Machine, a simulator requires instructions as form of input files with information on what operations to perform. These files are called trace files and they are generated with an instrumented JVM. Relevant operations are captured and printed into a file while running the JVM.

This master's thesis focuses on the generation of trace files that represent JVM operations as realistically as possible. At the start of this project, two

types of trace file generators already exist. Unfortunately, both of them contain errors that lead to a false JVM representation. Thereby, results gathered from simulators are unreliable. A new form of trace file generation is required that is able to produce correct inputs for a simulator. The project presented in this thesis captures JVM operations directly from IBM's J9 Java Virtual Machine's bytecode instructions. In addition, a comparison between previous and new trace files and their different effects on the simulator is part of this thesis.

Dedication

For my parents

Ioan Ilisei and Gunhild Ilisei

and my siblings

Samuel Ilisei and Lisandra Ilisei.

Acknowledgements

I would like to thank my advisors Prof. Kenneth Kent and Prof. Gerhard Dueck for their ongoing support and professional guidance throughout this degree. I would also like to thank Stephen MacKay for all his help he provided when I was writing this thesis. The continuous help regarding technical questions as well as the great work climate with my lab mates is much appreciated.

I would like to especially thank my family, who supported me throughout the whole course of my studies. Without their help, my degree as well as my own Canadian experience would not have been possible. I also thank Carolin Meier for giving me the motivation and love that I needed to keep on going and finishing my degree.

The funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program is gratefully appreciated.

Table of Contents

Abstract **ii**

Dedication **iv**

Acknowledgments **v**

Table of Contents **vi**

List of Tables **ix**

List of Figures **xi**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Statement 3

1.3 Thesis Organization 4

2 Background **5**

2.1 The Java Programming Language 5

2.2 The Java Virtual Machine 7

2.2.1 JVM Internals 11
2.3 Dynamic Memory Management 15
2.4 Garbage Collection (GC) Policies 17
2.4.1 Mark-sweep 18
2.4.2 Mark-compact 19
2.4.3 Copying GC 20
2.4.4 Reference Counting (RC) 24
2.5 The GarCoSim Framework 26
2.5.1 Trace Files 28
2.5.2 Post-Processing 32
2.5.2.1 Post Processor Version 1 33
2.5.2.2 Post Processor Version 2 37
2.5.3 Simulator Implementation 39

3 Searching for Zombies **42**
3.1 Raw Trace Files Examination 44
3.2 Missing Root Operations 48

4 Design **52**
4.1 Requirements 53
4.2 Instrumented JVM Version Bytecode 54
4.2.1 Monitoring Bytecode Instructions 54
4.2.2 Root References outside the Bytecode Interpreter . . . 57
4.2.3 Locking 58

4.2.4 Zeroing Out Stack Slots 59
4.3 Zombie Removal Processor 61

5 Experimental Setup 66

5.1 Instrumented JVM 66
5.2 Post Processor Version 1 70
5.3 Post Processor Version 2 and 3 70
5.4 GarCoSim Simulator 71
5.5 Zombie Detection Simulator 73
5.6 Zombie Removal Processor 74

6 Results 75

6.1 Removed Zombie Objects 75
6.2 Negative Root Set Count 76
6.3 Locking Statistics 77
6.4 Trace File Comparisons 78
6.4.1 File Size and Operation Count Statistics 78
6.4.2 Traversal Depth During GC 82
6.4.3 Root Count During GC 85

7 Conclusions and Future Work 87

Bibliography 93

Vita

List of Tables

2.1 The trace file instructions 29

3.1 Zombie objects of the DaCapo-9.12-bach benchmarks 46

4.1 Numbers of examined bytecode instructions and helper functions. 56

6.1 Remaining Zombie objects 76

6.2 Negative root pointer deletions 77

6.3 Locked and unlocked lines 78

6.4 batik Comparisons 80

6.5 fop Comparisons 80

6.6 luindex Comparisons 81

6.7 lusearch Comparisons 81

6.8 pmd Comparisons 81

6.9 xalan Comparisons 82

6.10 Traversal depth during GC 82

6.11 Root set pointers version 1 84

6.12 Root set pointers version 3 85

6.13 Average root count during GC 86

List of Figures

2.1 TIOBE Programming Community Index [5] 7
2.2 Executing Java Source Code 10
2.3 Java Virtual Machine Internals [8] 12
2.4 Copying GC Algorithm 23
2.5 Generating trace files 31
2.6 Traversal depth levels 37
2.7 Simulator Class Diagram 39

4.1 Example of zeroing out stack slots. 60
4.2 Final trace file generation process 62

6.1 Locking levels averaged over all benchmarks 79

Chapter 1

Introduction

1.1 Motivation

Java Virtual Machines (JVM) [15] are responsible for executing Java programs that have been compiled to the standard Java bytecode. These virtual machines are the connection between physical hardware and the virtual execution environment that supports device-independency. Java code can be executed on any device, independent of hardware or the operating system, as long as a JVM implementation is available.

The demand for Java-developers and software implemented in Java is greater than ever. According to the TIOBE index [5], Java is the most popular programming language since May 2015. In order to keep up with the fast growing and rapidly changing field of software development, the Java programming language as well as the underlying system, the JVM, have to develop further.

New features and improvements have to be provided with every new version in order to remain competitive.

Two of the largest manufacturers of JVMs are Oracle and IBM. Their JVM implementations are large-sized projects that have been around for decades and consist of many files, classes, functions, and much code. Every new release of the Java programming language entails a new JVM release. Research and improvements on Java Virtual Machines include reductions in execution time and memory footprint as well as the addition of new features and policies.

Therefore, working with these projects, getting an overview of the code and changing or extending it can be a difficult and time consuming task. Especially the outcome of new features or algorithms may be not beneficial enough for future use.

Java Virtual Machine simulators are able to reproduce desired JVM operations. They simulate a dynamic memory management system with a heap, allocation and mutation of objects, and they perform garbage collection. A memory management simulator has to perform only these operations, and thus is much simpler than a JVM implementation. Therefore, implementing and testing new features can be done in less time compared to a JVM implementation.

A simulator requires instructions as form of input files that contain operations that have to be performed. These files are called trace files and are roughly comparable to JVM's compiled bytecode. Trace files are generated

with an instrumented JVM. Relevant operations are captured during the execution of the JVM and are printed into trace files. They can also be created by using a synthetic trace file generator with a specified number and ratio of operations.

The goal of this master's thesis is to generate realistic trace files that reproduce JVM operations as closely as possible. The best simulator is not able to deliver trustworthy results if its input does not match a true JVM implementation.

The approach presented in this thesis captures operations from bytecode instructions. Therefore, the bytecode interpreter of IBM's J9 JVM was instrumented in order to capture references from the stacks to the heap (root-pointers). In case all root operations are directly performed by the bytecode interpreter, monitoring them can guarantee to find all root additions and deletions. Thus, it is possible to provide trace files that represent a JVM realistically.

1.2 Problem Statement

The main challenge of this project is not simply to generate trace files, it is to guarantee the correctness and comparability to the JVM. Two types of trace file generators preceded this work. The GarCoSim simulator was used for this project. It is able to execute and generates results with these trace files. Unfortunately, both trace file versions contain errors that lead to

a false JVM representation and thus results gathered from the simulator are unreliable. Their flaws are discussed in Section 2.5.2 and Chapter 3.

The approach presented in this thesis collects operations directly from bytecode instructions. All relevant functions, that are part of the bytecode interpreter, have to be examined. Operations have to be captured and printed into the trace file. Only if all reference creations and deletions are correctly monitored, can trace files be considered accurate.

1.3 Thesis Organization

This thesis consists of seven chapters. Chapter 2 gives the background of this work. This includes the Java programming language as well as the Java Virtual Machine. Dynamic memory management and garbage collection is explained on the basis of four standard collectors. Lastly, the GarCoSim Framework is presented with its simulator and trace file generation process. Groundwork about flaws in preceding trace files as well as the search for their errors is presented in Chapter 3. The design for a realistic trace file generator is laid out in Chapter 4 followed by its implementation and experimental setup in Chapter 5. Results are presented and evaluated in Chapter 6. Finally, the conclusion and proposal of future work is discussed in Chapter 7.

Chapter 2

Background

This chapter explains relevant information that is necessary to understand the remainder of this thesis as well as related topics about this project. The outline of the Java Programming language in Section 2.1 is followed by the Java Virtual Machine in 2.2. An overview of dynamic memory management in general is shown in Section 2.3. The four standard garbage collection policies mark-sweep, mark-compact, copying GC, and reference counting are described in 2.4. The GarCoSim Framework is an important part of this thesis and is presented lastly in Section 2.5.

2.1 The Java Programming Language

Java is a modern object-oriented programming language created by a team lead by James Gosling and was announced in 1995 [19]. Based on C/C++ [10],

it is classified as a high level language. Details of lower levels, like the underlying hardware and memory addresses, are invisible to the programmer. Furthermore, the task of allocating and freeing objects manually is of no concern to the programmers due to an automated memory management system. Compared to lower level languages such as C or Assembly, Java is easy to learn and advanced data structures can be built in little time. This is accomplished by many features, libraries, and APIs that are improved and extended with every new Java version. Additionally, an extensive amount of Java-compatible frameworks help developers. Java's latest edition, SE 8, introduces, among other new features, lambda expressions, a new javascript engine called Nashorn, as well as new date and time APIs [6].

According to the TIOBE index [5], Java was the most popular programming language in September 2016 with a popularity of about 19%. This index is calculated by multiple different measures, including search terms on Google, Bing, YouTube, or Yahoo!. Also, the number of professional programmers, available courses, and third party vendors are taken into account. An overview of the ten most popular programming languages over the last 15 years is shown in Figure 2.1. As displayed, between 2012 and early 2015, C was the most popular programming language ahead of Java. Only recently, Java was able to make it back to the top of the list. This is mainly due to the fast growing sector of smartphone applications for which Java is superior compared to C. After more than 20 years of existence, the importance and demand for applications written in Java is still highly relevant and not

expected to drop soon.

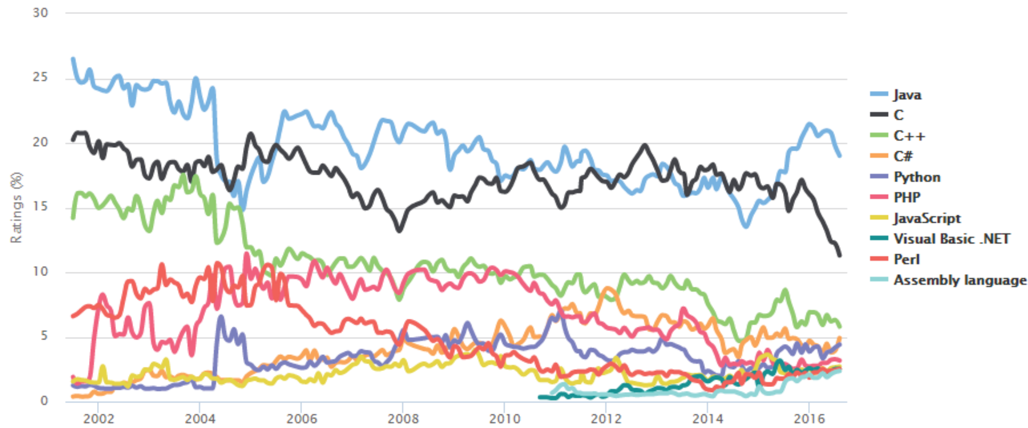

Figure 2.1: TIOBE Programming Community Index [5]

2.2 The Java Virtual Machine

In order to run a Java-program, a virtual machine is required that performs necessary operations and communicates with the underlying hardware. The Java Virtual Machine (JVM) is a computing machine and operates in similar manner as operating systems like Windows, Linux, or Mac OS. Whether a virtual machine is used to run an operating system (for example with VirtualBox) or compiled Java code, the advantages are usually the same as shown in [21] and [14]:

- **Independent**

  "Write once, run anywhere." This popular slogan created by Sun Mi-

crosystems portrays one of the most important features of virtual machines: Java code can be developed on any system, compiled into the standard Java bytecode, and is expected to execute on any operating system that is capable of running a JVM. Thereby, Java code is independent of hardware or the underlying system.

- **Secure**

  Java applications are running in an isolated environment; therefore making them more secure and protected against misuse and attacks like malware or computer viruses. During execution, the JVM accepts solely certain inputs that follow the predefined instruction set and denies malicious instructions. Furthermore, the Java language itself does not allow unsafe operations like pointer arithmetic, thereby making it impossible for programmers to access and change memory areas directly.

Various JVM implementations are available for almost every operating system since anybody can design and implement a JVM. However, the Java Virtual Machine specification [15], constituted by Oracle, has to be satisfied. One of the most important requirements is that every JVM implementation has to be capable of processing bytecode instructions and correctly performing the operations. Two of the most popular Java Virtual Machine implementations are Oracle's HotSpot and IBM's J9. They are differentiated by, for instance, garbage collection policies and the heap layout.

As the Java language improves and changes with every new edition, the Java Virtual Machine has to change as well. JVM implementations have to adapt in order to execute compiled Java code from new Java releases. Apart from this, other main goals for designing efficient and competitive Java Virtual Machines deal with optimizing throughput, decreasing execution time, and the reduction of the memory footprint of applications in execution.

Many JVM features are independent from the Java language and how to implement and utilize them is up to the developers. Garbage collection is one example. Every JVM implementation requires a dynamic memory management system that includes automated garbage collection (GC). What kind of GC policy is implemented, when to trigger a collection cycle, or how to organize the heap is up to the JVM developers. Specific settings, like the type of garbage collector, can often be chosen by users themselves. One example of a GC policy: IBM's latest JVM implementation features a GC algorithm, called *balanced* [20]. It splits the heap into up to 2048 regions and only collects a subset with each occurring collection cycle. The benefit is a single GC cycle takes less time since only a portion of the heap is collected. Furthermore, each collection takes roughly the same amount of time and they are relatively short. The downside is an overall longer execution time on average compared to other collectors. This is due to overall more GC cycles as wells as the overhead introduced by balanced.

The JVM itself is not capable of parsing Java code [15]. It solely has to process a certain binary code, called the class file format. It is obtained by compiling Java code with a Java compiler (for instance Javac, developed by Oracle). Each bytecode represents an instruction used directly by the JVM on a lower level compared to Java code. These instructions deal among others with object creation and mutation, but mainly on how to operate the stack.

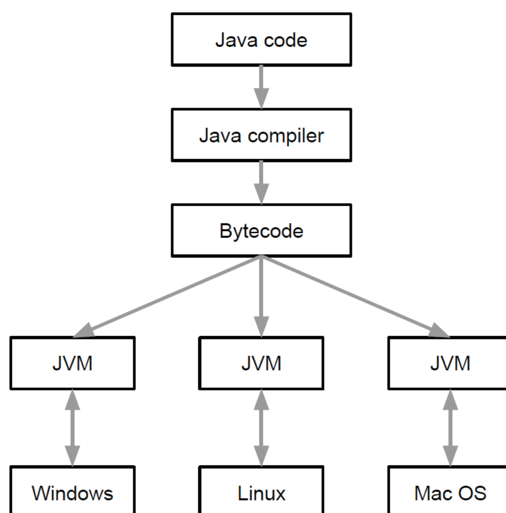

Figure 2.2: Executing Java Source Code

Figure 2.2 gives an overview of the JVM environment when executing a Java program: at first, Java-code needs to be compiled into the class file format. Depending on the underlying operating system, a different JVM implementation is required to run the program. The JVM is communicating

with the underlying hardware, which provides memory and CPU time.

2.2.1 JVM Internals

This section explains the internal, abstract architecture of every JVM as described by the Java Virtual Machine Specification, Java SE 8 Edition [15]. The run-time stack areas and heap as well as their inner components are presented in the following. Figure 2.3 gives an overview of the main components.

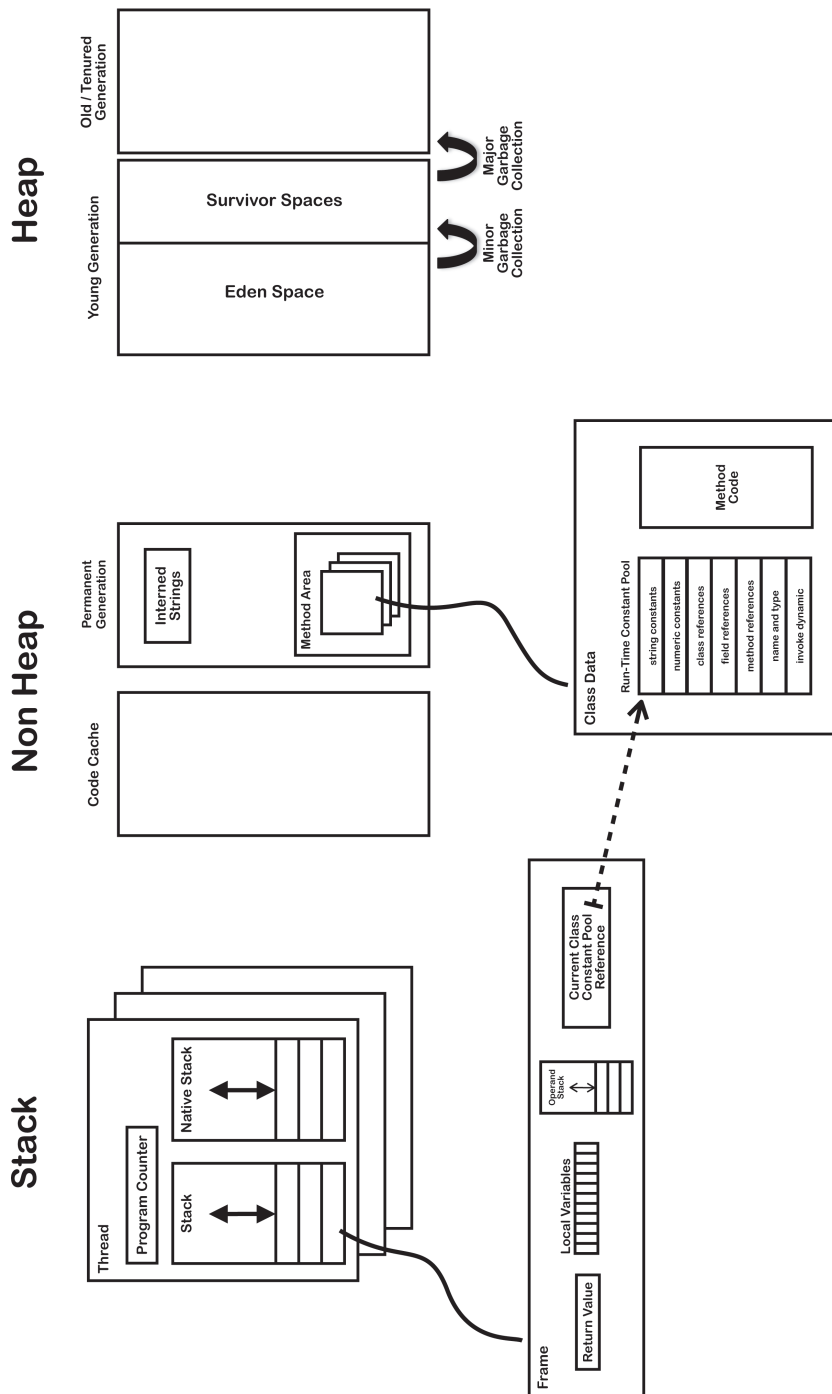

Figure 2.3: Java Virtual Machine Internals [8]

Run-time data areas are used by the JVM during execution. Some of these areas are created during startup, others belong to threads and therefore are created at the same time as the dependent thread.

Every JVM thread is associated with a stack created at the same time as the thread starts [8]. The stack is used as an execution area and is only operated by pushing and popping values in a last in, first out manner. A stack can be dynamic or fixed size. In case a thread requires a larger stack, a *StackOverflowError* is thrown and, depending on the JVM implementation, the stack might grow. On method invocation a stack frame is created. It is used by the current method and is destroyed at the moment of return. Each frame contains the following parts:

- **Return value**

  The return value is passed back to the method that invoked the current method that is being executed.

- **Local variable array**

  This array contains all local variables that are used by the method. Variables are primitive types as well as references to objects on the heap.

- **Operand stack**

  The operand stack is used to execute the current method by performing necessary operations. For example, the addition of two int variables a and b. In case a and b already reside on top of the stack, they first need

to be popped off the stack. Afterwards, a and b can be added together and the result is pushed on top of the stack for further operations. The result may also be stored inside the local variable array.

- **Class reference**

  The method holds a reference to its own class data as well as constant values that may be needed during execution.

The native stack shown in Figure 2.3 is not a mandatory component and typically only part of a JVM implementation if a C-linkage model for a Java Native Interface (JNI) Invocation API is used.

During startup, the heap is created as one of the first run-time data areas. It is shared among all Java Virtual Machine threads and used as memory area to allocate all class instances and arrays. Solely primitive data types, such as int, float, or double are put directly on the stack. Methods access objects on the heap by keeping references to them on the stack. All of these references together make up the so called root set. Depending on the garbage collection policy, the heap can be one contiguous space or divided into multiple areas. Figure 2.3 shows, for example, a heap layout used by a type of copying collector called generational. More on garbage collection is presented in Section 2.4

Objects used by the JVM itself are considered to be alive during most of the execution and are allocated in a special area called the Non-Heap mem-

ory. The method area for example stores class-structures as well as code for methods and constructors. It also contains the run-time constant pool with several kinds of constants such as literals or method and field references.
Java Virtual Machine specification Version 7 does not specify whether the method area has to be garbage collected or not.

2.3 Dynamic Memory Management

Automatic dynamic memory management is one of the biggest improvements for programming languages and is nowadays an essential component for software developers [13]. Providing a system that is able to allocate, deallocate, and to create references between objects automatically facilitates the work for programmers and, in general, promotes better code. Since first described in a paper in 1960 by McCarthy [17], continuous research, development, and improvements followed throughout the years. Thereby, two main goals are a decrease in execution time as well as the memory footprint of running programs.
Like the name suggests, dynamic memory management is responsible for managing the memory of running applications. The system executing the application (for example a JVM running a Java program) has to parse and process the program's code. The Java bytecode [15] instructs the JVM what operations to perform. The bytecode consists of several different instructions,

whereby many describe creation and mutation of objects. These are the core components of dynamic memory management [13] as it is used in the JVM:

- **Object Allocation**

  An object is allocated on the heap by a specific thread and is initially only accessible through the thread's root set. Depending on the deployed GC policy (Section 2.4) a specific algorithm decides where and how to store the object. Before doing so, enough space has to be available. If this is not the case, a garbage collection cycle or heap expansion is triggered.

- **Object References**

  Objects are able to point to other objects, thus creating connections between parents and their children. Previously created references can be updated (or mutated) to point to a new destination. All of these references combined make up the object graph of a running application. An object is only accessible (or alive) if it is reachable by a path from the root set.

- **Garbage Collection**

  In case not enough space is available to satisfy an allocation, a garbage collection is triggered. The task of the collector is to remove dead objects and only to keep those who are still alive. An object is considered to be dead as soon as it is no longer reachable by any other live object. Some GC policies might trigger their collection cycle before the whole

heap is occupied. This usually results in a longer collection time in total since more individual GC cycles are required. The benefit is a shorter collection time per cycle on average compared to whole-heap collectors. Applications that require a short response time can benefit vastly from such policies.

2.4 Garbage Collection (GC) Policies

Various garbage collection algorithms with different procedures and heap layouts are available in a JVM. Which GC policy is being used is up to the developers or users themselves. Benefits and drawbacks between different GC algorithms have to be considered. Variations exist in the collection time, the amount of available memory, elapsed time until dead objects are collected, fragmentation, and others. Depending on the kind of application and the environment, different aspects might be more or less important.

This section gives a brief overview of the four basic GC policies: mark-sweep, mark-compact, copying GC, and reference counting. Java Virtual Machine implementations use garbage collection policies that originated from these basic types. Modern GC implementations are more powerful and efficient compared to the standard types.

Most of the information presented in this section is gathered from the Garbage Collection Handbook [13].

2.4.1 Mark-sweep

One of the simplest and first implemented GC policies is mark-and-sweep garbage collection. The collection cycle is usually triggered when the heap is fully occupied, meaning no more memory is available to allocate new objects. The algorithm consists of two distinctive phases: the marking phase and the sweeping phase. The first phase walks the whole object graph, starting from the roots, and flags all reachable objects. The second phase simply frees every memory location on the heap that has not been marked thus making it reusable again. The major advantage of this algorithm is its simplicity: mark-sweep can be described only with a few lines of pseudo-code and the implementation is straightforward. Furthermore, the required collection-time is relatively short compared to other collectors. The main drawback of this algorithm is fragmentation of the heap. After collection, live objects remain at their original memory location. Thereby, each occurring collection cycle may cause new gaps between objects on the heap. This results inevitably in longer allocation times. The allocator has to find a free gap in order to allocate an object. Walking the whole heap and searching for free slots requires much time. Especially a first fit algorithm that starts every search at the beginning of the heap can take up a big portion of the execution time. The problem can be decreased through the use of more efficient algorithms. Next fit, for example, remembers the last allocation point on the heap and starts the next occurring search for a free slot at this location. This decreases the required time, but available memory chunks might never be used. An

optimal solution that solves fragmentation without moving objects does not exist.

2.4.2 Mark-compact

Mark-compact solves the main drawback introduced by mark-sweep. In order to avoid fragmentation, a compaction algorithm is included to the mark-sweep policy. This is accomplished by rearranging (copying) all live objects to one side of the heap. By doing so, the allocator only needs to check if enough memory is available. The location for new objects is always after the last allocation (this algorithm is called bump pointer allocation).

Similar to mark-sweep, this algorithm operates in different phases. The first phase is the marking phase. The following phases can differ between different mark-compact algorithms. However, a copying phase is always included. Since objects are moved, their parents might point to outdated addresses on the heap. So another required phase involves updating outgoing references. Several different mark-compact algorithms exist that offer variations on how to move objects on the heap.

One example is the often deployed Lisp 2 collector [13]. This relatively fast algorithm does little work in each of its phases at the cost of additional memory usage. Every object requires an extra slot that stores the address where it is being copied to (called forwarding pointer). Furthermore, the heap needs to be split into two regions: one contains all live objects, while the other one is used as a copy-to reserve. The pseudo-code for this policy with its three

phases is presented in Algorithm 1.

After marking all live objects (not part of the pseudo-code), a new copy-to location has to be found for every object. This is done by first iterating through the heap-region that contains the objects. In case an object is marked as alive, its forwarding pointer is updated to the current address of the copy-to reserve and the copy-to pointer moves forward by the size of the current object. The next phase is responsible to update all root and parent references. At first, all roots are updated according to the forwarding pointer. Afterwards, the same is done for all heap-objects. The last phase reallocates live objects by moving them to their forwarding address.
This algorithm solves the problem of fragmentation but increases the memory overhead by splitting the heap into two regions and by using a forwarding pointer. Furthermore, the garbage collector has to perform more operations compared to mark-sweep. The major advantage is relatively short allocation time throughout the whole execution.

2.4.3 Copying GC

As presented, mark-sweep works fast and simple, but promotes fragmentation. Mark-compact on the other hand eliminates fragmentation, but requires multiple passes and a comparably longer collection time. The next approach, copying GC, tries to solve both of these issues by copying live objects directly. Similar to mark-compact, the major drawback of this algorithm is a heap that

Algorithm 1 Lisp 2 Algorithm

```
function compact
    computeLocations(HeapStart, HeapEnd, HeapStart)
    updateReferences(HeapStart, HeapEnd)
    relocate(HeapStart, HeapEnd)

function computeLocations(start, end, toRegion)
    scan = start
    free = toRegion
    while scan < end do
        if isMarked(scan) then
            forwardingAddress(scan) = free
            free = free + size(scan)
        scan = scan + size(scan)

function updateReferences(start, end)
    for each fld in Roots do
        ref = *fld
        if ref not equal null then
            *fld = forwardingAddress(ref)
    scan = start
    while scan < end do
        if isMarked(scan) then
            for each fld in Pointers(scan) do
                if *fld not equal null then
                    *fld = forwardingAddress(*fld)
        scan = scan + size(scan)

function relocate(start, end)
    scan = start
    while scan < end do
        if isMarked(scan) then
            dest = forwardingAddress(scan)
            move(scan, dest)
            unsetMarked(dest)
        scan = scan + size(scan)
```

has to be split into two parts of the same size; thus cutting the amount of available memory in half.

A simple implementation of a copying GC algorithm is presented in the following:

At startup, the heap is split into two parts of equal size called fromspace and tospace. Figure 2.4 shows an example heap layout with five objects. New objects are allocated in tospace by simply incrementing a bump pointer. A GC cycle is triggered as soon as tospace does not offer any more free memory to satisfy the next allocation. As displayed, the bump pointer points to the far end of tospace, thus a GC is triggered with the next occurring allocation. The collection cycle performs the following tasks:

1. At first, the roles of fromspace and tospace are switched by swapping the labels of the regions as shown in Figure 2.4 (b).

2. The copying phase begins by traversing the whole object graph, starting from the root pointers either in a depth-first or breath-first manner.

3. Every live object is copied into tospace directly (objects O1, O2, and O5). A bump pointer is used to determine the current tospace-location.

4. A forwarding reference is added for every object in fromspace (marked as black pointers). Since old copies are no longer needed, any space can be used to store the forwarding pointer, thus no additional memory is wasted. Figure 2.4 (b), shows copied objects in tospace marked with a star. Parents in the new region are still pointing to children in

(a) Before Copying

(b) During Copying

(c) After Copying

Figure 2.4: Copying GC Algorithm

fromspace (for example Object 1). Objects O3 and O4 have not been copied since they are not reachable by any live parent or from the roots.

5. After copying all live objects into tospace, every reference has to be updated. This is done by once again traversing the object graph starting from the roots. However, this time every reference from a parent to a child is updated according to the forwarding pointer address. The same is done for root pointers as well.

6. Figure 2.4 (c) shows the final outcome: all live objects are copied into tospace and references are updated correctly. The collection phase is completed and new objects can be allocated at the current bump pointer location. Fromspace will be used again for the next collection cycle. Many collectors choose to zero out old objects in fromspace for safety reasons.

2.4.4 Reference Counting (RC)

Reference counting is the only GC policy presented in this thesis that does not follow a tracing algorithm approach. A tracing collector finds live objects by traversing the object graph. RC operates directly on reference changes. Tracing collectors might keep dead objects for a long period of time before freeing them. Reference counting collects dead objects as soon as they are no longer reachable. In order to do so, objects keep an additional slot that stores the number of incoming references. The reference count is updated

directly on mutation.

Every write operation activates the write barrier. A simple pseudo-code for a RC implementation is shown in Algorithm 2. Function *process* represents the write barrier. An object, that lost an incoming reference is passed as *oldChild*. A newly created pointer to an object leads to an increase of the reference count for *child*. Depending on the write barrier-implementation, both *oldChild* and *child* can be null.

Algorithm 2 Reference counting

```
function PROCESS(oldChild, child)
    if child then INCREASEREFERENCECOUNT(child)
    if oldChild then DELETEREFERENCE(oldChild)

function DELETEREFERENCE(object)
    if object then DECRESEREFERENCECOUNT(object)
        if GETREFERENCECOUNT(object) == 0 then
            children = GETCHILDREN(object)
            while children do
                child = GETNEXTCHILD(children)
                DELETEREFERENCE(child)
            FREEOBJECT(obj)
```

DeleteReference is called inside the write barrier to decrease the reference count. In case the count drops to zero, the object is deleted. Before doing so, all children have to be traversed and their reference count is decreased as well. This might trigger a chain reaction, leading to the deletion of multiple objects.

Advantages of this algorithm are relatively short collection times and the removal of unreachable objects at the moment of death. The downside is

mainly an increased memory usage due to the additional reference count field. Furthermore, this policy promotes fragmentation in case no additional compaction algorithm is included.

Another major problem with reference counting is object-cycles that cannot be captured. Objects that are pointing to each other without any incoming reference from other reachable objects are considered to be dead. Unfortunately, their reference count never drops to zero and so they are not deleted. Advanced cycle-detection algorithms, as for instance the recycler [7], are able to detect cycles and remove dead objects. The cost is a complex algorithm with more overhand and an even longer execution time.

2.5 The GarCoSim Framework

Various Java Virtual Machine implementations have been available (for example Oracle's HotSpot, initially released in 1999 [4]). These systems are complex and consist of much code (possibly also written in diverse and outdated languages). Programming, experimenting, and testing new features can be a complicated and time consuming task. Furthermore, testing new code for correctness and efficiency, comparing different policies and algorithms, or running benchmarks require a lot of work and time. The outcome might not be assessable during development and in case improvements are not sufficient much effort may result in no improvements. One way to tackle this problem is by using a simulator that reproduces desired JVM behavior.

By working with a JVM simulator, implementing new features and testing them, as well as evaluating, can be relatively easy and requires little effort. If results are promising, changes can be ported to a JVM implementation. This requires the simulator to operate at a realistic level. Only then changes and features implemented in the simulator are expected to lead to the same results in the JVM. For example: a new garbage collection policy implemented in the simulator leads to a significant reduction in execution time compared to other collectors. Similar results are expected when adding the new collection policy to the JVM.

JVM simulator implementations are able to reconstruct heap activities and perform garbage collection. Dieckmann and Hölzle [9] implemented a trace file generator alongside a JVM simulator. However, instead of developing a JVM testing tool for new algorithms, their work focused on the analysis of the memory usage of the SPECjvm98 benchmark suite [3]. First, trace files (simulator input) are generated with an instrumented Sun JDK1.1.5 VM while executing a benchmark. In a second phase, the simulator (programmed in Java) parses and executes the trace file. Allocation, pointer assignments, and garbage collection is simulated while statistics are computed and gathered. Experimental results presented in Dieckmann's and Hölzle's work covers among others heap size and object lifetimes, comparison between instance objects and arrays as well as size and alignment of objects.

The GarCoSim Framework [18] is used as part of this project. It is capable of parsing memory management instructions and to simulate heap operations. This framework contains two separate parts: the simulator itself as well as a trace file generator. The generator is able to produce synthetic trace files that can be used as input for the simulator. Trace files are comparable to the JVM class files: a set of instructions with information on what operations to perform. Synthetic trace files consist of randomly generated instructions. Some parameters as ratio between allocate, read, and reference operations can be specified. The trace file generator itself is not relevant to this project and thus not further discussed in this thesis. However, more details on the trace file structure, how to generate them from Java applications, and information about the simulator is presented in the following sections. The GarCoSim framework is available on GitHub [2].

2.5.1 Trace Files

The GarCoSim simulator requires instructions as input that determine what operations to perform on the heap. A set of these are found in *trace files* [16]. Table 2.1 gives an overview of all valid instructions. Attributes of trace file operations are presented in the following:

- Ti: i is a thread id (i >= 0)
- Oj: j is an object id (j >= 1)
- Cj: j is a class id (j >= 1)
- Sk: k is the size of a object in bytes (k >0)

MM Operations	Notation	Usage
Allocation	a	Ti Oj Sk Nl Cj
Add a reference of an object to the rootset a thread	+	Ti Oj
Store/write a primitive field into an object	s	Ti Oj Ix(/Fm) Sn Vo
Store/write a static primitive field into a class object	s	Ti Cj Ix(/Fm) Sn Vo
Store/write an object reference field into an object	w	Ti Pj #k Ol Fm Sn Vo
Store/write a static object reference field into a class object	c	Ti Cj Ix(/Fm) Ol Sn Vo
Read a primitive or a reference field from an object	r	Ti Oj Ix(/Fm) Sn Vo
Read a static primitive or reference filed from a class object	r	Ti Cj Ix(/Fm) Sn Vo
Delete an object reference from the rootset of a thread	-	Ti Oj

Table 2.1: The trace file instructions

- Nl: l is the no. of reference slots in an object (l >= 0)
- Ix(/Fm): x(/m) is the index(/offset) of a filed in an object (x(/m) >= 0)
- Sn: n is the size of a field in an object (n >= 8)
- Vo: o is either 0 or 1 represents the field type either non-volatile or volatile
- Pj: j is an object (Parent) id (j >= 1)
- Ol: l is an object (Child) id (l >= 1)
- #k: k is the slot number of a reference field in an object

The simulator is started with a trace file as input and parses the file line by line. Objects are allocated, references are created and may be removed later on. As with the JVM, the simulator is capable of performing garbage collection in case no more space is available—or sooner, depending on the specified GC policy.

Out of the nine available operations, four are essential to simulate a functioning heap:

- **Allocation (a)**

  Objects are allocated from one specific thread. Both object and thread

are identified by their ID. Further attributes are the object size given in bytes and the class ID. Names of all classes and threads are stored in additional files together with their attributes. Every object has a certain number of reference slots available—the number is specified after the index 'N'. For instance: an object allocated with five slots can store up to five outgoing references to children simultaneously.

- **Reference operation (w)**

  Reference operations create connections between parents and children. Only if the specified slot number 'k' is valid (the parent has enough outgoing slots available) and the child can be found on the heap, a pointer is created. A reference operation can also point to object zero. This leads to the deletion of any previously stored reference in the given slot.

- **Root set addition (+)**

  A root set addition is created for the specified object.

- **Root set deletion (-)**

  A root reference is removed for the specified object.

Trace files are generated by running Java applications with an instrumented JVM. Java programs used in this approach are typically the DaCapo [1] and Standard Performance Evaluation Corporation (SPEC) [3] benchmarks. Figure 2.5 gives an overview of subsystems and files that are part of the trace file generation process.

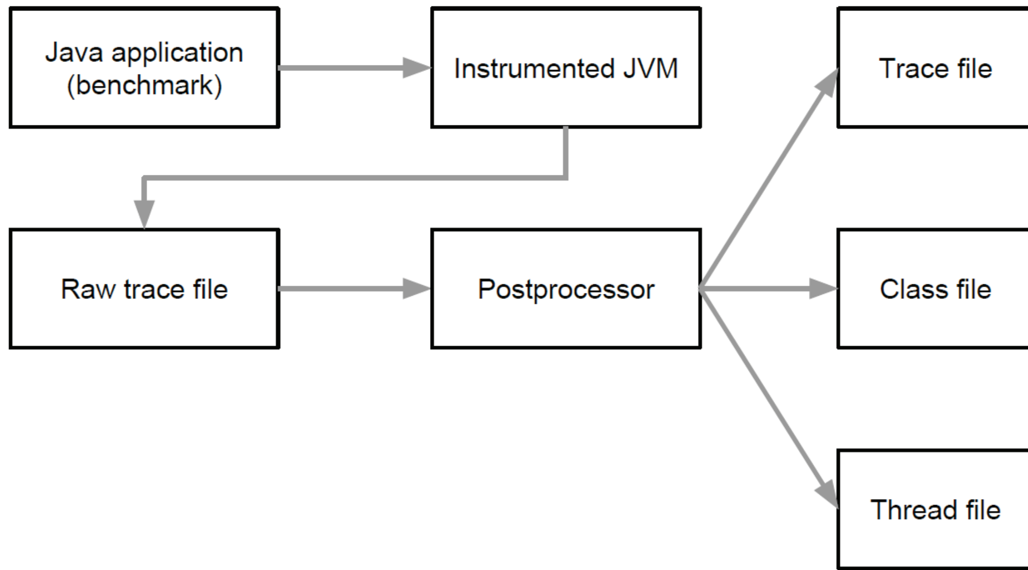

Figure 2.5: Generating trace files

The instrumented JVM used in this approach originates from IBM's J9 Java Virtual Machine in version 9. It has been modified to print operations during execution to an output file, the raw trace file. Modifications were inserted at every location of the JVM code that deals with object allocation or mutation. Additional information as object-sizes, class names, or thread names are captured as well.

Object allocations, read and store operations as well as references between objects are printed directly into raw trace files as they are executed by the JVM. This is a short example with two allocated objects and a reference operation:

a ‘(main)’ O1 S112 N9 int

a ‘(main)’ O2 S32 N1 int

w ‘(main)’ P2 #0 O1

Unfortunately, the instrumented JVM is not able to capture root set additions or deletions directly at execution time. It is solely possible to capture the current state of the roots for all threads at predefined moments during execution. This is an example for a root dump:

root 2:

‘(main)’ O1

‘(main)’ O3

‘(main)’ O4

Objects O1, O3, and O4 are all accessible directly from the roots by thread ‘main’ at the moment this dump was generated.

2.5.2 Post-Processing

The simulator is not capable of parsing raw trace files. The reason is root dumps: the raw file contains information about the current state of the roots. However, the simulator requires exact points for a root set addition or deletion to be able to execute the operation. In order to solve this problem, post processing was added to the generation process. It is responsible for

transforming root dumps into root set additions and deletions.

Two post-processors are available with two different algorithms that are able to generate the missing root set additions and deletions. The following task is performed by both post-processors in similar manner:

As shown above with example lines of a raw trace file, threads and classes are specified by their names (for example 'main' or 'int'). In order to minimize the size of trace files (some are larger than 100GB), thread and class names are replaced by identification numbers. The corresponding names are put into an additional class and thread file. This makes trace files faster to parse by the simulator as well.

Objects are only accessible by incoming references. To be able to access newly allocated objects, both post-processors add one additional root set addition for every object immediately after allocation. This initial root reference is removed eventually. The moment of removal is the main difference between post processor 1 and 2. They are presented in the following subsections.

2.5.2.1 Post Processor Version 1

The first post processor solves the root dump conversion problem by keeping root pointers to objects as long as they are accessed in the trace file. The processing-procedure is explained in the following with a small example. Thread and class names are already converted into IDs and initial root set additions were added to every allocated object:

a T1 O1 S112 N9 C1

+ T1 O1

a T1 O2 S32 N1 C1

+ T1 O2

w T1 P2 #0 O1

The next root dump consists only of object O1:

root 2:

'main' O1

Post processor 1 keeps track of objects that are accessible by root pointers. Up to the root dump, both objects one and two are accessible by their initial root reference. O1 is part of the root dump, O2 is not. No changes are necessary for O1 since it is already accessible by the roots and simply remains in the root set. Object two on the other hand is not part of the root dump and thus it is no longer accessible through the roots. As result, a root set deletion is printed into the trace file:

a T1 O1 S112 N9 C1

+ T1 O1

a T1 O2 S32 N1 C1

+ T1 O2

w T1 P2 #0 O1

- *T1 O2*

w T1 P2 #0 O0

The next occurring operation after the root dump is the creation of a reference from O2 to O0. Creating a reference to the nonexistent object zero results in the removal of any previously stored reference in the appropriate slot. This already shows the problem of root dumps: object two is no longer accessible by any incoming reference after its root set deletion. At this point of time it is garbage and could be deleted by the collector (for instance, a reference counting policy would delete object two immediately). Unfortunately, O2 is yet required for another write operation.

In order to keep object two alive as long as it is accessed, the newly created trace file is processed two additional times.

The first pass locates the last access to every object and stores this information. In the previous example, this is the last write operation for O2. The second pass moves the root set deletion after the last access. Object one is removed from the roots as well:

a T1 O1 S112 N9 C1

+ T1 O1

a T1 O2 S32 N1 C1

+ T1 O2

w T1 P2 #0 O1

- T1 O1

w T1 P2 #1 O0

- *T1 O2*

This solves the problem of losing objects prematurely since every object is accessible by its root pointer as long as it is used in further operations. However, this leads to an unrealistic and inflated root set compared to the JVM. The garbage collector is able to access all of these objects with one step only by traversing the root pointers.

Figure 2.6 shows the number of objects per traversal depth level from a trace file generated by the first post processor. Level 1 objects are accessed by one step only, these are the root pointers. Level 2 objects are accessed by parents that are accessed themselves by root pointers. The levels in this example go up to 18. The simulator that generated these results used the mark-sweep garbage collector with a breadth-first traversal algorithm. In order to count depth levels during GC, small modifications were applied to the simulator. Traversal depth levels shown in this example originate from a small Java program. Actual results on benchmarks are presented in Chapter 6.
The majority of objects are actually not directly accessible by root pointers. The reason behind these results is how and when post processor 1 prints the last root set deletion. Every object that is used in read or write operations is accessible by a root pointer. However, objects can be alive but never be used again. For example, a live parent has a reference to a child object. The child is never used in the trace file but it is still alive through its parent's

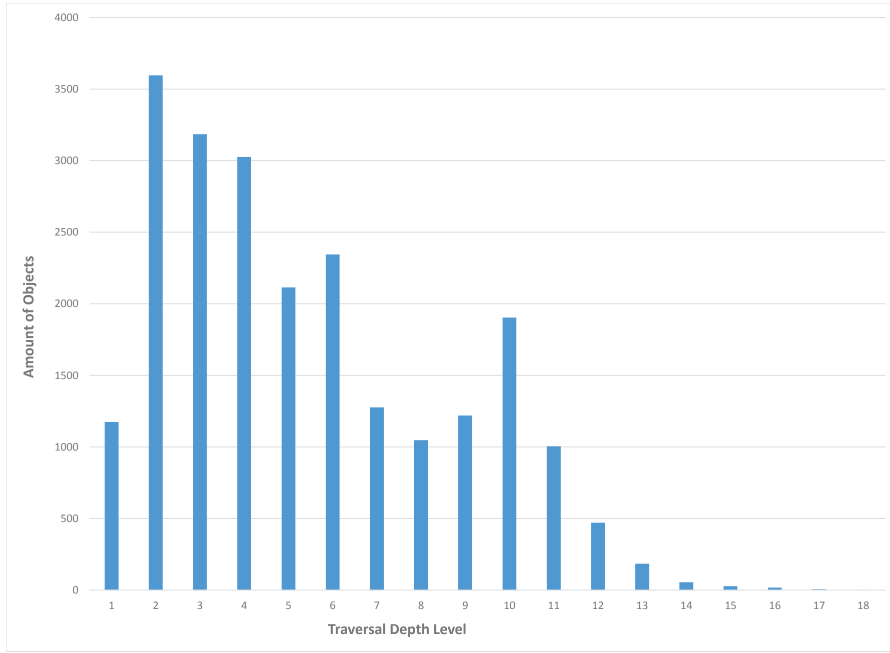

Figure 2.6: Traversal depth levels

pointer. In other words, all objects that show a higher traversal depth level than one are not involved in future operations in the trace file.

2.5.2.2 Post Processor Version 2

The second approach of post-processing attempts to solve the problem introduced by post-processor 1. The solution is to compare every root dump to the previous one. With this comparison, it is possible to determine if an object was added, deleted, or already part of the current root set:

- An object that is part of the current root dump but not part of the previous one is added to the roots. The root set is considered to be empty at the first line in the trace file.
- An object that is both part of the current and the previous root dump was already added to the roots. No operation is required.
- An object that was part of the previous root dump but not part of the current root dump is removed from the roots.

This approach does not require additional processing phases. Compared to the first post-processor, the new procedure is more realistic and accurate. Root references are not kept to every live object during the whole execution. Mistakes can only occur between two root dumps: it is not possible to determine if an addition or deletion happened right after a previous dump or momentarily before the next root dump—an inaccuracy that has negligible consequence for the simulator.

Unfortunately, the simulator is not able to operate on the new trace files. Executing the simulator with trace files generated by post-processor 2 results sooner or later in accesses to objects that are already deleted. An object is allocated and accessed correctly until it becomes garbage (no more incoming references). The object is deleted by the garbage collector. Later on, during execution, this object is accessed again in the trace file. Obviously, the simulator is not able to find the deleted object and the simulation results into termination with an error message. The exact reason and origination behind

this problem is discussed in Chapter 3.

2.5.3 Simulator Implementation

The GarCoSim simulator is capable of simulating a dynamic memory management system. It does so by parsing trace files and performing operations on the heap. It is developed in C++ and aims to be highly modular, thus making it easy to add new collectors, allocators, or write barriers. Figure 2.7 shows the basic class diagram with notable functions. It is far from being complete; some classes and many functions were deliberately excluded to provide a clear illustration.

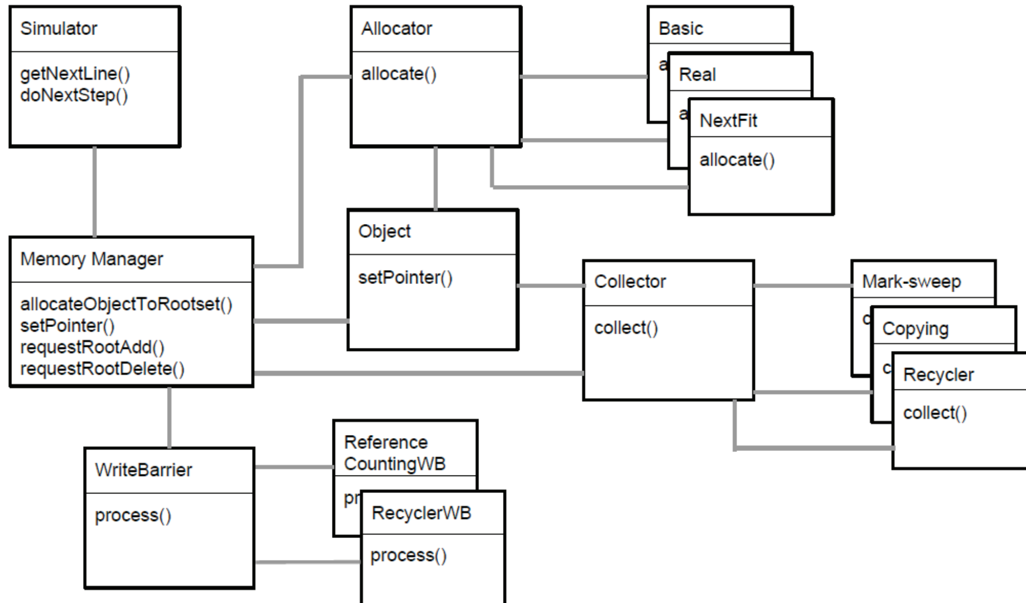

Figure 2.7: Simulator Class Diagram

The simulator is started via command line arguments with several different options available. The desired collector (mark-sweep, copying, recycler), allocator (basic, real, next fit) as well as the write barrier (reference counting, recycler, none) can be specified. Further required arguments are the trace file location and the initial heap size. Depending on the collection policy, the heap is able to grow during execution. Invoking the simulator with a heap size of 1000 bytes, next fit as the allocator and mark-sweep as collector without a write barrier may look like this:

./traceFileSim exampleFolder/exampleTraceFile.trace -h 1000 -a nextFit -c markSweep

After initializing all required components, the simulator instance starts to parse the trace file by reading line after line. Depending on the current operation, the appropriate memory manager function is called. This is the main component of the project and is connected to the allocator, collector as well as the write barrier.

In order to allocate an object, the specified allocator is called with the required size as an argument. If enough memory is available to satisfy the allocation, the object is allocated. The memory manager then creates the object with all specified arguments. In case no memory is available, the allocator returns null and a collection cycle is triggered. The collect function of the specified collector is called by the memory manager and dead objects are removed in order to continue with the allocation. If not enough memory is available to satisfy the second allocation for the same object, the whole

simulation is aborted with an out-of-memory error message.

Reference changes are directly created by the memory manager. In case a write barrier is selected, it is called during reference operations.

After execution, a log file is created that contains statistics including every collection cycle with info about the number of freed and live objects, GC duration, or occupied memory in bytes.

Chapter 3

Searching for Zombies

The objective of this project is to generate correct trace files and to avoid flaws of post processors 1 and 2. In order to so, at first trace files in previous versions are examined. The source of the error is actually located in raw trace files as will be revealed later in this chapter. In order to solve errors of trace files, at first the root of the problem needs to be revealed and examined. This includes an investigation on zombie objects to understand why they are part of trace files and to determine their origin.

What is the real source of error of processed trace files and, more important, what part of the generation process is responsible for the flaws? It is impossible to produce accurate trace files as long as this question remains unanswered. In order to find a solution, it is necessary to take a second look at post processing as well as the raw trace files.

It is easy to see that post processor 1 is not able to generate reliable inputs for the simulator. As presented in Section 2.5.2.1, every allocated object is accessible through the root set as long as it is used in future operations in the trace file. The last root deletion is always printed after the last access to an object, thus resulting in an overloaded root set. No matter how correct the input may be, flawed trace files are always the result. Therefore, post processor 1 will no longer be considered as a viable option or as basis for this project.

The error source of post processor 2 cannot be explained this easily. All that is known at this point is trace files contain objects that are accessed after they become garbage—the so-called zombie objects. In theory, the second post processor should be able to generate accurate trace files. Testing showed, running it with artificially created input always results in flawless output. The problems must reside in raw trace files. Therefore, post processor 2 is not able to produce a correct outcome with the currently available raw trace files. Another algorithm for post processing might be able to generate correct trace files. However, it still has to make some assumptions in order to work around the flaws of raw trace files.

3.1 Raw Trace Files Examination

The zombie objects have to be detected in order to be able to investigate the problem in more depth. Only then, can zombies be traced back to their origin—the instrumented JVM. By now, it is known that zombies reside in raw trace files and are simply transferred into processed trace files. A slightly modified GarCoSim simulator is able to find almost all zombie objects in trace files.

To achieve this, the reference counting policy (see Section 2.4.4) is used. As soon as an object's reference count drops to zero it becomes garbage and is deleted. Every object that is accessed but already deleted by the reference counting policy must be a zombie. Read, write, store, and root operations are monitored and in case access to a dead object occurs, its ID is stored. At the end of the execution, all of the saved IDs are printed into a log file—the zombie objects are detected.

This approach cannot guarantee to detect every single zombie object. A pure reference counting algorithm is not able to detect cycles. For example, three objects are pointing to each other but are not reachable from the roots or any other object. They are unreachable, and thus garbage. However, their reference count never drops to zero and those objects will not be deleted. It could be possible to extend reference counting with a cycle detection algorithm, as for instance with the recycler. This policy is able to find all unreachable objects, including those in cycles, with a cyclic reference counting algorithm.

This was not done for the following two reasons:

- Trace files generated by the instrumented JVM have huge file sizes. For example, lusearch from the DaCapo benchmark suite results into a trace file with a size of over 56GB with more than three billion lines of operations. Running these files with the simulator is a time–consuming task that can last up to days. Extending the reference counting policy with cycle detection that is able to detect dead objects immediately at the moment of death increases the already long execution time exponentially.

- It is simply not necessary to be aware of every single zombie object. The goal is to reveal zombies and their origin in the instrumented JVM. The information on the actual amount of zombies is merely an additional, although interesting statistic.

The modified simulator was used to detect zombie objects in trace files that were generated by the instrumented JVM. The Java programs used to generate the raw trace files are benchmarks from the DaCapo-9.12-bach suite. Table 3.1 shows the amount of zombies as a percentage of allocated objects.

Zombies range between 0.06% up to 8.67% compared to the total amount of allocated objects. Only if trace files do not contain any zombie objects, the simulator can guarantee to execute without faults. One access to a deleted object is enough to abort execution.

Benchmark	Allocated objects	Zombie objects	Zombie percentage
avrora	1,972,773	4,585	0.23%
batik	1,181,050	22,888	1.94%
fop	3,039,465	133,353	4.39%
luindex	407,841	35,353	8.67%
lusearch	13,326,074	7,486	0.06%
pmd	7,642,402	9,976	0.13%
xalan	6,661,173	160,102	2.40%

Table 3.1: Zombie objects of the DaCapo-9.12-bach benchmarks

Zombie objects are successfully revealed in various trace files. The next step is to investigate their origin in order to detect their cause. To reveal how and why unreachable objects are accessed, a sample of zombies was examined directly in raw trace files. One example is presented in the following. These are the operations for object 142 from the raw trace file generated by the fop benchmark:

a '(main)' O142 S32 N1 java/lang/String

a '(main)' O143 S24 N0 char

w '(main)' P142 #0 O143

s '(main)' P142 F20

r '(main)' O142 F24

root 2:

'(main)' O140

'(main)' O142

r '(main)' O142 F0

root 2:

'(main)' O140

r '(main)' O140 F0

r '(main)' O142 F20

Several operations occur between allocation up to the moment object 142 becomes a zombie at the last shown line. Some unnecessary operations have been removed that do not deal with the relevant object. This applies also for repeating read operations that do not add any useful information. The total number of lines between allocation and zombie creation is 242 in the raw trace file.

Object 142 is allocated as a String. Every String object has an associated char array, which is allocated as well. They are connected by one reference. According to the first root dump, O142 is directly accessible from the roots. This is the only reference that keeps object 142 alive. The following read operation can be executed by the simulator. Unfortunately, O142 is not part of the next root dump. At this moment, a root delete is printed into the trace file and the garbage collector (for instance reference counting) might delete it immediately. The next read operation tries to access the already

deleted object without success.

All of the examined zombie objects show similar characteristics that can be grouped and outlined into these three states:

1. A zombie object is allocated. It can be part of the next occurring root dump but does not have to be.
2. At some point the object is not reachable anymore. It has no live parent pointing at it and is not part of a root dump. The garbage collector might delete it.
3. Write, read, or root operations try to access the already dead object.

3.2 Missing Root Operations

The reason for zombie objects in processed trace files has now been presented and examined. However, the question remains: why are zombie objects part of raw trace files and what part of the instrumented JVM is responsible for their creation? Zombie objects appear to be dead at some point in the trace file but are accessed later again. This indicates that the zombie object was never dead up to the moment of the access operation. In order for an object to be alive it has to be accessible by an incoming reference.

Three explanations are possible that describe reasons for missing references:

1. Not all write operations are captured, and thus objects lack incoming pointers.

2. Missing root additions: the periodically printed root dumps are not complete, root pointers are missing.

3. JVM internal references keep objects alive but are not captured in trace files. This explanation includes all possible internal connections, such as references and data structures.

Zombie objects occur in trace files because of missing references that would keep them alive. The three presented explanations about the origination of zombies cover all the possibilities of missing references. Therefore, out of these three explanations, at least one must be the cause of zombie objects. In the following, missing root additions as well as JVM internal references are proven to be the cause of zombies. This is done by eliminating the other explanation of missing write operations. Furthermore, the reasoning behind missing root pointers is given.

In order to verify that all write references are captured as well as no operations are printed falsely, the already implemented instrumented Java Virtual Machine was examined. Write, read, and store operations are captured through J9's object access barrier. This class consists of roughly 100 functions that all create references or return values for all types of objects and date types supported by the JVM. With an enabled access barrier, the JVM performs all object accesses inside the barrier. All that needs to be done is

to ensure that every relevant operation in the access barrier is monitored. Every function was examined and tested thoroughly while stepping though the JVM in execution using the GNU Debugger. Fortunately, most of these functions are rather simple and operations are performed with a couple of lines only. During examination, no absence of read or write operations was discovered. Captured references between objects were observed; especially accesses to known zombies were investigated to guarantee their correctness. Nothing unusual or incorrect was discovered during these observations of references to zombie objects.

This examination excludes missing write operations as reasons behind zombies. The second explanation for zombie objects is missing root additions. Root dumps are printed periodically into raw trace files. The hypothesis is root pointers are not captured completely during the creation of a root dump. Triggering a root dump results in the iteration of every stack's root set in order to find all of its root pointers. IBM's J9 has a built-in stack iterator that traverses all slots for every thread. It was used in this approach to print the root dumps. The iterator is only reliable if it guarantees to find all references from stacks to heap. Investigation as well as literature suggests it is not. The reason for missing root pointers is references from native C/C++ code. These operations are hidden from root dumps and are not captured in the instrumented JVM. This leads to the absence of some root pointers, and thus the creation of zombies.

This observation corresponds with Dieckmann's and Hölzle's work [9] presented in Section 2.5. They found several objects in trace files that appear to be dead (unreachable) but are still used later on in operations. This is due to pointer stores from C code that have been ignored, since the JDK 1.1.5 VM used in their approach contains too many places that directly manipulate Java objects. In order to remove zombies, their solution is to resurrect zombie objects in the simulator in case they are accessed after deletion.

The third potential reason for zombies are JVM internal operations on specific objects. Several hash-tables and similar data structures are used to store references to immutable or immortal objects that are never deleted. Classes are one example. During class loading, before the first object of a certain class is initialized, the class itself has to be created. These class objects are stored in the constant pool, a part on the heap that is usually not garbage collected. Class objects are accessible by JVM internal data structures through static references. These connections are not captured in the instrumented JVM. Another example is String objects. Strings are immutable objects that can be garbage collected but are always accessible through an internal String table in case their content (the character array) is reused.
This concludes the investigation of the origin of zombie objects.

Chapter 4

Design

The goal of this chapter is to sketch an outline that describes the necessary steps that were performed as part of the implementation in order to create accurate trace files. As presented in Chapter 3, reasons behind the origination of zombie objects are missing root pointers, references from native C-code as well as internal JVM data structures. This chapter describes a new approach that is able to generate realistic trace files that do not contain any zombie objects.

It is organized as follows:

1. The requirements section 4.1 lists all challenges that need to be solved by the new instrumented JVM on the basis of findings in Chapter 3.

2. Section 4.2 presents the advanced instrumented JVM with the detection of root set operations on a bytecode level. Further design decisions that minimized the amount of zombies are shown as well.

3. The zombie removal processor removes all remaining zombie objects that might reside in trace files. It is presented in Section 4.3. This processor is the final part of the system and its output is a zombie free trace file that is usable by the simulator.

4.1 Requirements

The main drawback of the previous instrumented JVM with post processing is the extraction of root references out of root dumps. On this basis, a correct representation is difficult, if not impossible. Information of reference changes between two root dumps is lost and cannot be recovered. Therefore, the approach presented in this thesis captures root additions and deletions on occurrence, thus eliminating the post processing step. Complete trace files are created directly by executing the instrumented JVM. However, post processing is still partly required to transform class- and thread names into identification numbers and to perform similar tasks.

The new instrumented JVM has to accomplish three main tasks:

- All root additions and deletions have to be captured on occurrence.
- In order to keep internal JVM objects alive, the initialization of class objects during class loading and similar operations have to be monitored.
- References as well as root set additions and deletions originating from

native C code have to be captured.

4.2 Instrumented JVM Version Bytecode

In this section, the design for the new instrumented JVM (named version bytecode) is presented. The main component and core for this design is the extraction of root set additions and deletions from bytecode instructions. The new approach revealed several issues that complicated the generation of zombie free trace files. They are discussed later in this chapter as well.

4.2.1 Monitoring Bytecode Instructions

In order to capture root set additions and deletions on occurrence, JVM's bytecode instructions [10] are monitored. The Java Virtual Machine Specification (Java SE 8 Edition) lists 149 instructions in total. An instruction consists of a one-byte opcode specifying the operation to be performed. It may have zero or more operands that supply arguments or data that are used by the operation. The inner loop of a JVM is the bytecode interpreter. It is responsible to parse the class file, where all of the bytecode instructions of a compiled Java program are found. One after the other, every single instruction is executed by the JVM through the bytecode interpreter. It performs appropriate operations, calls JVM internal functions, and manipulates the stack. Required values during execution are stored by pushing and loaded by popping off stack slots (more details on the stack can be found in Sec-

tion 2.2.1). During execution, the stack is filled with relevant data and grows or shrinks as needed. Depending on the current bytecode instruction, values are pushed or popped off the stack.

Pushing a reference to a heap-object on the stack creates a root set addition. Popping that pointer later on results in a root set deletion. Out of the total 149 bytecode instructions, some manipulate the root set and some do not. Here are three examples:

- **iadd: addition of two int values.**

  Two int values are popped of the top of the stack. They are added together and the result is pushed onto the stack. The root set is not manipulated.

- **dup: duplicate the top stack value.**

  The top value on the stack is duplicated and pushed onto the stack. An addition to the root set only occurs, if the pushed value is a pointer to an object on the heap.

- **laload: load long value from array.**

  The reference to an array as well as the array-index are already located on top of the stack. First, both reference and index are popped off. Afterwards, the long value at the index of the array is retrieved and pushed onto the stack. The popped reference to the array results into a root set deletion.

IBM's J9 Java Virtual Machine includes a class called *BytecodeInterpreter*. Fortunately, all bytecode instructions are executed through this single class. In addition to the 149 instructions defined by the JVM Specification, the *BytecodeInterpreter* class contains several helper functions. They are called directly from bytecode instructions and perform additional tasks.

Bytecode instructions either push or pop values on the stack. Some do both, while others do neither. All instructions were examined and in case references to heap objects are pushed or popped, the relevant operation has to be printed into the trace file. Table 4.1 shows the numbers of push and pop operations that manipulate the root set, including bytecode instructions as well as helper functions.

Push	Pop	Push & Pop	Neither
41	19	5	269

Table 4.1: Numbers of examined bytecode instructions and helper functions.

Out of the total 334 functions found in the *BytecodeInterpreter* class, 65 are relevant to this project. All of these functions were modified in order to print root set additions or deletions into the trace file. Several functions, as for instance the previously presented pop instruction, can be executed with any data type. It might be a number, character, reference to a function or a root pointer. In order to capture only root references, the value is compared

against the base and top heap address. Solely if the value falls in between these two addresses, it is classified as a reference to the heap and the root set operation is printed. Regular numbers, as for instance integers, cannot fall in this range. This is due to the required size of integers and pointers: an integer requires a size of 4 bytes, while a reference requires 8 bytes in IBM's J9 JVM. It is not possible for a 4 byte number to fall in between the heap base and top.

4.2.2 Root References outside the Bytecode Interpreter

The *BytecodeInterpreter* class is not the only location in IBM's J9 Java Virtual Machine that performs root set operations. During work on this project, several classes, functions, or macros were discovered that add and remove root set references. Especially monitoring three macros that either push or pop one single object helped to significantly decrease the amount of zombies. These macros are used heavily in many functions throughout the J9 JVM.
Further root set additions and deletions occur in JNI (Java Native Interface) helper functions and classes. The JNI enables Java programs to execute and call native applications written in languages such as C or C++. These classes and functions were located and monitored.
Finally, code that is mainly used during the JVM initialization phase manipulates the stack. Particularly the initial class loading that has to be performed before a class is available to be used results into several root set additions.

4.2.3 Locking

Bytecode instructions are, if solely considering the context of garbage collection (GC) as interruptions, atomic operations. All operations triggered by the initial instruction have to be completed before a safe collection of dead objects can occur. In case a GC cycle is performed during an instruction, provisions have to guarantee not to delete objects prematurely.

One example to illustrate this problem is the allocation of a String object. The J9 JVM allocates alongside every String object a corresponding character array. For instance the sequence of letters *"zombie"* stored as String object in a Java program is externally accessed by the String object. Internally, *"zombie"* is stored in a character array. During initialization of String objects, several operations are performed between String and character array as well as to other internal objects. The String is only accessible (alive) after it is returned to the bytecode instruction and added to the root set. A garbage collection during the initialization might delete the String object prematurely and leaves the current execution in an inconsistent state. Several bytecode instructions trigger the allocation of objects, whereby an object is only then accessible after it is returned and added to the roots. IBM's J9 prevents this scenario from happening. The simulator does not have mechanisms to prevent the premature deletion of objects while parsing a trace file. In order to solve this problem, locking was introduced in the instrumented JVM as well as in the simulator. Every bytecode instruction that triggers the allocation of an object and adds this object to the root set after a return

was extended by locking. A locking operation is printed into the trace file before a relevant function call. The lock is released by an unlock operation after the print of the root set addition.

The simulator was extended to be able to parse locking operations. In case a GC is triggered by an allocation during a locked period, operations that already have been performed during the locked period are reverted. The current point of execution is set back in the trace file before the locked period and the garbage collection can be performed safely. Statistics on the numbers of locked and unlocked lines in trace files is presented in Chapter 6.

4.2.4 Zeroing Out Stack Slots

IBM's J9 JVM pops off stack slots by moving the stack pointer (sp) one slot ahead. The popped off value remains in the stack slot, only the stack pointer moves. The value in this slot is never used again by the JVM but it might get overwritten at some point.

During work on the J9 JVM it was discovered that several stack slots are popped off twice without ever accessing the value in the corresponding slot. Figure 4.1 gives an example explaining the problem and gives the solution. Object 42 resides on the stack and is already popped off in Figure 4.1 (a), as result a root set deletion is printed into the trace file. The stack pointer moves forward but eventually jumps in front of the already popped off stack slot as shown in Figure 4.1 (b). In case the stack pointer now jumps from this slot further down the stack (for example during a function return) the

same slot is popped off a second time (Figure 4.1 (c)). A simple solution to this problem is zeroing out popped off stack slots by replacing the original value with zero as indicated in Figure 4.1 (d).

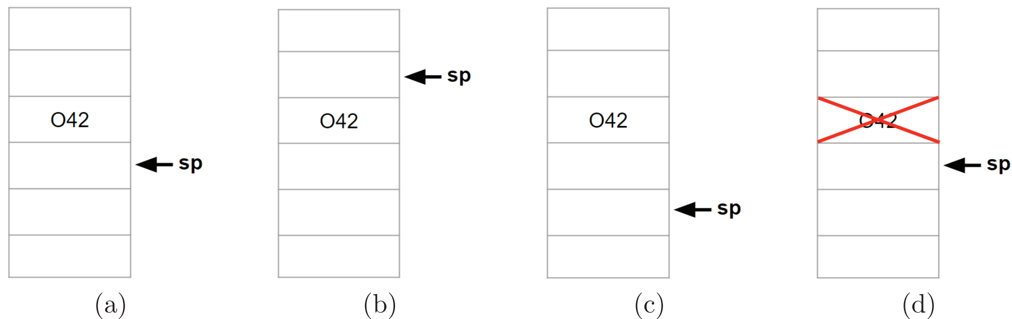

Figure 4.1: Example of zeroing out stack slots.

It is important to clarify that the JVM never accesses a popped off value twice. Multiple root set deletions from one stack slot only occur during the return of a function. The stack pointer jumps several slots down the stack and by doing so removes a whole stack frame. Values in these slots are never accessed directly. However, in order to capture all root set deletions, the instrumented JVM has to examine every slot and print a root set deletion for root pointers. By doing so, the instrumented JVM might access a stack slot twice, while the value is only used once by the JVM. Zeroing out stack slots each time a value is popped off prevents this from happening.

4.3 Zombie Removal Processor

Many modifications and additions to IBM's J9 were included in order to attempt to capture all root set operations. Unfortunately, it was not possible to remove all zombie objects within the instrumented JVM for every given input. Too many locations in J9's code operate on the stack and manipulate the root set. The instrumented JVM in version bytecode cannot guarantee to generate zombie free trace files. The final number of remaining zombies in trace files generated from the DaCapo benchmark suite is presented in Chapter 6.

A system was required that is able to remove zombie objects from the generated trace files. The proposed solution in this project uses the simulator together with a further processing step to remove the last zombie objects. The simulator was already modified to be able to capture zombies in trace files (see Section 3.1). It is used as part of the zombie removal system. An additional processing step was included that removes zombie objects. Figure 4.2 shows the final system from Java application to the generation of a zombie free trace file.

The zombie detection simulator uses reference counting to detect zombies. It generates a file that contains the lines at which the zombie objects actually become zombies (the first access to a dead object). This file, together with

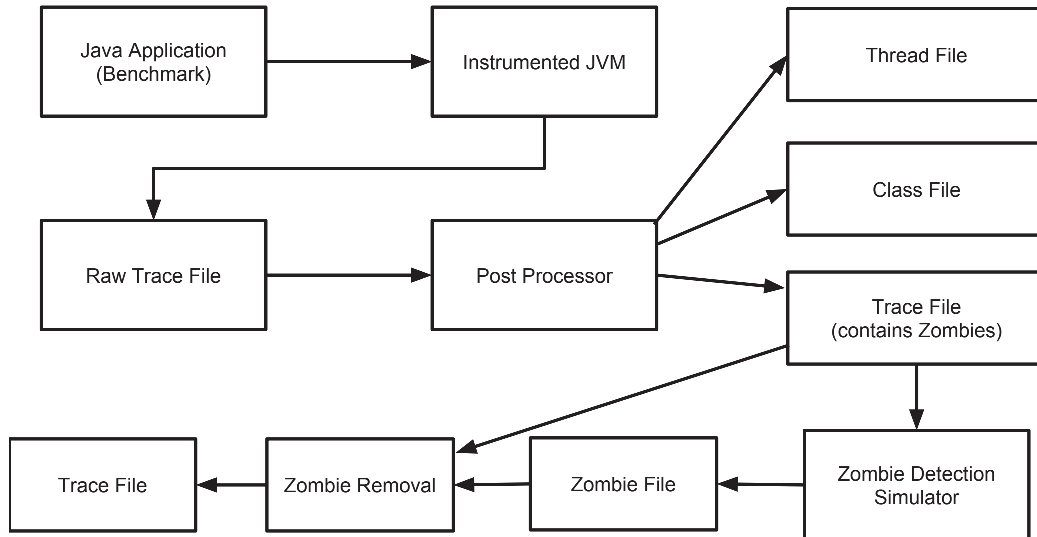

Figure 4.2: Final trace file generation process

the trace file, are the inputs for the zombie removal processor.

This processor removes zombies objects in two steps:

1. The first pass locates the last root set deletion for every zombie object before it actually becomes a zombie.

2. In a second pass, the whole trace file is copied unmodified except for one small change: the already located root set deletions are omitted and simply ignored.

The following example demonstrates the algorithm:

1: a T1 O1 S112 N9 C1

2: + T1 O1

3: r T1 O1

4: - T1 O1

5: r T1 O1

Object one becomes a zombie at the last read operation (line 5). This line number is printed into the zombie file by the zombie detection simulator. The zombie removal processor locates the last root set deletion before O1 becomes a zombie (at line 4). The last step is to copy every line, whereby the last root set deletion for every zombie object is omitted:

1: a T1 O1 S112 N9 C1

2: + T1 O1

3: r T1 O1

5: r T1 O1

The number of zombie objects that were removed by the zombie removal processor is presented in Section 6.1.

This approach removes all zombie objects. Unfortunately, yet another problem remains in trace files: negative root counts. A few objects have more root set deletions than root set additions, and thus resulting into a negative root count. This relates to the zombie objects. Not all root set operations are captured entirely within the instrumented JVM. Since some root pointers are missing, it can happen that more root set deletions than additions are captured for one object.

The zombie removal processor performs another step that removes root set deletions that lead to a negative root count. Here is an example with a negative root count for one object :

1: a T1 O1 S112 N9 C1

2: + T1 O1

3: a T1 O2 S112 N9 C1

4: + T1 O2

5: w T1 P2 #0 O1

6: - T1 O1

7: - T1 O1

Object 1 is removed from the roots at line 7 for a second time with only one initial root addition. It is accessible by object 2 (line 5), and thus not a zombie. The zombie removal processor counts the root set additions and deletions for every object while copying each line. In case the root count drops below zero, the appropriate line is omitted:

1: a T1 O1 S112 N9 C1

2: + T1 O1

3: a T1 O2 S112 N9 C1

4: + T1 O2

5: w T1 P2 #0 O1

6: - T1 O1

The number of removed root set deletions that resulted into a negative root count is presented in Section 6.2.

The final outcome is a zombie free trace file with slight modifications compared to the original trace file generated directly by the instrumented JVM version bytecode. Advantages and disadvantages of this approach are discussed in Chapter 7.

Chapter 5

Experimental Setup

This chapter gives an overview of the software and hardware together with the experimental setup of this project. Implementation details of the instrumented JVM, the post processors, as well as the simulators are presented.

5.1 Instrumented JVM

IBM's J9 Java Virtual Machine was used in its most recent release as the basis for this project. This is the precise JVM version number:

Java(TM) SE Runtime Environment (build pxa6490ea-20161205_06)

IBM J9 VM build 2.9, JRE 9 Linux amd64-64 12150101_000000

(JIT disabled, AOT disabled)

J9VM - bc3a74e

OMR - 4963113

JCL - 20161128_02 based on Oracle jdk-9+113

Both instrumented JVMs (root dump and bytecode version) are implemented on the basis of this Java Virtual Machine. The original instrumented JVM that preceded this work was implemented with IBM's J9 in build 2.7. During work on this project, both JVM instrumentations were ported to the latest release in 2.9. The reason for this migration was mainly due to assembly code in version 2.7. Many operations that manipulate the stack are performed within assembly code, including root set additions and deletions. These operations made the capture of root pointers particularly difficult and were enough reason to do the migration. IBM's J9 JVM in version 2.9 has few lines of assembly code. Almost all of the old code was migrated to C or C++. No root set additions or deletions were discovered from assembly code during work on this project in version 2.9.

The instrumented JVM itself was executed inside a virtual machine created with Vagrant. This was the most convenient way to run the JVM on one CPU only. A major drawback during work on this project was the multi threading nature of the JVM implementation. Especially the locking instructions (Chapter 4.2.3) often interfered with each other. Setting up the JVM on a Vagrant virtual machine with only one CPU decreased the overall locking levels significantly (more on locking levels in Section 6.3).
This is the precise Vagrant version number with its operating system:

Vagrant 1.8.6 VM

Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-108-generic x86_64)

1 CPU

The DaCapo benchmarks were all executed with the same JVM command line arguments in both instrumented JVMs:

-Xjvm:default -Xint -Xms16g -Xmx16g -Xgcpolicy:optthruput -Xgcthreads1

-Xgc:alwayscallwritebarrier -XX:-PackedObject

-jar dacapo-9.12-bach.jar benchmarkName -s small

The initial and maximum memory were both set to 16 GB in order to prevent garbage collection. The option *Xint* disables the Just-In Time (JIT) and Ahead-Of-Time (AOT) compilation. Optthruput is a mark-sweep collector with a disabled concurrent mark phase. The application only stops executing if an allocation request cannot be satisfied. This policy was selected in order to prevent any kind of interaction by the garbage collector. The number of GC threads was set to one. Packed objects are disabled, while write barriers are enabled in order to capture read and write operations.
The DaCapo benchmarks (fop in the previous example) are all executed with their small size option (-*s small*) in order to reduce file size and processing time of trace files.

One last implementation detail is the object identification number in trace

files. As presented in examples in previous chapters and sections, objects have a unique ID. However, objects in raw trace files do not have these IDs. They are identified by their address on the heap, which is printed into the raw trace file. Assigning and keeping identification numbers during the execution of the instrumented JVM is not a trivial task. Storing an ID directly with the object on the heap alters the original heap layout since every object requires one additional slot. An off heap data structure, such as a map, that stores all addresses and corresponding IDs, would be a possibility. However, a much simpler and faster solution is to transform addresses into identification numbers during post processing. While parsing the raw trace file, a map stores every address with its corresponding ID starting from one. Objects are printed into the processed trace file with their assigned ID.

Another problem that is solved during the transformation of addresses is the initial JVM heap expansion. IBM's J9 JVM always starts executing with an initial heap size. At some point, during startup, the heap expands to the size specified by Xms [11]. Heap expansion can only be performed after a garbage collection cycle. The initial GC results in the deletion of some objects on the heap. Newly allocated objects can be placed at previously used locations. As a consequence, an object can have a previously used address in an allocation operation in the trace file. A new ID is assigned to this object. Since the old object is already garbage collected in the J9, any future access to this address has to be to the new object and the new ID is used.

Verbose logs [12] show the initial heap expansion as well. A verbose log gives

information about GC cycles together with the heap size. The initial garbage collection, as well as the heap expansion, was found in the verbose log file.

5.2 Post Processor Version 1

The first post processor (Section 2.5.2.1) preceded this work. It was used as part of the results chapter, in order to reveal changes in new trace files. Post processor 1 is implemented in C++ and consists of two phases that are separated in two programs:

1. **raw2zombie** transforms raw trace files into processed trace files that might contain zombie objects. Thread and class files are generated as well.

2. **zombie2tracefile** moves root set deletions artificially below the last access for every object.

Addresses are transformed into identification numbers with a separate script implemented in Python 3.4.5. Both post processors use this script to transform addresses.

5.3 Post Processor Version 2 and 3

Both post processors in version 2 (Section 2.5.2.2) and 3 (Section 4.3) are implemented in Python 3.4.5. Post processor 2 was used as part of the zombie

search and raw trace file examination in Chapter 3. Trace files generated by the second post processor are not usable by the simulator due to remaining zombie objects. Post processor 3 transforms raw trace files created by the instrumented JVM in version bytecode (Section 4.2) into trace files that might contain zombie objects.

Post processors 2 and 3 have several independent processing steps that can be connected via pipes. All individual steps can be executed automatically by calling a bash script. Both post processors use the same Python programs except for *approximateRootsets.py*, which is only used by processor 2.

The five individual processing steps are the following:

1. **processThreads.py** generates the thread file.
2. **processClasses.py** generates the class file.
3. **transformIDs.py** transform addresses into identification numbers.
4. **formatLines.py** removes arguments that are not required in the processed trace file.
5. **approximateRootsets.py** replaces root set dumps found in its input trace file with approximated root set additions and deletions.

5.4 GarCoSim Simulator

The GarCoSim simulator (Section 2.5) is implemented in C++ and is able to simulate a JVM heap together with allocation, mutation, and garbage

collection. After execution, a log file is generated with information about the specified collector, allocator, write barrier, and heap size. Every GC cycle is printed with statistics for the number of freed and live objects as well as the amount of free and used heap. The simulator is started by executing *traceFileSim*.

Seven command line options are available:

1. The **trace file path** has to be the first argument when calling *traceFileSim*.

2. **heapsize** specifies the heap size in bytes. The default heap size is 600,000 bytes.

3. **logLocation** specifies the location of the log file. The default location is the path of the trace file.

4. **collector** followed by the collection policy specifies the garbage collector. Available collectors are: markSweep, traversal, and recycler. The default is traversal.

5. **traversal** specifies the traversal order during garbage collection. Available options are breadthFirst and depthFirst. The default is breadthFirst.

6. **writebarrier** followed by either recycler or referenceCounting activates the write barrier. It is disabled by default.

7. **finalGC** activates the final GC. It is performed after the last operation in the trace line.

5.5 Zombie Detection Simulator

The zombie detection simulator (Sections 3.1 and 4.3) is an extension of the GarCoSim simulator and implemented in C++. It features additional command line arguments that enable the detection of zombies as well as the generation of statistics that are required for the results.
These are the five available options:

1. **catchZombies** activates the detection of zombie objects. This option can only guarantee to catch all zombie objects with an enabled reference counting write barrier.

2. **countTraversalDepth** counts the traversal depth for every object during garbage collection. A log file is generated after execution that gives overall statistics about the traversal depth.

3. **countRootPointers** counts the number of root pointers per object during a garbage collection cycle. A log file gives details about the overall root count statistic.

4. **findNegativeRoots** captures objects with more root set deletions than additions. The appropriate objects are printed into a log file together with the final number of negative root pointers.

5. **printLockingStatistics** generates a log file with information about locking. The total number of locked and unlocked lines as well as locking levels are given.

5.6 Zombie Removal Processor

The zombie removal program (Section 4.3) is implemented in Python 3.4.5 and eliminates zombie objects by removing relevant root set deletions. It requires three input arguments: a trace file that contains zombie objects, a corresponding file with lines for zombies, and a path for the output trace file.

Chapter 6

Results

This chapter presents results and statistics on the final trace files generated by the DaCapo-9.12-bach benchmark suite. At first, statistics on the new trace files in version 3 are presented. The numbers of removed zombie objects and deleted root pointers are given. Furthermore, statistics on locking operations are presented. Finally, Section 6.4 shows several comparisons between trace files in versions 1 and 3.

6.1 Removed Zombie Objects

Trace files generated by the new instrumented JVM contain a small portion of zombie objects. Table 6.1 shows the number of remaining zombie objects that were found by the zombie detection simulator. These objects are artificially kept alive by the zombie removal processor. Zombie objects are shown as

quantity and percentage dependent on the number of allocated objects.

Benchmark	Allocated objects	Zombie objects	Zombie percentage
batik	570,544	1,220	0.21%
fop	654,380	5,961	0.91%
luindex	162,256	325	0.20%
lusearch	1,435,083	40,047	2.79%
pmd	211,390	1,121	0.53%
xalan	653,039	1,672	0.26%

Table 6.1: Remaining Zombie objects

The lusearch benchmark resulted in the highest number of zombie objects with more than 40,000 and 2.79%. Apart from lusearch, the number of zombies is manageable with less than 1% compared to the number of allocated objects.

6.2 Negative Root Set Count

This section gives results on the removal of root pointers that led to a negative root count. Due to the zombie object problem, and thus the absence of root pointers, some objects show a higher number of root set deletions than additions. For example an object with one root set addition and two deletions shows a negative root count. It is not possible to have more root set deletions than additions per object. These negative root pointers were removed in the zombie removal processor.

Benchmark	Removed root set deletions	Percentage
batik	15,002	0.0098%
fop	22,792	0.0173%
luindex	4,369	0.0094%
lusearch	4,364	0.0013%
pmd	9,542	0.0166%
xalan	13,396	0.0055%

Table 6.2: Negative root pointer deletions

Table 6.2 shows the numbers of removed root set deletions due to negative root pointers. The percentage of removed root set deletions compared to the overall number of root set deletions is presented as well. This percentage shows how small the number of removed root pointers really is.

6.3 Locking Statistics

Trace files in version 3 had to be extended by locking. These operations are required in order to keep objects alive until they are added to the roots. Statistics on the number of locked and unlocked lines are shown. Furthermore, locking levels are presented. Due to multi threading and function calls within functions, locks can be printed multiple times from different code locations before the unlock operation is reached. Therefore, locking levels can be greater than one at certain lines in the trace file. Locking levels are counted as the lock is set to the appropriate value. Fortunately, most of

the trace lines remain unlocked. Thus, the effect on the garbage collector is acceptable.

Table 6.3 shows the number of unlocked compared to locked lines as percentage per benchmark. Furthermore, Figure 6.1 shows the locking levels averaged over all benchmarks.

Benchmark	Unlocked lines percentage	Locked lines percentage
batik	70.84%	29.16%
fop	58.21%	41.79%
luindex	74.07%	25.93%
lusearch	85.81%	14.19%
pmd	48.95%	51.05%
xalan	78.05%	21.95%

Table 6.3: Locked and unlocked lines

6.4 Trace File Comparisons

This section gives multiple comparisons between former (version 1) and new (version 3) trace files.

6.4.1 File Size and Operation Count Statistics

The following tables show how trace file versions differ in file size as well as in the number of root set operations. As expected, trace files in version 3

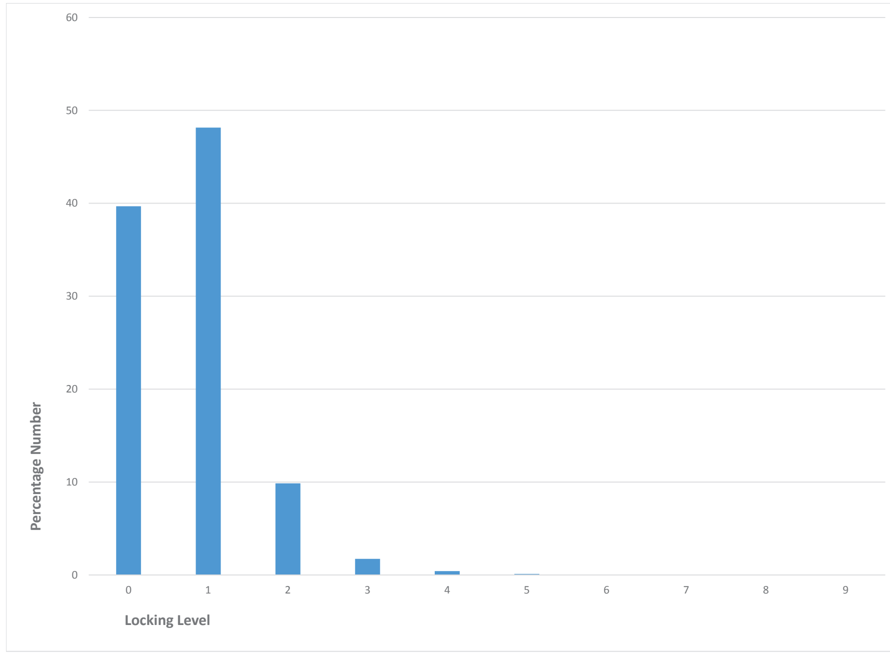

Figure 6.1: Locking levels averaged over all benchmarks

show a much higher number of root set additions and deletions. In fact, root set additions and deletions increased on average by 269 times. This is a big factor but shows how many root set operations were missing in former trace files. Post processor version 1 has to extract root additions and deletions from root dumps. Information about root pointers in between two dumps is lost. The Java Virtual Machine executes operations mainly from and on the stack. Many of these operations involve adding and deleting root pointers. Trace files in version 1 lack these operations. This explains the big difference

of root set operations between the two trace file versions.

Every benchmark is compared between version 1 and 3. The percentage of root additions and deletions compared to the overall number of root operations is presented as well. The optimal outcome for the ratio of root additions and deletions is 50% each. This would indicate that every root pointer is popped off at the end of execution. The percentages of root operations in the following tables show how close trace files in version 3 are to the 50% mark. The ratio from version 1 to 3 for file size as well as the number and percentage of root pointers is included as well.

batik	Version 1	Version 3	Ratio
File size	5.48 GB	7.88 GB	1.438
Root additions	591,005	154,190,729	260.896
Root deletions	568,419	153,214,641	269.545
Root additions percentage	50.97%	50.16%	0.984
Root deletions percentage	49.03%	49.84%	1.017

Table 6.4: batik Comparisons

fop	Version 1	Version 3	Ratio
File size	3.33 GB	5.83 GB	1.751
Root additions	674,820	132,667,212	196.596
Root deletions	652,488	131,773,678	201.956
Root additions percentage	50.84%	50.17%	0.987
Root deletions percentage	49.16%	49.83%	1.014

Table 6.5: fop Comparisons

luindex	Version 1	Version 3	Ratio
File size	1.46 GB	2.24 GB	1.534
Root additions	171,100	46,719,839	273.056
Root deletions	158,761	46,423,746	292.413
Root additions percentage	51.87%	50.16%	0.967
Root deletions percentage	48.13%	49.84%	1.036

Table 6.6: luindex Comparisons

lusearch	Version 1	Version 3	Ratio
File size	11.0 GB	17.1 GB	1.555
Root additions	1,443,473	339,722,226	235.351
Root deletion	1,433,072	339,096,789	236.622
Root additions percentage	50.18%	50.05%	0.997
Root deletions percentage	49.82%	49.95%	1.003

Table 6.7: lusearch Comparisons

pmd	Version 1	Version 3	Ratio
File size	1.54 GB	2.52 GB	1.636
Root additions	224,897	57,787,026	256.949
Root deletions	208,374	57,622,578	276.534
Root additions percentage	51.91%	50.07%	0.964
Root deletions percentage	48.09%	49.93%	1.038

Table 6.8: pmd Comparisons

xalan	Version 1	Version 3	Ratio
File size	7.97 GB	12.3 GB	1.543
Root additions	689,374	246,011,035	356.861
Root deletions	658,498	245,047,181	372.130
Root additions percentage	51.15%	50.10%	0.979
Root deletions percentage	48.85%	49.90%	1.021

Table 6.9: xalan Comparisons

6.4.2 Traversal Depth During GC

The traversal depth during garbage collection shows how far the collector has to go down the object tree in order to reach all live objects. Objects that are accessible by the roots are level one. Children of root objects are level two, and so on. The collection cycle, that generated these results, was triggered at the end of execution. Table 6.10 shows the traversal depth averaged over all reachable objects during GC. The percentage change from version 1 to version 3 is presented as well.

Benchmark	Version 1	Version 3	Difference
batik	10.43	2.10	-79.87%
fop	5.78	1.98	-65.74%
luindex	3.81	1.84	-51.71%
lusearch	13.83	2.47	-82.14%
pmd	6.11	2.28	-62.68%
xalan	3.81	2.52	-33.86%

Table 6.10: Traversal depth during GC

The results show a significant reduction of the traversal depth of trace files in version 3. This outcome was not expected at the beginning of this project.

Rather the opposite was anticipated:

Trace files in version 1 are generated with post processor 1. This post processor removes the last root set reference for every object after its last access in the trace file. Every object is accessible directly by the roots as long as it is used in further operations. This artificial root set manipulation keeps objects accessible longer by the roots than they really are compared to the JVM. Therefore, the traversal depth should be lower compared to the JVM implementation.

The anticipated result of new trace files was a higher traversal depth compared to the previous version. Objects are no longer artificially kept alive in the root set. Root set deletions are printed into the trace file as soon as they occur in the JVM.

The final result showed that the opposite has happened: the traversal depth levels are much lower in the new trace files compared to the previous version. These low traversal depth numbers are due to the absence of root set deletions that are not captured correctly in the instrumented JVM. Tables 6.4 to 6.9 show the number of root additions compared to root deletions for both version 1 and 3. The ratio of root set operations is closer to 50% in version 3 trace files in all benchmarks. However, due to the vast increase of root set operations in new trace files, more root set entries are found in new trace files with merely the same number of allocated objects.

Tables 6.11 and 6.12 give more insight about the reason for low traversal depth levels. Table 6.11 shows results of version 1 trace files and Table 6.12

of version 3. The first column gives the number of allocated objects. These numbers differ between the two versions. Every execution with the instrumented JVM with the same input and arguments results in a slightly different output. Therefore, the number of allocations can be different in every execution. The root set pointers at the end of execution are presented as well. This number is calculated by subtracting the root set deletions from root set additions. It shows how many pointers remain in the root set at the end of execution. The last column shows the ratio of root set pointers per allocated objects. This ratio clarifies how big the difference between the two trace file versions really is.

Benchmark	Allocated Objects	Root set pointers	Ratio
batik	570,423	22,586	0.0396
fop	654,430	22,332	0.0341
luindex	162,360	12,339	0.0760
lusearch	1,435,073	10,401	0.0072
pmd	211,421	16,523	0.0782
xalan	664,503	30,876	0.0465

Table 6.11: Root set pointers version 1

The number of root set pointers is significantly higher with trace files in version 3 compared to version 1. On average, every object in version 1 trace files is accessible by 0.0469 root pointers at the end of execution. With version 3 trace files the average number is 1.2652, an increase by a factor of 26.96. This

Benchmark	Allocated Objects	Root set pointers	Ratio
batik	570,544	976,088	1.7108
fop	654,380	893,534	1.3655
luindex	162,256	296,093	1.8249
lusearch	1,435,083	625,437	0.4358
pmd	211,390	164,448	0.7779
xalan	653,039	963,854	1.4760

Table 6.12: Root set pointers version 3

increase of root set pointers in version 3 compared to version 1 decreases the traversal depth level significantly, since more objects are directly accessible by the roots.

This analysis solely takes the number of root set additions and deletions from trace files into account. Root set entries that are created in the simulator by static reference operations from class objects are not considered in Tables 6.11 and 6.12. These static class operations add a root set entry to a specified object in every thread's root set.

6.4.3 Root Count During GC

The root count during garbage collection is a similar statistic compared to the traversal depth. The incoming root pointers per object are counted during the final GC at the end of execution. Objects that show the same number of root pointers are added together. Table 6.13 shows the root count averaged over all objects. The difference from version 1 to version 3 is presented as well.

Benchmark	Version 1	Version 3	Difference
batik	9.05	6.29	-30.50%
fop	7.64	5.55	-27.36%
luindex	3.75	5.90	57.33%
lusearch	5.03	2.72	-45.92%
pmd	5.49	3.45	-37.16%
xalan	4.45	16.04	260.45%

Table 6.13: Average root count during GC

The luindex and xalan benchmarks in version 3 are the only benchmarks with a higher average root count compared to version 1. Both luindex and xalan showed a couple of objects during the final garbage collections with a root count of more than 10,000. The highest root count for one object was discovered in the xalan benchmark with 51,331 root entries. These objects are responsible for the overall increase of the average root count. The highest root count in the xalan version 1 benchmark is 711. In version 1, only the benchmarks batik and fop show two objects each with a higher root count than 1,000.

Chapter 7

Conclusions and Future Work

The goal of this thesis was to generate realistic trace files that can be used as input for a JVM memory management simulator. The approach presented in this work captures operations within an instrumented IBM's J9 JVM. The main challenge was the correct representation of the root set. Therefore, the bytecode interpreter was monitored in order to capture root additions and deletions.

Unexpected conditions, such as root operations outside the bytecode interpreter, references from internal JVM data structures, and the necessity of locking operations, complicated this project. Most of the implementation time on this project went into the fight against zombie objects. Numerous work cycles dealt with detecting locations in the J9 that are responsible for the creation of zombie objects. The final version of the instrumented JVM captures most root set operations, and thus the number of zombies was min-

imized.

Unfortunately, it was not possible to remove every zombie object. Thereby, the final project was extended by a zombie removal processor. With this relatively minor distortion of the original layout of root pointers, trace files are usable by the simulator.

A comparison between former and new trace files showed a significant change in the traversal depth during garbage collection. As explained in Section 6.4.2, at the beginning of this project, a higher traversal depth was expected with new trace files compared to the previous version. The traversal depth of trace files in version 3 is around two, and thus lower than expected.

The relatively small remaining number of zombie objects in raw trace files, and thus the required removal of root set deletions is the final result. It is questionable that it is possible to remove all zombie objects inside the instrumented JVM for every given input with the presented approach.

It is difficult to say whether newly created trace files in version 3 are closer to the actual JVM implementation compared to trace files in version 1. Both versions come with flaws: version 1 keeps every object artificially alive in the root set, whereas version 3 trace files have a low traversal depth level due to too many root set additions. It is fair to say that neither trace file version represents a JVM completely realistically. The presented approach in this thesis showed how difficult it really is to capture all root operations in the JVM correctly.

There are some possibilities for future work around the generation of realistic trace files. One approach that might be able to deliver better results is the generation of trace files with a simpler and smaller JVM implementation compared to IBM's J9 JVM. A possible downside of these trace files might be that they do not represent a competitive JVM implementation. Another project could be the extension of the synthetic trace file generator. If benchmarks can be analyzed in order to find out their effects on a JVM implementation, the trace file generator might be able to create trace files that correspond to benchmarks and the JVM. Finally, a different setup or extension of the instrumented JVM could improve results. For example, garbage collection could be forced periodically within the JVM. By doing so, objects that are already garbage are detected. This information could be used during post processing.

Bibliography

[1] *The dacapo benchmark suite*, `http://dacapobench.org/`, Accessed: 2016-09-12.

[2] *Garcosim framework on github*, `https://github.com/GarCoSim/`, Accessed: 2016-09-12.

[3] *Standard performance evaluation corporation (spec's benchmarks)*, `https://www.spec.org/benchmarks.html`, Accessed: 2016-09-12.

[4] *Sun announces availability of the java hotspot performance engine*, `http://www.thefreelibrary.com/Sun+Announces+Availability+of+the+Java+HotSpot+Performance+Engine%3B...-a054477747`, Accessed: 2016-09-08.

[5] *Tiobe index for august 2016*, `http://www.tiobe.com/tiobe-index/`, Accessed: 2016-08-22.

[6] *What's new in jdk 8*, `http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html`, Accessed: 2016-10-11.

[7] David F. Bacon, Clement R. Attanasio, Han Bok Lee, V. T. Rajan, and Stephen E. Smith, *Java without the coffee breaks: A nonintrusive multiprocessor garbage collector*, ACM SIGPLAN Conference on Programming Language Design and Implementation (Snowbird, UT), ACM SIGPLAN Notices 36(5), ACM Press, June 2001, pp. 92–103.

[8] James D Bloom, *Jvm internals*, `http://blog.jamesdbloom.com/JVMInternals.html`, Accessed: 2017-01-03.

[9] Sylvia Dieckmann and Urs Hölzle, *A study of the allocation behavior of the SPECjvm98 Java benchmarks*, 13th European Conference on Object-Oriented Programming (Lisbon, Portugal) (Rachid Guerraoui, ed.), Lecture Notes in Computer Science, vol. 1628, Springer-Verlag, July 1999, pp. 92–115.

[10] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley, *The java language specification, java se 8 edition*, 1st ed., Addison-Wesley Professional, 2014.

[11] IBM, *Heap memory and garbage collection*, `https://www.ibm.com/support/knowledgecenter/en/SS6QYM_9.1.0/com.ibm.help.perf.manage.doc/c_FND_PM_IBMJ9JVMHeapMemoryAndGarbageCollection.html`, Accessed: 2017-05-04.

[12] ———, *Verbose garbage collection logging*, https://www.ibm.com/support/knowledgecenter/SSYKE2_7.1.0/com.ibm.java.aix.71.doc/diag/tools/gcpd_verbosegc.html, Accessed: 2017-05-04.

[13] Richard Jones, Antony Hosking, and Eliot Moss, *The garbage collection handbook: The art of automatic memory management*, 1st ed., Chapman & Hall/CRC, 2011.

[14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley, *The java virtual machine specification, java se 7 edition*, 1st ed., Addison-Wesley Professional, 2013.

[15] ———, *The java virtual machine specification, java se 8 edition*, 1st ed., Addison-Wesley Professional, 2014.

[16] K. B. Kent M. M. Rahman, K. Nasartschuk and G. W. Dueck, *Trace files for automatic memory management systems*, 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016.

[17] John McCarthy, *Recursive functions of symbolic expressions and their computation by machine, Part I*, Communications of the ACM **3** (1960), no. 4, 184–195.

[18] Konstantin Nasartschuk, Marcel Dombrowski, Tristan Basa, Mazder Rahman, Kenneth Kent, and Gerhard Dueck, *Garcosim: A framework*

for automated memory management research and evaluation, Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'15) (Berlin, Germany), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016, pp. 263–268.

[19] Oracle, *The history of java technology*, `http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html`, Accessed: 2017-04-12.

[20] Ryan Sciampacone, Peter Burka, and Aleksandar Micic, *Garbage collection in WebSphere application server V8, part 2: Balanced garbage collection as a new option*, IBM Middleware Technical Journal for Developers (2011).

[21] James E. Smith and Ravi Nair, *The architecture of virtual machines*, Computer **38** (2005), no. 5, 32–38.

Vita

Candidate's full name: Johannes Ilisei

Universities attended:

Esslingen University of Applied Sciences (2010 - 2014)
Bachelor of Engineering in Software Engineering

University of New Brunswick (2015 - 2017)
Master of Science in Computer Science

Publications: N/A
Conference Presentations: N/A