Virtual Machine-Assisted Collaborative Junk Object Detection

by

Ming Matthew Ma

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Memory leak is unrecoverable software bug that causes performance degradation and reliability issues to software applications. Although memory management systems exist in modern Object Oriented Language to reclaim unused memory store, memory leak can still happen, and continue to exhaust memory resource. In Java, where there is garbage collection for releasing unused objects, memory leaks manifest itself in the form of unused object retention. Since Java Programming language allocates objects on heap, the lifetime of an object deviates from the stack discipline, which can be a challenge in detecting Java memory leak.

In this thesis, we propose a collaborative approach in detecting Java memory leaks through verifying Object Lifetime Specification at runtime. We designed a runtime verifier that leverages Java Virtual Machine technologies to monitor and extract annotated information from the user application, and use that information to verify against Java Virtual Machine events to detect unintentional object retention in the Java application under test.

We implemented our runtime verifier with Maxine Virtual Machine, an open source, meta-circular virtual machine developed by Oracle Lab, and conducted experiments and DaCapo benchmark to evaluate its accuracy and performance efficiency. The results show that the runtime verification tool successfully identifies junk objects for different semantic cases proposed in this thesis with certain runtime overhead. Through the research and experimental results, we further make implications on how to improve the performance overhead associated with current design and implementation methods in detecting unused object retention, which in the long term constitute memory leak and performance bug.

Acknowledgements

I would like to thank all the people who made this thesis possible. I would like to thank my advisor, Professor Derek Rayside for providing me guidance throughout the two years of my graduate study and research. I would also like to express my gratitude to Professor Werner Dietl for providing me invaluable suggestions and help. I also wish to thank Mr. Mick Jordan and Mr. Simon Douglas from Oracle Corporation who provided me insight to Maxine VM design and responded to my issue report to Maxine VM. I thank all other colleagues from my research lab for discussions and most of all, great friends for me. Lastly, I would like to express my sincere appreciation to Professor Werner Dietl and Professor Krzysztof Czarnecki for reading my thesis and providing valuable feedback and suggestions.

Dedication

This thesis is dedicated to my mother and my father for educating me since I was born, and educated me to become a responsible and upright person.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software performance and reliability are important to be considered before putting a software program into release. High throughput rate, consistent behavior and deterministic results are particularly important to software applications that run extended amount of time, such as server side programs. Software verification and testing are key constituents in program analysis, and play an integral role in developing reliable bug-free software applications. While verification can find the satisfiability of the specifications through model checking, testing can take various forms to uncover software bugs statically or dynamically. Software bugs are mostly language independent; however, the differences in language design, resource management and runtime execution strategies would result in distinct branch of techniques in how the same type of bug is to be detected. For instance, we often use the term, reachability in memory leak related bugs; however, the role which reachability takes can be quite different for different programming languages. Memory leak refers to objects/storage that is no longer needed, but is not released. Those unneeded objects will be persistent in the program until the program terminates, and if they accumulate, they will use up the amount of available memory, and cause the system to slow down or crash. For language, such as C/C++, where explicit memory management is used, memory leak occurs when an object is allocated but never deallocated due to the lost reference to the unneeded object. In this case, it is said that the object is no longer reachable and is a memory leak. On the other hand, in garbage collected language, such as Java, when objects are still reachable but no longer used, this unintentional object retention is considered as a constituent to memory leak in long term. Therefore, it can be seen that different methodologies and strategies needs to be applied in verification as well as testing to detect and fix the same type of software bugs.

Computer memory is the fundamental unit in storing data used by the system or

application processing unit. Since it is finite, running short of memory not only degrades performance by lowering throughput rate but also incurs reliability issues. Memory leak bug can ultimately use up all the memory space allocated, and causes the software to crash or system to halt. This is extremely undesirable behavior for both the software provider as well as to the clients for their loss resulted from inefficient software performance, wasted computing resources and low reliability.

In this thesis, we focus on memory leak in Software programs written in Java, and investigate the runtime verification approach to detect unused object retention, therefore eliminating the performance bottleneck caused by memory shortage and its resultant software reliability issues. Memory leak in the context of this thesis is both considered as a design bug as well as a performance bug. However, the way we detect this type of bug is not through traditional performance monitoring/testing, such as heap analysis and load test. We develop a runtime verifier to find the discrepancies between user specified Object Lifetime Specifications and JVM garbage collection events to detect Java memory leak. Our contribution is developing a collaborative framework between user and JVM, and apply Object Lifetime Specifications in detecting Java memory leak.

1.1 Why Java

Java was first introduced in 1995. In early 90s, when extending the power of network computing to the activities of everyday life was a radical vision, Java project was initiated by a small group of Sun engineers led by James Gosling [46]. Now, Java not only permeates in the Internet, but also has been the important technology behind many applications and devices that power people's everyday life, which Oracle Corporation depicts as "invisible force" [47]. Java technology also inspires other programming language, and is widely used in smartphone, handheld game apps and navigation systems as well as e-business solutions. Java's wide application usage has its own reasons. Oracle articulates the design principle of Java language concisely in the five points listed below[35].

- Simple, object-oriented and familiar
- Robust and secure
- Architecture-neutral and portable
- Execute with "high performance"
- Interpreted, threaded, and dynamic

Simplicity and extensive built-in libraries allows programmer apply theory to the practice with short startup time. Yet, the language design principle to be object oriented from day one, give Java the ability to function within increasingly complex computing environment. Extensive compile-time checking, followed by the runtime checking catches syntactic errors and even semantic errors which makes Java reliable during software deployment. As a distinguishing nature, Java uses two-step compilation process; the Java source code is compiled into bytecode, and then this platform neutral bytecode is executed by Java Virtual Machine (JVM). In other words, Java program can be executed in any platform where JVM is installed. Portability scarifies code optimization to certain degree; however, with the Just-in-time (JIT) compilation support from JVM, it is possible to compile the bytecode to the native instructions of "hot spot" for the CPU in the deployment platform on the fly, and consequently improving the program performance. Furthermore, in an Java interpreted platform, the link phase of a program is simple, incremental, and lightweight. For all the above nature, Java language has been used widely in different area of software application, and makes it particularly interesting to study and research on.

1.2 Java Memory Allocation

Java language uses dynamic memory allocation scheme which differs from other object oriented language, such as C++ where objects can be statically allocated on the stack. Although primitives and local references are stored on the stack, objects are all allocated onto heap based on runtime information. Object allocation in the heap is costly in terms of performance as articulated in other literatures [7, 4]. However, Java runtime takes advantage of the JVM memory management model to reduce the memory page swapping and fragmentation. This is further discussed in Section 3.3, nevertheless, it is sound that memory related activities (desired one and undesired one) work closely with Java runtime that results from JVM specifications and it has big influence to the software performance, which is important to be researched on.

1.3 Runtime Verifier

Given a specification M, some property φ and satisfaction relation, verification is the process of determining whether:

$$M \models \phi \tag{1.1}$$

Verification contains two subcategories, runtime verification and static verification. Dynamic verification is performed during the software program execution, and it is concerned with monitoring and analysis of software and hardware system. Runtime verification techniques are crucial for system correctness, reliability, and robustness; they are significantly more powerful and versatile than conventional testing[11], and more practical than exhaustive verification approaches. Runtime verification can be used prior to deployment, for testing, verification, and debugging purposes.

Java language and runtime system are dynamic in their linking stages. Memory allocation are done dynamically at runtime. In this thesis, we design a runtime verifier to uncover unused objects in Java. Our runtime verifier tool verify against what is specified in the Object Lifetime Specifications to the actual JVM events. The design goal of this tool is to assist debugging and verification in performance testing prior to the deployment. As a research goal, we also want to find implication on what is recommended program design, coding structure and potential improvements in language design, that can reduce the risk of memory leak, and improve the performance of Java applications.

1.4 Thesis Organization

This thesis consists of eight chapters with Chapter 1 being motivation and background introduction, Chapter 8 being the conclusion.

In Chapter 2, we start our discussion by describing the effect of memory leak to application performance. We then provides four types of Java memory leak followed by examples. We conclude Chapter 2 by articulating the challenges we face with traditional approach in uncovering memory leak in Java.

Chapter 3 is dedicated to Java Virtual Machine discussion as JVM technologies are critical to our runtime verifier. We discuss about internal components of JVM, JIT optimization that improves the performance of the application, and finally the memory management system in JVM.

Chapter 4 introduces Object Lifetime Specification, which is an important concept we use in our runtime verification. We start our discussion from describing what is Object Lifetime, and how we can use it to allow deviation from stack discipline. We also explain how Group is used to associate objects into a cluster of same Object Lifetime Specification. We then use an example to show how we can use Java annotation to embed Object Lifetime Specification into the program code.

In Chapter 5, we start our discussion and analysis on our runtime verifier design and implementation details. We first brainstorm and propose the design architecture by identifying key components of the runtime verifier. We then describes our implementation strategy and methodology of the runtime verifier with a meta-circular VM, Maxine VM; we provide implementation details with respect to each functional component of the runtime verifier, and modifications on Maxine VM. Finally, we describe implementation with Jikes RVM, an alternative JVM used when we had to wait for Maxine project team to fix a stack related bug which we reported.

In Chapter 6, we conduct experimental analysis on our runtime verifier, in terms of its semantic accuracy and performance. We show performance evaluation with Dacapo Benchmark; we identified the major source of performance slowdown, and provide improved implementation to reduce the amount of overhead associated with our runtime verifier.

In Chapter 7, we discuss and analyze the limitations to our tool that comes from design, choice of JVM, and provide discussions that leads to future improvement. We also discuss related works in each major concept that we used in completing our runtime verifier.

Chapter 8 is the concluding paragraph where we summarize our methodologies, implementation and achievements.

Chapter 2

Java Application Performance and Memory Leak

Software performance is one of the key criterion in software evaluation. As the functionality and the structure of the software gets more and more complex, and as we employ new distributed and large scale systems which encompass multiple components and services, especially in enterprise applications, the performance gain or loss gets enlarged to have bigger influence to the system. System states determine the way requests are processed; processor, current system load, network usage, as well as the complexity of the request and other application factors impact the application's responsiveness. The characteristics of responsiveness can be divided into three basic measures to characterize the performance of the application [54]:

Response time: Response time is a direct measure of how long it takes for the application to process a request, and it is one of the most widely used metric for performance measurements.

Throughput: Throughput is a straightforward count of the number of requests that the application can process within a defined time interval. Throughput rate can be defined slightly differently for different applications; for instance, in web related applications, a count of requests or page impressions per second is used as a measure of throughput rate.

System availability: System availability is usually expressed as a percentage of application running time minus the time the application cannot be accessed by users. This is an indispensable metric, because both response time nor throughput are zero when the system is unavailable.

Performance can also be defined by resource requests and by measuring resource requests in relation to throughput [54]. This becomes a key metric in the context of resource planning. For example, to know which resources are needed to achieve full application scaling, is directly related to understanding the frequency of requests for each resource type. Therefore, performance can be seen to have direct correlation with resource consumption in this resource-centric view.

2.1 Memory Management and Performance Degradation

Memory is the fundamental resource as an storage in storing data used by the system or application processing unit. As a finite resource, it can become performance bottleneck, and can affect the throughput and reliability of a running application. In Java, memory management involves various components from the JVM, and also runtime information which makes it complicated to analyze and predict. When a Java application is started, JVM obtain memory from the operating system. JVM generally has five discrete memory areas: Text Resident Segment, Permanent Generation, Code Cache, Active Heap and Garbage Heap (details discussed in Section 3.2). The design of JVM is to insulate user applications from the host machine platform-specific memory layout, and let it work only with an isolated memory space, which is known as Java heap memory. Java heap space is the runtime data area, and is located at bottom of address space which grow upwards. Whenever Java application creates an object using new operator, the object is allocated memory from Java heap and when object is no longer referenced by other active objects, it is garbage collected, and its memory is put back to heap space.

Although the garbage collection process effectively free heap space for new object allocation it involves determining a safe point and "lock"(or partially) the heap area that it is going to operate on. This means it not only incur certain performance overhead in monitoring and determining the safe point but also inevitably slow down the application thread as user thread cannot have access to the heap area that is to be garbage collected. For this reason, JVM has its own garbage collection mechanism to determine when is the "good time" to do garbage collection, and controls the frequency of garbage collection to not degrade application performance; in other words, even an object has no reference, it will not be garbage collated right away. However, if the application heap is continue to grow to certain limit, garbage collection will take place to free the heap space by releasing unreferenced objects. Therefore, increased memory footprint incur more frequent garbage collection, therefore, causing throughput rate to decrease. This leads to our motivation in

keeping the heap "clean" by detecting unused junk object that is retained in heap space without user's knowledge.

Unintentional object retention due to design error, misunderstanding of Java language specification or sheer programmer's mistake causes heap to be filled with unused objects, which not only incur performance overhead associated in triggering more frequent garbage collection, but also potentially cause the application to running out of memory, and leads to crash or other unstable behavior.

2.2 Java Memory Leak Examples

Memory leak refers to objects/storage that is no longer needed, but is not released. Those unneeded objects will be persistent in the program until the program terminates, and if they accumulate, they will limit the amount of available memory, and cause the system to slow down or crash. For garbage collected language, such as Java, objects are automatically reclaimed during Garbage Collection; however, memory leak can still occur in a form as unused objects retention even after their intended lifetime. In Java, memory leak occurs when the references to unneeded objects are retained so that reference counting-based approach and reachability test falsefully categorizes the object as active and makes it ineligible for Garbage Collection (GC).

The type of memory leak that we want to detect in this context is the design bug which manifest itself in Java applications. Unintentional object retention can be a semantic error where programmer forgets to null the reference of an obsolete object. This type of memory leak will silently manifest itself in the program with the effect of low performance caused by increased memory footprint; it might cause disk paging or program failure due to running out of memory which incur severe reliability problems [9].

To understand better about the correlation between Java memory leak with application performance, and also to study potential occurrences and challenges associated with detecting unused object retention, we present a subset of Java memory leak led by unused objects retention. These examples range from Java language specifications to code design issues which leads to unintentional object retention.

2.2.1 Dormant Reference

An typical memory leak in Java is *Dormant reference*. As the name indicates, it means reference to an object persist after its suppose lifetime. This happens when a reference

is created when the object is active, and the reference persists after the object becomes unused. Sample code snippet provided in Figure 2.1 illustrates this. In an imaginary server application, `cdatalog` object is used during setup process, and it should be no longer needed after the setup completes. However, the user fail to abide with the lifetime specification of `cdatalog` object, and due to the declaration, `cdatalog` is persistent until the belonging instance of `SocketServer` class is garbage collected. Since the lifetime of `SocketServer` instance might be much longer than that of `cdatalog`, `cdatalog` object lives much longer than its intended lifetime. If `cdatalog` object consists of other data structures, its size might be large enough to be influential as unneeded memory consumption.

```
public class SocketServer {

    private ServerSocket serverSocket;
    private int port;
    private ConfigDataLog cdatalog; //a config entity that is self-logging
    private Data dataA;
    private boolean isSetup;

    public SocketServer(int port, Data config) {
        this.port = port;
        cdatalog = new ConfigDataLog(config); //instantiate
    }

    public void start() throws IOException {
        System.out.println("Starting the socket server at port:" + port);
        serverSocket = new ServerSocket(port);

        isSetup = cdatalog.setup(); //do set up
    if(isSetup){
        System.out.println("Waiting for clients...");
        Socket client = serverSocket.accept();
        sendWelcomeMessage(client);
        run();
    }

    …
}
```

Figure 2.1: *Memory Leak due to Dormant Reference*

This type of memory leak is the most common and well-known memory management anti-pattern [52, 14], and the solution is to remove the reference of unused object by setting it to null or remove the element from the dynamic data structure or use weak references if necessary.

2.2.2 Scope and Static Field

Another type of design related memory leak problem results from misunderstanding in the scope of a variable. Scope, identify the lifecycle of object. For instance, if user declares a variable inside a method, this is called a method scoped variable; the variable only exists while that method is running and is eligible for garbage collection when the method exits. If user declares a variable inside a class definition, then it is object scoped. The variable will be garbage collected as soon as there is no reference to that object. Static variable is unique in the sense that it is class scoped. The classes in java are usually loaded at startup and its lifetime bounds to the JVM. If user declare a variable as static it will live for the entire lifetime of the JVM, unless it is individually nulled out. If a static variable references non static objects, it effectively makes those objects static as well. Therefore, misuse of static variable leads to memory leak in Java.

Static fields and collections are often used to hold caches or share state across threads. As discussed, mutable static fields need to be nullified explicitly. If the user does not consider every reference that static variable refers to, the release will not take place, resulting in a memory leak.

2.2.3 Java Collection Framework

Java Collection Framework is introduced in Java 1.2 to replace legacy class, such as Vector, and also provide a common interface for existing data structures. It is widely used to hold caches, page map in various applications. It is also prone to memory leak if object reference is not dealt appropriately. Since elements in the Collection is often used by other objects, so it is subjected to *Zombie reference*, which is a reference to an object that is no longer used. Rayside et.al pointed out that if a junk object is created, it is possible to exacerbate the problem through object interactions[52]. For instance, object A has reference to a junk object, B; object C might also create a reference to B based on the fact that there is A already points to B. This will make the removal of junk object B harder. Figure 2.2 below illustrate this.

```
public class Stack {
    private Object[] element;
        private int size = 0;
        ...
        public void push(Object e) {
            ...
            element[size++] = e;
        }
        public Object pop() {
            if (size == 0)
                 throw new EmptyStackException();
            //memory leak; should set theremoved object to null before return
            return element[--size];
    }
}
```

Figure 2.2: *Memory Leak due to Improper Handling of Collection Element*

In this example of implementing stack using array, the user forget to set the Object to null after decrementing the size count in the pop method. Although the size of the stack is reduced by one, the size of memory it holds is unchanged without user's knowledge.

This problem can be mitigated by using standard data structure from the Collection framework; remove-related method in standard collection will set the object reference to null automatically. However, design error can continue to make Collection vulnerable for memory leak. Consider the use of ArrayList from the Collection framework in the example below (Figure 2.3). Due to improper boundary selection in deleteData method, one element in the ArrayList is leaked for each iteration defined in the outer for loop.

```
public class DataPacket{

    private static List myArrayList = new ArrayList();
    private int iteration, count;
    ...

    public static void constructData(Message msg) {
        for (int i=0; i<iteration; i++) {
            for (int n=0; n<count; n++) {
                myArrayList.add(Integer.toString(n+i));
            }
        }
        Log.writeln(msg);
    }

    public static void deleteData(Message msg) {
        for (int i=0; i<iteration; i++) {
                // design error: n should be greater or equal to zero
            for (int n=count-1; n>0; n--) {
                myArrayList.remove(n);
            }
        }
        System.out.println(myArrayList.size());
        Log.writeln(msg);
    }

}
```

Figure 2.3: *Memory Leak in Collection*

2.2.4 Inner Class Example

The Java inner class is when a class definition is contained within another class. There are two types of inner class, the static one and non-static one. The key characteristics of the two is summarized in Table 2.1 .

Type	Static Inner class	Non-Static Inner class
Construction	Do not require an instance of the containing class to be constructed	Require an instance of the containing class to be constructed
Reference	May not reference the containing class members without an explicit reference	Automatically have an implicit reference to the containing instance
Lifetime	Have its own lifetime	Lifetime is supposed to be no longer than that of the container

Table 2.1: Characteristic of Static and Non-static Inn class

The memory leak arises when a non-static inner class lives longer than its container. Since there is an implicit reference from the inner class to the containing class, the container object becomes memory leak. This can happen if an object outside of the containing class keeps a reference to the inner object, without regard to the containing object. This leads to a situation where the inner object is alive but the references to the containing object has already been removed from all other objects. Therefore, the inner class object is keeping the containing object alive with the implicit reference.

This type of memory leak is widely seen in Android Application, in which Activity, View and Context is used frequently. An Activity is an application component that provides a screen with which users can interact in order to do some task[1]. Activities contain a lot of information to be able to run and display. Activities are defined by the characteristic that they must have a View Tree; each View keeps a reference to the rendered Bitmap that represents its display. The problem with having non-static inner class in Activity expose the risk of having reference from inner class even after its lifetime, consequently, the entire layout hierarchy that Activity holds will be leaked.

2.3 Challenges

Traditional preventative approach focus on the stack disciplines to prevent the retention of unused objects. In extreme example, one can forbid dynamic memory allocation which forces all objects to be attached to stack frame, and will be guaranteed to be released after the end of the frame. Less restrictive approach, such as Realtime Specification for Java allows dynamic memory allocation but forbid object from referencing objects at deeper stack frame [2]. This limits the expressibility, and also introduce compatibility problem with standard libraries which restricts developer from general purpose development. On the other hand, detection approaches researched in previous studies that base on staleness heuristics and inference for object relationship [53] produce weak guarantee by either underestimate or overestimate the unused objects in the Java program. As an example discussed in Section 2.2, unused objects (garbage object) that merely serve as a cache in the hash table might not be identified as junk by staleness heuristics as the hash table rehashes its elements after the load factor reaches 0.7.

Although anti-patterns are studied, memory leak in Java program is still hard to detect and prevent. One reason is that, it is difficult to distinguish meaningful patterns in program's heap; further, implicit memory management adds layers of abstraction within highly optimized industrial JVM, which makes it difficult to comprehend the traits of memory leak bug. In next chapter, we articulate and discuss the core JVM technology that memory management details that are crucial in solving Java memory leak problem and increase the performance of Java applications

Chapter 3

Java Application Performance and Java Virtual Machine

In previous chapter, we explored the relation of memory leak and Java application performance. We presented the performance criterions, impact of memory leak to the application performance, and we have presented four types of Java memory leak examples derived from language design and application design. In this chapter, we look into the Java performance considerations from another perspective, from JVM internals. This is an important part to consider in addition to memory leak because JVM governs many of the important Java runtime work which is absent from other object oriented language; further, JVM also plays an important role in making Java highly portable, scalable and efficient. Last but not least, we integrate our runtime verifier into a JVM, therefore, understanding JVM is a necessity.

We start our discussion from Java Virtual Machine internals to introduce core parts of JVM; we then move onto Just-in-time optimization, which is a runtime optimization engine that drives Java code efficient on targeted platform; lastly, we investigate memory management scheme of JVM where memory allocation and garbage collection mode are studied with performance considerations.

3.1 JVM Internals

Each Java application runs inside an entity that is a runtime instance of some concrete implementation of the abstract specification; such a entity is Java Virtual Machine. In Java programming language, source code are not directly compiled into a target specific

native code; to improve the code mobility, it is first compiled to a intermediate level abstraction language called bytecode for a virtual machine that is running on the memory of target platform. JVM acts as a bridge between platform independent bytecode and the machine specific native code by compiling or interpreting the bytecode to machine specific instruction sets. With extra layer of abstraction, JVM also enforces type-safe reference casting, null reference checking, structured memory access and array bounds checking, which makes Java program secure.

As an abstract computing machine, JVM consists of Thread, Code Cache, Permanent Generation, Heap Area and their components to run Java application. Figure 3.1 below illustrates the internal structure of JVM.

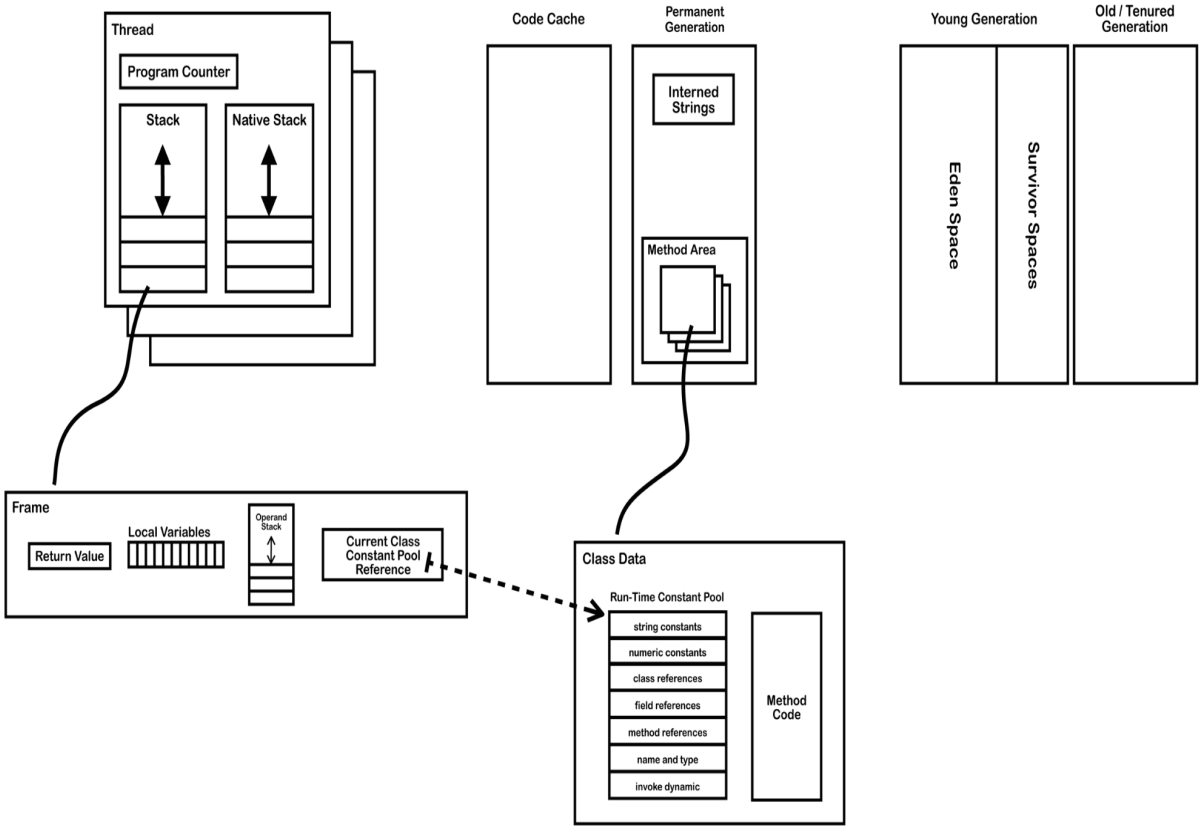

Figure 3.1: *JVM Internal Structure [48]*

Java Thread is a single unit of execution in a Java program. While maintaining Java Thread, JVM also governs JVM System Thread. JVM System Threads are the background threads that runs with the application main thread. In a HotSpot JVM, JVM System Thread includes VM Thread, Compiler Thread, GC Thread, Interrupt Thread [49]. Each thread has a Program Counter which records the instruction to run. Each thread also has its own stack that holds the frame for each method to be executed. Frame is a container that holds specific information associated to a method; it contains reference to runtime constant pool for class of the current method (Figure 3.1), local variables, operands and the return value. Frame is added to the stack upon method invocation and popped when method returns.

Code Cache belongs to non-heap memory space, and it holds native code compiled by Just-In-Time compiler during the JIT compilation process. The reason for having JIT compilation is to improve the runtime performance; JVM tries to find the frequently executed code block, and instead of doing interpretation every time it visits that code block, JIT compiler compiles it optimized machine code specifically for the target platform.

The Permanent Generation memory area is used to store internal metadata required by the JVM to execute a Java application. Permanent Generation contains interned Strings and Method Area. Method Area stores class information, such as Classloader Reference; Runtime class constant pool that holds method reference, field reference and attributes; field data and method data that has type, modifiers and attribute related information; and method code which holds method's bytecode, local variable table and exception table.

Heap area, which is the key focus under Java memory management (discussed in later section), is used to allocate new objects and arrays. Heap is divided into different part base on memory management strategies in Garbage Collection. For Generational Garbage collection scheme, that bases on the hypothesis that most of the objects die in early stage, heap is divided into *Young Generation* and *Old Generation*. Young generation contains objects that is newly allocated, and they are promoted to Old Generation after they survive enough times. Note that the benefit of having generation concept is that JVM can apply different garbage collection algorithm and at different frequency to achieve optimal application performance. For example, it is possible to apply simple Mark-and-Sweep collector at Young Generation frequently to recollect unreferenced objects so that new objects can be allocated in Eden space, while more thorough garbage collection is done less frequently in Old Generation to reduce the time for "stop-the-world" collection.

While running Java application, JVM also does important performance optimizations. The next section discusses a key optimization engine in JVM, Just-in-Time Compiler

3.2 Just-in-Time Compilation

In traditional compiled language, such as C/C++, code optimization is done during compilation process; optimized machine code is generated for host CPU which is aimed to run in the most efficient way. This brings in one limitation: it only runs on targeted platform. One notable aspect of JVM that increase Java application performance is the Just-in-Time compilation process; it can be seen as the second stage of Java compilation which try to balance the portability and performance of Java program. Java source code is first compiled into bytecode which is platform independent. Bytecode is stored in class file, and is an intermediate code, that needs to be verified, linked and interpreted into platform specific machine code. The interpretation process can be seen as a dictionary translation, from bytecode to machine code; there is no optimization done during the interpretation. Interpretation shows advantages when the method bytecode block is large and infrequently used, which is common in servelet application. However, with the overhead in branching, decoding and manipulating stacks, doing interpretation during every run of the frequently used method and code body is inefficient. To address this problem, most of the JVM, such as JRocket, Hotspot support JIT compilation, which is to determine the hotspot based on dynamic runtime information of a Java application, and compile that part of the code to be machine specific and highly optimized instruction set.

In a nutshell, JIT optimization includes inlining a method, instead of calling method on an instance of the object it copies the method to caller code to prevent any overhead; eliminate locks if monitor is not reachable from other threads; replace interface with direct method calls for method implemented only once to eliminate the overhead of calling virtual functions ; join adjacent synchronized blocks on the same object; and to eliminate the dead code.

3.3 Memory Management

In Java, memory management involves various components from the JVM, and also runtime information which makes it complicated to analyze and predict. JVM generally has five discrete memory areas: Text Resident Segment, Permanent Generation, Code Cache, Active Heap and Garbage Heap (Figure 3.2).

The Text Resident Segment memory is the part that contains operating system libraries as compiled, executable native code (ie. libjvm.so, libnio.so). These libraries are platform-specific, and is read-only so that it and cannot be modified by the JVM.

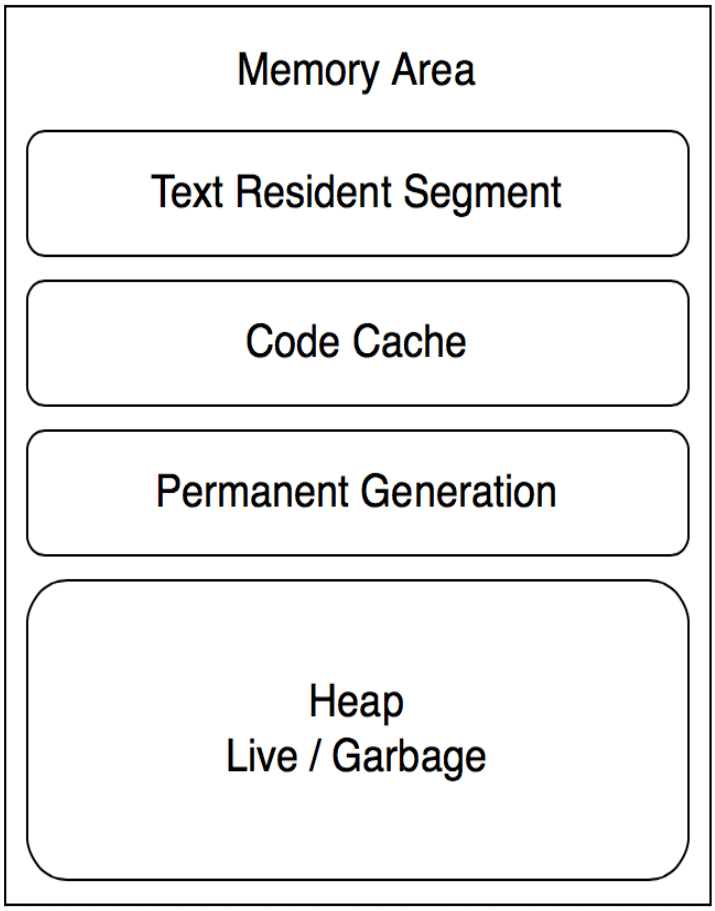

Figure 3.2: *JVM Memory Area*

The Permanent Generation memory area is used to store internal metadata required by the JVM to execute a Java application, such as classfile bytecode and class definitions. Since every time a class is loaded into the JVM, the JVM allocate some of its Permanent Generation memory area to store the necessary internal metadata and definitions for each loaded class, Permanent Generation memory size should be adjusted accordingly when a large number of classes is to be loaded (ie. application that uses many jars, big classpath).

The Code Cache memory area is the memory space used to store all the native code produced by the JIT compiler during the execution and compilation of the Java application in runtime. As being described in previous section, JIT is a dynamic compilation techniques to increase application performance, by compiling or recompiling the "hotspot" of the application methods to incrementally improve application performance.

The Java heap is the runtime data area where JVM allocates memory for all of the Java objects, both scalar and array used by Java application. The Java heap space is the most frequently tuned feature of a JVM, and is configured with the `-Xms` and `-Xmx` command line options. `-Xms` indicates the minimum heap size, and `-Xmx` means maximum heap size. The reason for having these two parameters is that the Java heap space is by default lazily allocated by the JVM as the application executes; therefore, it is dynamically growing. When a Java application is first started the Java heap space is initially very small, and then over time grows to occupy the full maximum reserved size specified with the `-Xms` and `-Xmx` command line arguments. Java heap can be divided into different parts, live heap and garbage heap, which as the name indicates, means the heap area for active objects, and the other heap for garbage collected objects. To improve Java application performance, heap layout can be also specified into generations which makes objects reclamation more efficient.

3.3.1 Memory Allocation

Whenever a class instance or array is created in a running Java application, new objects are allocated onto heap. The Java heap memory area comprises the space taken up by the currently live objects, objects which are dead but have not yet been garbage collected. The live-heap memory area is the portion of the Java heap space that contains the live (reachable) objects currently in active use by the Java application. As an application executes, the size of the live-heap memory area changes dynamically.

Stack is also a place where memory allocation can happen. When a new execution thread is created, the Java virtual machine creates a new Java stack for the thread. Java stack is used to store the thread's states in discrete frames. Stack frame has three parts:

local variables, operand stack, and frame data. The sizes of the local variables and operand stack are measured in Words, and their size are determined at compile time and included in the class file data for each method. Those statically allocated memory are bounded by the scope of stack frame, and they are not part of the automatic memory management scheme.

Each Java application has its own binding JVM, and there is only one heap in a JVM instance, which all threads share it. Consequently, there is no possibility of having more than one Java application compete for single heap resource. This indicates that memory management happens in each individual Java application, and can be tuned depends on each application's memory usage characteristics to attain improve application performance. Although JVM has instruction to allocate new memory on heap when object allocated, it does not directly free the memory for garbage objects. This is done by garbage collector.

3.3.2 Garbage Collection

While JVM is responsible for deciding whether and when to free memory, garbage collector's responsibility is to actually reclaim the memory occupied by an object that is no longer referenced by root or active object in the running Java application. JVM specification does not explicitly states that way memory should be freed. However, it is critical that finite heap space is used in a sustainable way so that there is memory space available for Java application to allocate new objects when running.

There are different algorithm in garbage collecting unused objects; each method has its own advantages and disadvantages, and all of them incur certain overhead to the executing application. Trade-off exists in a sense that either more computing resource is needed or application has to record lower throughput rate during garbage collection. For example, any variation of sequential garbage collection requires to "lock" heap space to be collected; this will prevent the Java application to keep on allocating new objects to that space (application has to wait.). This can be improved by doing "partial locking", to collect part of the heap at a time; however, this requires extra information to remember the part of the heap space being operated on, and monitor each part of the space to determine which part requires garbage collection. Further, having heap "divided" into separate smaller part for garbage collection requires more frequent garbage collection, consequently, overhead associated with starting the garbage collection and any safe-point check would be increased. Another way to reduce the amount of waiting time for sequential collection is to have multiple garbage collector with support from multiprocessor, and let them do "parallel collection". This will reduce the amount of time to lock the heap, but introduce hardware

requirements and also overhead associated for coordinating different garbage collector so unused objects are reclaimed.

Fragamentaion is also an important factor that garbage collection has to handle to reduce its impact to the performance of the Java application. The reason for fragmentation to occur is that objects that are spatially next to each other will not necessarily become unreachable at the same time. This means that heap space may become fragmented after a garbage collection, so that the heap become many small free spaces. This incurs problems when Java application tries to make allocation of large objects that do not fit into the small free space. In addition, those small free spaces on the heap, if they are smaller than the minimum thread local area size, they can not be used. The garbage collector discards them as "dark matter" until a future garbage collection frees enough space next to them to create a space large enough for a TLA [50]. This inefficient use of heap, if not dealt appropriately with garbage collection algorithm will result in reduced space available heap size for Java application and potentially cause out of memory error.

In this chapter, we analyzed JVM internal structure and its core runtime elements with respect to Java application performance; specifically we made correlation between the memory management with performance considerations, which leads to our solution and strategies discussed in next chapters to improve Java application performance.

Chapter 4

Object Lifetime Specifications

To detect the memory leak in Java application, we use an important concept called Object Lifetime. Object Lifetime is a term used to describe the time interval an object is alive in the running program. It is different from the term scope because scope defines the range in which variable is visible and accessible. Further, scope can be also influenced by the access modifier defined in the language, while Object Lifetime does not. Object Lifetime can be thought of as a flexible way of defining the usability and life of a dynamic object[20].

In Object Oriented language there are four type of objects that makes up an application program. *Global Object*, *Automatic Object*, *Static Object* and *Dynamic Object*. Global Objects, also referred to as *External Object* are objects that are persistent and visible through out the lifetime of the program, and its identities and attribute remain constant throughout the program. Automatic Objects, also know as local object, are persistent only in the local scope where it is instantiated. Consequently, Automatic Objects are automatically destroyed when that particular local scope is about to exit. Static Objects' lifetime is bounded by its containing class's lifetime; it is shared among all the instance of the class where Static Object is declared. Last but most important to our research in this thesis, is the Dynamic Objects. Dynamic Object's lifetime can be controlled differently in different language; however, one thing in common is that there is no static binding frame/scope to deallocate it, unless otherwise specified by the user.

In this chapter, we present one of the key ideas our runtime verifier uses for detecting unused objects in Java heap. First, we introduces the concept of *Object Lifetime*. Next, we discuss about the way to associate objects to a "relatively" fixed stack frame using *Group*. We then discuss how this is realized in the context of Java programming language with a feature called *Annotation*. Towards the end of this chapter, we present an example

program code to illustrate how Object Lifetime is embedded into a real Java program.

4.1 Object Lifetime

Object Lifetime indicates the time between when an object is created and destroyed. This duration in Java is "fuzzy" especially towards the end because object reclamation can be influenced by Java runtime, and decision as part of memory management. Further, conservative approach in determining the root set can also retain dead object alive during minor collection especially in incremental garbage collection.

In order for an object to be created, Java runtime first has to locate the .class file on the disk, and load it into main memory. An object is then created from the class with new keyword. To initialize, Java allocates memory space for the object, and set a reference to that object so that the Java runtime can make reference of it. Finally, Java calls the constructor, which initializes the fields and complete other tasks for the object to be usable. The object will now allow access to its method and state variable following specified access specifier, and the duration between first-use time and last-use time will be considered as the active time of that object. An object should become garbage collectable when it does not have any influence on the output of the program; namely, there is no more reference to that object. Garbage collection within Java runtime will then remove the internal reference to that object, and remove the junk object from the main memory so that memory becomes available again.

In general, knowing when an object becomes garbage is undecidable. Runchiman used staleness, which is the time between object's last use and the time when it becomes garbage to approximate the "junkness" of an object [53]. However, this approximation is flawed with overestimation or underestimation, therefore producing weak guarantee. Therefore, it can be seen that determining the memory leak based on predicting the Object Lifetime still produce unreliable results. Further information has to be obtained from profiling, which requires either modifying JVM or dynamically attaching extension, in both cases, it will result in certain amount of overhead on running Java application, therefore, affecting the performance.

Thus, in this research, we define the object lifetime to be strictly based on when it is created and usable, and the time when there is no more reference to it, and is collected by garbage collector. As we will discuss in next section, we explicitly include the Object Lifetime Specification in Java program with developer written annotations instead of applying heuristics in approximating or predicting the object's lifetime.

4.2 Object Lifetime Specification and Annotation

In previous section, we introduced the concept of Object Lifetime, and briefly discussed the limitation of relying on heuristics to predicatively identify junk objects. In this thesis, we use a methodology to explicitly state the Object Lifetime for objects, and use this as the property to verify at runtime; in other words, instead of predicting the Object Lifetime through analyzing profiling data to detect unused junk objects, we specify the Object Lifetime as a property of objects a head of the time based on user knowledge and software design requirements, and check the satisfiability of this property during runtime. We say this is the runtime verification of Object Lifetime Specification. Since the Object Lifetime information is specified by user, it needs to happen at Java source code level. The difficulty comes in as there needs to be a way to instrument the user code, and carry over the Object Lifetime information to runtime without perturbing the program execution and its internal state. Below, we discuss a Java technology that helps us solve this problem: Java Annotation.

Java Annotation is introduced in JDK1.5, and it is a form of metadata that provide information about a program that is not a part of the program itself [36]. Being a meta data, annotation dose not interfere with normal program execution; however, other components can use this meta data information to make logical decisions, such as class consistency check, parameter validity check or compiler instructions. For instance, when dealing with Java serialization (write an object into a stream to transport over a network, and later being rebuilt), we traditionally use transient keyword to say a field is not serializable; however, with the functionality of annotation, we have a generic way of adding information to class, method or field; these information can be processed by user, compiler and most importantly, the Java runtime. Java language also has standard annotations, such as *@Override* to tell the compiler that a method overrides the corresponding methods in it superclass or an interface. There are other standard annotations, such as *@Deprecated*, *@SurpresWarnings*, *@SafeVarargs* [44] that can be used to provide information to compiler, and validity check at runtime. Annotation provides a wide range of usage, summarized in the following three categories[36]:

Information for the compiler: Information contained in annotation can be used by the compiler to detect error or suppress warning

Compile-time and deployment-time processing Annotation information is used by software tools to generate XML files and code

Runtime processing Available for runtime processing

Java Object Lifetime is runtime information about objects; therefore, we are particu-

larly interested in the third usage of Java annotation. We see annotation's main powerfulness as to be able to be carried over to runtime in addition to compilation level processing. To persist the annotation information, the *Retention Policy* should be specified to inform the Java compiler and JVM that, the annotation should be available by reflection at runtime.

With these features from Java Annotation, our goal is to build a runtime verifier that lets the developer to embed Object Lifetime information at source code level as a specification, and we verify this specification against actual JVM events to determine the satisfiability of Object Lifetime Specification at runtime. If the Object Lifetime Specification is satisfied, we are sound that the objects' lifetime is "controlled" in the way user wanted (specified through annotation), and agree with the actual JVM memory management on the objects; therefore, there is no memory leak. Otherwise, objects' lifetime specification is not satisfied, and it indicates there is a discrepancy between what user's belief on objects' lifetime with actual memory management tasks happened inside the JVM. We illustrate this idea with an example in next section.

4.3 Example

As discussed in previous section, we enable the user to embed Object Lifetime specification at source code level through annotation, and our tool does runtime verification on the satisfiability of the property specified. To convert Object Lifetime to a specifiable property, we note the beginning and the end of the Object Lifetime by introducing custom annotation *@Create* and *@Release*. The usage is shown in Figure 4.1 below.

```
@Create("GroupA")
public void allocate_a(){
    node = new Tree()
    objOne = new Object();
}
```

```
@Release("GroupA")
public void deallocate_a(){
    node = null;
    //objOne = null; //Junk!
}
```

Figure 4.1: *Object Lifetime Specification with Annotation*

The example provided in the figure illustrates a simple code snippet of two method annotated by custom annotation, @Create and @Release. @Create indicates the start of

the Object Lifetime and @Release indicates the end time of Object Lifetime of associated objects. Since the annotation is done on method level, it governs all objects declared in the method. For instance, *node*, and *objOne* is instance variables that is instantiated in *allocate_a* method; therefore, user uses @Create("GroupA") to indicate the start of the Object Lifetime for objects referenced by node and objOne; it has same specification influence to both node and objOne. To indicate this relation between annotation and objects declared inside of the annotated method, we use the concept *Group* to associate objects to a group [6]. In the example above, node and objOne is associated to GroupA. By associating objects to group when object is created, we effectively assign a *"tag"* to objects with same Object Lifetime. This becomes useful when we specify the end of Object Lifetime. When the user specify the end of Object Lifetime to a group of objects, @Release annotation is used. In the example code snippet, @Release("GroupA") is annotated on *deallocate_a*; by doing so, user is indicating that any objects whose Object Lifetime is associated with GroupA should be reclaimed and consequently eligible for garbage collection. Observing into the content of deallocate_a method, node is assigned to null while objOne is not; therefore objects referenced by objOne is still reachable, and will not be collected in the next garbage collection cycle. Consequently, the user annotated Object Lifetime Specification is not satisfied, and we report this as junk object retention to the user for his further inspection.

Chapter 5

Runtime Verifier Design

In previous chapters, we have discussed Java performance considerations in terms of JVM runtime and Memory Leak. We have defined unused object retention in Java application, which in the long run constitutes memory leak because of fixed heap size. We have also studied four types of Java memory leak, looking from language design and code design perspectives to show how unused junk objects can manifest itself in the code without being noticed. We have researched into JVM internal components to understand better about JVM memory management scheme and how our verifier can work with JVM to improve the Java memory management so that the Java application performance can be increased. We then identified our main idea to be used for runtime verification, Object Lifetime Specification. In this chapter, we proceed our discussion into the design and implementation details of our runtime verifier, and shows how our tool can be applied, and work with JVM to detect unintentionally retended objects.

5.1 Concept

Solving memory leak in terms of unused object retention is not an easy task for two reasons. Firstly, real Java application allocates many objects on heap at runtime, thus, heap usage is dynamic and unpredictable; furthermore, since heap activity is managed by Java runtime, it is compulsory for our runtime verifier to work with it with minimal perturbance to the JVM, increasing the requirements for interfacing and modularization. Second, since unused object retention comes from design error, which can be user's sheer negligence, there is no direct and precise template to determine memory leak.

With all those factors in considerations. We propose the Java Virtual Machine-assisted collaborative junk object detection with Object Lifetime Specification. Conceptual-wise, we move forward from predictive approach of using Object Lifetime to allowing the user to specify Object Lifetime; user in this context is the developer or performance testing engineer who have exclusive access to the source code of the Java application as well as the design documentations. As the unused object retention in Java program results from design error, we give users the ability to do the Object Lifetime specification. We believe that user of our tool is the one that is most familiar with the design and semantics of his Java program; he understands the design factors/goals, requirements, and performance considerations which is not accessible to the public nor JVM runtime. Consequently, he has the direct knowledge of Object Lifetime information of the interesting objects in his code. Note that, we are using the term "interesting object", namely because, we give the user the freedom to choose which objects are of greater significance and should be runtime verified for memory leak. We foresee the amount of overhead associated to our verification tool, and we provide guidance but let the user to choose the amount of annotation (specification), he wish to leverage on his code. Our tool will extract the Object Lifetime specification from the user annotations in Java source code, and observe the runtime JVM events to verify if the specifications have been met or not. In the case of property violation, we report this to user through log.

Based on the design concepts, we delineate the following top-level design goal for our tool as part of the architectural design considerations and detailed component requirements discussed in the next section:

1. **Clean and Robust:** We want our tool to be light-weight based on lean design principle, and fault tolerant and can handle exceptions that occur inside of our runtime verifier.

2. **Minimal Perturbation to JVM and User Program:** Program under the test is running within JVM, we do not change or alter the internal state neither JVM nor the Java program following isolation principle [38].

3. **Verification Accuracy:**Our runtime verifier should correctly process user specifications, and can verify Object Lifetime information with actual JVM event.

4. **Performance:** Overhead is expected, but should be controlled to have minimal influence to the application performance.

5. **Effective Information Report:**We imagine our research tool to be used in both Prototype Verification Test and Engineering Verification Test. Therefore, the infor-

mation we report is crucial to the user to determine the source of memory leak, and must be precise and adequate (not too much and too less)

In the next section, we start our discussions on the design architecture and main components of our runtime verifier tool. We leverage our discussion closely to a research metacircular Java Virtual Machine from Oracle Cooperation of which our runtime verifier is to be integrated on for experiments and performance evaluations.

5.2 Design Architecture

The design of the runtime verifier is modularized and each components is designed to be able to communicated with each other. Specific components are also interfaced with JVM to observe VM events for verification. The design architecture is derived from the runtime process flow. This process results from normal execution flow of a Java program running on a JVM, and the runtime verifier will act as an extension to the JVM , without perturbing the process. The flow chart provided in next page illustrates this (Figure 5.1).

As shown in the Figure 5.1, the lifetime of runtime verifier is started when the JVM starts running. While the Java application under the test is running, our runtime verifier observes for the JVM event advice callbacks; we are observing VM events, when method is invoked and when object is created. When our runtime verifier is notified of method invocations, we check for if an annotation exists on the invoked method. If the custom annotation `Create` is annotated on the invoked method, we extract the annotation value from it (ie. Group information), and store it into runtime verifier's data structure, which will be later retrieved and written into Java Object Header for objects created inside of this method. If annotation `Release` exists, we record the annotation value into a static data structure in our runtime verifier, which keeps track of the groups of objects to be released. To meet the goal of robustness, we deal invalid annotation by logging the warning message, and does no operation on it; we do not throw exception nor use stdout to inform user about this at runtime. While user program is executing, and our runtime verifier process Object Lifetime Specification, GC thread might become active to do garbage collection. When GC thread has judged an object as active object (survived), we read from the object's header field, and check if this object belongs to the group that needs to be released. If so, we report to the user about this contradiction between user specified object lifetime to the actual events happened during JVM memory management.

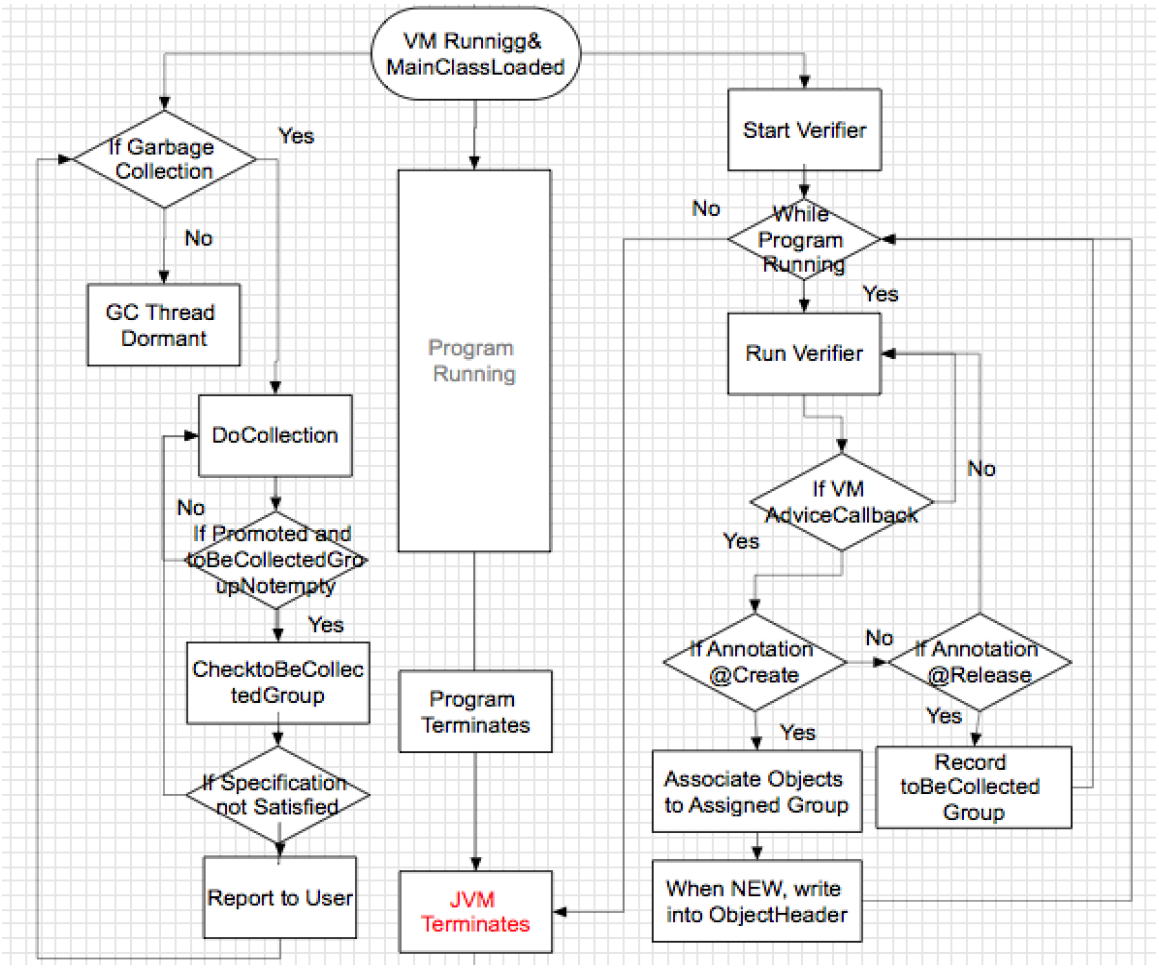

Figure 5.1: *Runtime Verifier Control Flow*

5.2.1 Design Components

In analyzing the above mentioned control flow, we propose the following five major components as part of our runtime verifier architecture. We summarize these components below.

1. **VerifierMain:** VerifierMain is the central module of our runtime verifier. It should:
   - determine lifetime of runtime verifier
   - create static data structures to be accessible for other modules and JVM
   - provide interfaces to facilitate communication between submodules
   - handle and recover from erroneous state
2. **EventHandler:** EventHandler implements *Observer Pattern* [21] for JVM events. It should handle:
   - new object/object array creation advising
   - write annotation value to object header when object is created
   - method invocation for virtual/static/special methods
   - notice before and after garbage collection
3. **AnnotationExtractor:**AnnotationExtractor retrieves annotation value if there is annotation on a method. It should manage:
   - annotation interface to supply custom annotations
   - @Create annotation and @Release annotation
   - convert annotation value into unique and deterministic long value
   - store unique long val into VerifierMain's static data structure
4. **JunkReport:** JunkReport is used to report junk object information to the user at runtime. It should:
   - log into a user specified log file; display via user interface of the source of junk objects

Note that since our verifier acts as an extension to the JVM, JVM also have to be modified accordingly to meet the design goal. For instance, Java runtime scheme as to be modified to instantiate the runtime verifier; object layout has to be modified to set extra Wort field to record objects' associated group information; garbage collection also needs to be intercepted with an extra check on active objects to see if it violates the user specification of Object Lifetime. These modifications are discussed in depth in the following sections.

To validate the concept and communication between the modules described above, a test design architecture is created and tested with mock objects; the proposed design architecture is tested for its conceptual validity. The garbage collection process from Java runtime, and annotation extraction process is substituted by mock objects in this Design Verification Test. A UML diagram of the proposed design architecture is provided in the Figure 5.2 to show the basic interaction between the modules and JVM.

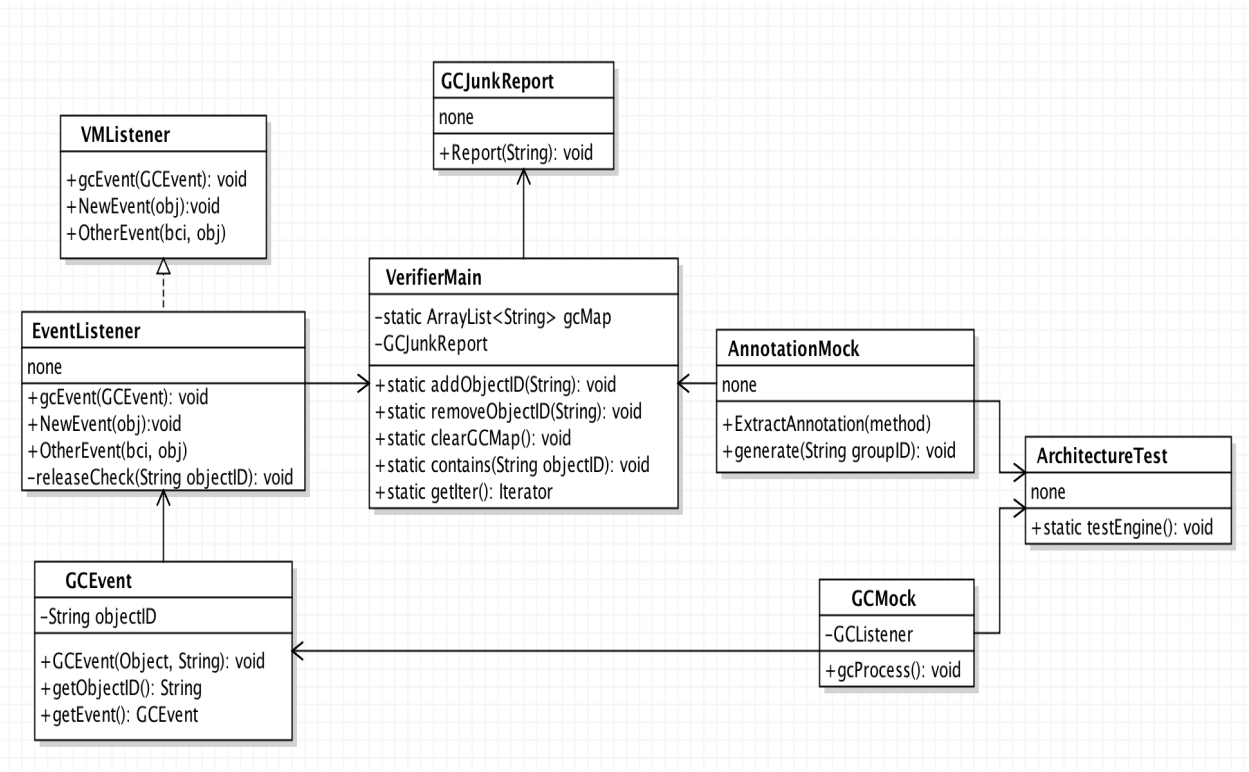

Figure 5.2: *UML Diagram for Verifier Design Architecture*

5.3 Implementing Runtime Verifier with Maxine VM

Maxine Research Virtual Machine is a open source meta-circular JVM created by Oracle Laboratory. In contrast to industrial VM which is written in multiple languages, Maxine VM is designed for and written in the Java Programming Language which is aimed to provide simpler research and experimental platform to the researchers. As being articulated in the Maxine project wiki, Maxine VM is designed with an emphasis on leveraging componentized design, and code reuse to achieve flexibility, configurability, and productivity for academic and industrial virtual machine research [28]. Since Maxine VM is a meta-circular JVM, its design integrates closely with Oracle's standard Java Development Kit packages, and exploits the advanced language features such as annotations, static imports, and generics, which makes Maxine VM suitable to be studied, and modified with support from current JDK. Figure 5.3 shows an comparison between conventional VM to meta-circular VM[32]. From the figure, it can be seen that conventional VM establish itself on top of C/C++ language and does not use Java language; the interfacing with JDK is also low compared to meta-circular VM which reside itself in Java Language and JDK. This meta-circularity allows the JVM itself to benefit from the features it is intended to provide, and further maintain a co-evolution of features in the itself and the application it is running.

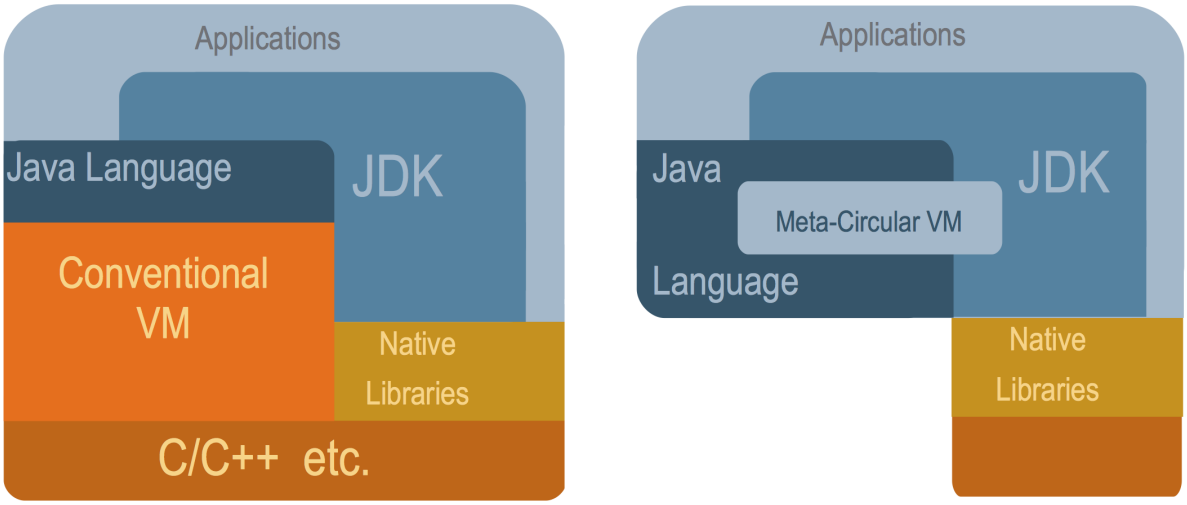

Figure 5.3: *Conventional VM and Metacircular VM [32]*

Although Maxine VM is designed to facilitate JVM research, it is intended to generate comparable performance to existing industrial JVM. Maxine includes C1X optimizing JIT compiler, which is an refined version of C1compiler from HotSpot JVM; C1X retains most of C1's original design, with some front-end improvements, simplifications, and cleanups while porting to Java[30]. It should be noted that Maxine VM does not have interpreter; methods are first compiled with a lightweight baseline compiler, which translates Java bytecode into preassembled native instructions. Then C1X optimizing compiler is triggered to further optimize methods that are frequently invoked. During JIT optimization, C1X does optimizations, such as inlining, constant-folding, strength reduction, local value numbering, and load elimination. In addition to optimized JIT compiler, Maxine also includes Generational Heap scheme as its memory management strategy. Generational Heap scheme enables aging mechanisms which makes full garbage collection less frequent, and therefore improving the throughput of the application performance.

The rationale of choosing Maxine VM is that, first, it is written in Java, and includes more interface and experimental extension to allow external modification compared to industrial VM whose implementation is not intended to be modified due to the optimizations made for high performance. Second, Maxine VM leverages on JDK technologies which means our runtime verifier can user Java language features to realize our design goals; moreover, support of JDK enables us to conduct experiments on real Java open source project with Maxine VM. Although not elaborated in this research, Maxine VM has a VM level debug assistance tool, called Maxine inspector which helps user to understand VM level operations in terms of memory management.

During initial experimentation with Maxine VM, it is observed that Maxine VM requires more memory in running a program compared to standard openJDK; however, the running speed is comparable to highly tuned industrial JVM. The increased memory requirement should mainly come from usage of the internal components of Maxine VM than the operations associated with the objects from the Java program running. Our investigation to increase the Java program size (more objects and layers of abstractions) did not significantly increase the total memory consumed.

5.3.1 Runtime Verifier Interfacing

The first step of our implementation starts from researching where in the hierarchy, our runtime verifier should be inserted in. From the design goal requirement, we want our verifier to be clean and not interfering with the internal state of JVM nor the program execution of user program under the test. In this sense, our runtime verifier needs to be

well isolated from both JVM and user program. Further, our runtime verifier needs to have lifetime longer than the user program to be able to capture all annotated specifications but not more than the JVM itself. We start our research into possible Java technologies that offers us this capability as well as the extension mechanism Maxine Research VM provides to us. The options studied and evaluated are described below.

Adding Extension to BootImage

Maxine VM is a meta-circular VM; therefore starting an instance of the Maxine VM requires the use of a previously constructed boot image through bootstrapping. The boot image is a near-executable binary VM image that includes an initially populated heap and compiled code for the target platform[27]; heap is populated with class metadata and objects implementing the VM itself, and the code cache is populated with code produced by the optimizing compiler. A small C program from the substrate, maps the boot image into memory before calling the VM entry point via an indirect call [56]. This effectively hands control over to Java code that implements the Maxine VM. Bootimage generation process is intended to have all the classes and methods that are needed to bootstrap the VM to the point where it can dynamically load further classes from the file system and compile methods for execution. Since our runtime verifier works closely with Maxine VM's core components, such as memory management and runtime information access, it makes sense that our verifier be included as part of the boot image. From our study into Maxine VM, we determined that Maxine supports two modes of boot image extension: *Static* and *Dynamic Extension* for the VM bootimage[27]. Static Extension, as the name suggests, adds extra classes to the boot image during boot image generation process; this is realized through setting boot image argument shown below:

```
mx --Jp @-Dmax.vma.handler.class=<Package>.ExtensionClass image
```

On the other hand, Dynamic Extension loads external class files when running Java application. A Dynamic Extension needs to be packaged in a jar file, and loaded by the VM classloader at runtime, just before the main class of the user program is loaded. A special Manifest file needs to be defined to specify the VMExtension-Class attribute that specifies the class that is the entry point to the extension[26]. To load the Dynamic Extension, the following flag has to be set when running Java program:

```
mx vm -vmextension:acme_vmextension.jar=args TestApplication
```

While considering extension mechanism offered from the Maxine VM, we also studied other Java technology that is common to all JVM; the benefit of this is we can have our verifier work with any JVM through standard technology.

Adding Extension as Java Agent Another alternative way of attaching our runtime verifier is through the use of Java agent. Java agent is introduced in Java5, and is a self contained component through which application classes can pass in byte code instructions in the form of byte arrays; this enables Java agent to apply dynamic instrumentation to the user code[34]. One of the advantage Java agent offers is that it is relatively light weighted, and has standard support from JVM. However, a major problem we find in Java agent is that, it does not allow communication between the agent to the JVM which we require as part of our runtime verifier's functional requirements; we need to be able to interact with VM events and Garbage collection which we find it impossible with Java agent. To determine the precise reason for not being able to reference VM internal class, we continued our investigation into Java agent, and determined that, although Java agent shows similarity with Maxine VM's Dynamic Extension mechanism, Java agents is still considered application extensions, and it is loaded by the system classloader, whereas a VM extension is loaded by the VM classloader. Furthermore, it does not have the ability to reference the VM classes in the boot image while VM extension classes does.

We evaluated (shown in Table 5.1) above mentioned three options of adding our runtime verifier with respect to the functional requirements and performance considerations of VerifierMain identified in the previous chapter, and determined that Static Extension is the best choice for our purpose. Maxine VM's Static Extension mechanism provides a gateway to add a set of classes to the bootimage, and allows extension class to reference Maxine VM classes and its features. Although building extension into boot image results in increase in spatial overhead, we determined that compared to the existing classes added as part of boot image for VM startup operation, this size is negligible; through experiment, we found that the bootimage size on 64 bit machine is roughly 85MB, and our runtime verifier extension classes has total size less than 1MB. Further, the runtime overhead in loading extension class during runtime as of the case in Dynamic Extension is also eliminated as the extension class is build into the bootimage, and is part of the running Maxine VM when main class of the Java program is loaded.

Extension Options	Interact with JVM	Space Overhead	Runtime Overhead
Static Extension	Possible	Low-Mid	Low
Dynamic Extension	Possible	Low	Low-Mid
Java Agent	Not Possible	Low	Low-Mid

Table 5.1: Evaluation of Runtime Verifier Extension Options

5.3.2 Event Advice Handling

JVM event handling is a crucial part in our runtime verifier implementation, because it is the main venue that we obtain user annotated specification through JVM. However, before being able to observe, handle and record Object Lifetime specifications, we need to have a running verifier instance. To ensure the verifier to have correct life span described section 5.3.1, we engaged in a series of debug trace analysis to determine the appropriate entry point for Maxine VM's initialization sequence, particularly focusing on Maxine VM's `JavaRunScheme`. To determine Maxine VM's startup sequence, we inserted log statement in VMConfig class, and `RunScheme` class from `com.sun.max.vm.run` package. Based on the analysis of the startup logs (provided in the Appendix A.1), we found that `JavaRunScheme` is invoked by the VM after it has started basic services, and is ready to set up and run Java program; `JavaRunScheme` from `com.sun.max.vm.run.java`, starts up normal JDK services and then loads, and runs the user-specified Java main class. Therefore, we instantiate our runtime verifier right after the conditional check when the phase of `JavaRunScheme` is registered as RUNNING.

After instantiating our runtime verifier, we move forward to start observing the JVM events. The goal of observing VM event is to retrieve Object Lifetime Specification from user code, and register this information into each object. To achieve this goal, we observe the following VM events:

1. **New Object Creation:** We write associated group information into object header when the object is created
2. **Method Invocation:** When a method is invoked, we check if annotation exists
    - If @Create custom annotation is found, we store the annotation value (ie. Group Information)
    - If @Release custom annotation is found, we store the to be released group information into VerifierMain's static data structure
3. **Before Method Entry and Exit:** We update the stored annotation content just before the annotated method exits so that we do not write group information into objects declared in a method that is not annotated. Further, this allows us to handle nested method with annotated with different Group Information

For debugging purpose, we also observe before and after garbage collection process.

Maxine VM provides a Virtual Machine-level Analysis(VMA) interface to facilitate the VM event advice handling. Maxine VMA is an experimental extension to support analysis of the code executing on the virtual machine. The analysis is implemented primarily by advising the execution of the bytecodes. Since the translation of language-level constructs to bytecodes is well-defined by T1X template-based compiler (details provided in the Appendix A.2), language-level advising could be achieved with a separate layer of mapping [45]. VMA shares some similarity with the JVMTI API, most notably the method entry, exit and field watch capabilities and some of the runtime advice[29]. Maxine VMA extension project is currently implemented in the `com.oracle.max.vm.ext.vma` and `com.oracle.max.vma.tools` package. In order to enable the advice mechanism from VMA, a custom VMA-enabled image has to be built with with:

```
mx image @vma-t1x
```

This instructs Maxine VM to build bootimage to include custom VMA schemes, and use VMA T1X compiler to dynamically compile loaded class with T1X template. The interface that defines the supported VM event advice callback is `BytecodeAdvice` class from `com.oracle.max.vm.ext.vma` package. Another useful abstract class that bridges between user defined advice handler to `BytecodeAdvice` is `VMAdviceHandler` abstract class that extends `RuntimeAdvice` which subclass `BytecodeAdvice`. `VMAdviceHandler` abstract class contains a method, `initialize(MaxineVM.Phase phase)` which must be overridden by the subclass that handles generated advice calls from the VM. We implemented our own VM event advice handlers by subclassing `VMAdviceHandler` with respect to the VM events we want to observe, which is described earlier in this section. The handler we implemented is summarized in the Table 5.2 below.

Advice Supported from VMA	**Event to Observe**
adviseAfterNew(int bci, Object obj)	New Object Creation
adviseBeforeInvokeVirtual(bci, object, methodActor)	Method Invocation
adviseBeforeInvokeStatic(bci, object, methodActor)	Method Invocation
adviseBeforeReturn(int bci, Object obj)	Before Method Exit
adviseBeforeGC()	Before GC Process
adviseAfterGC()	After GC Process

Table 5.2: Event Advice Handlers

This leads to our discussion on how we are processing user annotated Object Lifetime Specification at runtime, and associate objects instantiated in the same method into the same group based on annotated Group Information.

5.3.3 Runtime Annotation Processing

Runtime annotation processing is an important part of our runtime verifier design that extracts the Object Lifetime specification from the user program. As we described earlier, we give user the ability to embed Object Lifetime Specification into method level annotations (Figure 4.1). Currently, we support two custom annotations, *Create* and *Release*. `Create` custom annotation allows user to associate objects created in the method into a group that has same Object Lifetime, and indicates the start point of the Object Lifetime; while `Release` custom annotation allows the user to specify the end time of a collection of objects' lifetime.

Java Annotation is a form of metadata that provides information about a program that is not a part of the program itself [36]. In order to obtain the custom annotation that the user writes in his Java Program at runtime, we use *Java Reflection* API to retrieve declared annotations from the method handle. Specifically, to retrieve the Object Lifetime Specification, we reflectively check wheather a method is annotated with annotation; if annotation exists, we retrieve the value of annotation based on the annotation type (`Create` and `Release`). From the VMA `adviseBeforeInvokeVirtual` callback, we get three parameters which we can use: `int bci, Object obj, MethodActor method`. Of three passed-in parameters, we are interested in the last one, *MethodActor* for our runtime annotation extraction process. `MethodActor` is an Maxine VM internal representation of Java language entities[22], and it implements the entity's runtime behavior; `MethodActor` can be seen as a mirror object of the actual method with enhanced reflection. The `MethodActor` becomes the entry point for reflection operation, which allows us to introspect the presence of annotation and its value.

After writing isolated test cases, and validated the methodology in retrieving annotation at runtime as being discussed above, we inserted our code for runtime annotation extraction using the API methods offered in `java.lang.reflect.Method`. We used:

- `isAnnotationPresent(Create.class)`: check if annotation `Create` is associated with class methods; used differently from `isAnnotation()`: which just check if the tested object represent an annotation type.

- `getDeclaredAnnotations()`: if annotation is present, we obtain all annotations that are directly present on the method. Note that this method ignores inherited annotations.

- `getAnnotation(Create.class)`: returns the annotation for the specified type. If the annotation is not present, return null value

When testing the runtime annotation processing in Maxine VM, we noticed that we were unable to retrieve the annotation because the `isAnnotationPresent(Create.class)` check fails with respect to the type of annotation we are providing to the user. We investigated this issue from different perspectives in `MethodActor` class capability and annotation retention policy. From offline testing, we are certain the way we are retrieving the annotation from method handle is correct. To determine if we are getting the correct `Create` class from reflection, we inspected into the class information by inserting debug statement, and compared the hash code of the annotation object that we obtained from `getDeclaredAnnotations` with hash code of `Create.class`. The result shows that two classes have different hash code. We then proceeded to inspect the ClassLoader instance that loaded the `Create.class` and the "same" class retrieved through annotation, and determined that they are loaded by different ClassLoader: `VMClassLoader` and `AppClassLoader`. To solve this issue, we studied the reason why the entity is loaded by different classloaders. Through our investigation, we found that, for meta-circular VM, any class that might be written into the boot image must first be loaded by the boot image generator, and the classloading can happen in two ways. First, the class that is used by the user applications is loaded by the normal host classloading mechanism using the system classloader. On the other hand, the bootimage generator explicitly loads classes that might be written into the boot image by scanning the class path and searching for sub-packages of `com.sun.max.config` that contains a class named Package which subclasses `com.sun.max.config.BootImagePackage`. This second set of classes is loaded by a special classloader, `com.sun.max.vm.hosted.HostedVMClassLoader`, that performs additional actions, such as creating the Maxine representation of classes used at runtime. This explains the reason why our custom annotation class is loaded by `VMClassLoader` and the reflectively retrieved annotation class is loaded by `AppClassLoader`. We proposed two solutions to this problem: reload class using `VMClassLoader` or do string level comparison instead of class entity comparison. We evaluated these two solutions with respect to performance and implementation complexity, and decided to choose the later. The rationale is that classload operation is managed by Maxine VM hosted internals, and forcing classload involves going through layers of abstraction; this not only adds overhead to the performance by dispatching method calls but also poses potential unknown exception problem, because

we are not fully following the VM internal procedure in loading the class. We are aware that there is overhead associated with string comparison, but we prioritize our goal of not perturbing the internal state of JVM and also want to increase the robustness, Following the revised implementation, we successfully retrieved the Object Lifetime Specification from the annotated methods for both `Create` and `Release` custom annotations.

5.3.4 Recording Object Lifetime Specification

After retrieving the Object Lifetime Specification, we need to associate this analysis-specific information with an object, so that during the Garbage Collection phase, we can retrieve this information to do specification verification. To do so, we have investigated different technologies. We first studied the bytecode rewriting technology which is to modify the objects in bytecode level. Specifically, we studied the approach that uses ASM bytecode manipulation framework to modify the bytecode stream. However, based our design goal, we want to follow isolated design principle where we do not intercept user code (keep minimal). Further, using ASM framework requires redirection the bytecode stream into ASM interface library first, and this indirection would cause performance decrease. Interfacing ASM with object layout also poses problems because there is no object header in bytecode level. We also studied the approach of using a map to associate objects reference with its Lifetime information. This approach gives us precise information associated to each object; however, it introduces significant space overhead in the case of large program. Furthermore, it perturbs the behavior of the garbage collector by keeping the object reachable from the map. Although weak reference can be used to mitigate the effect of preventing garbage collecting a unneeded objects, but adds overhead to the memory management on the heap. After investigating possible approaches, we decided to focus on the approach of adding Object Lifetime information to Java object object header to see if there is any innovative approach we can seek after. Since object layout definition can be JVM specific, we studied into the object representation in Maxine VM.

In Maxine VM, object layout is the lowest level details of memory layout of objects with respect to headers, contents, and pointers[43]; VM's specific layout scheme is configured during boot image creation process through the use of `layout` flag. In Maxine VM, there are three object layout scheme, OHM, HOM and XOHM.

1. `OhmLayoutScheme`: Ohm object layout scheme is described as Origin-Header-Mixed, and is implemented by class OhmLayoutScheme in package `com.sun.max.vm.layout.ohm`[23]. The OHM layout packs tuple objects for minimal space consumption, observing alignment restrictions.

2. `HomLayoutScheme`: The Hom object layout scheme, described as Header-Origin-Mixed, is implemented by class HomLayoutScheme in the package `com.sun.max.vm.layout.hom`. This layout enables more optimized code on SPARC for accessing array elements smaller than a word [24].

3. `XohmLayoutScheme`: Xohm layout is an extended Ohm layout that includes additional Words in the header field to store analysis-specific state of an object[29].

To record Object Lifetime Specification into an object, we find Xohm useful in our implementation. Although using Xohm layout scheme increases the header size of an object, and might incur more frequent garbage collection due to increased object size, it gives us the ability to add Object Lifetime information in an isolated way where neither user nor the JVM uses this information in transitioning from one state to another. Support of Xohm layout is provided by `com.oracle.max.vm.layout.xohm` package where interface is defined in `ObjectState` class. We modified the `XohmGeneralLayuot` class, and also provided concrete implementation of `ObjectState` in `VerifierObjectState` class to enable the mechanism of writing and reading Object Lifetime information into the object header field. Note that Object Lifetime specification is passed in as group information when object is created. The representation of group information is by `String` type, where header field expect *Word*, an unboxed type to represent internal memory by the Maxine VM. We investigated different mapping strategies from `String` to `Word` while being attentive to the spatial and memory overhead. We finally added a mechanism to translate `String` into an unique and deterministic `Long` in our annotation extraction engine before writing into the header field. It should be noted that we use the word "deterministic", because group information with same string laterals retrieved at different stage of the verification would construct new `String` object, and traditional hash code identification will result in non-consistent `Long` representation. Another implementation strategy we took in recoding Object Lifetime Specification is considering nested method calls. In order to control the propagation of Object Lifetime Specification, we observe the stack trace element to update the stored group ID obtained from annotation.

5.3.5 Specification Verification

In previous sections, we described our implementation approach in associating Object Lifetime Specfication to each object. In this section, we discuss the methodologies we used to verify the annotated specifications with actual VM event.

As being discussed in Chapter 4, we verify user annotated Object Lifetime Specification with the actual VM event that takes place. Particularly, we intercept the garbage collection process which is an important part of JVM memory management. As a brief review, user can specify the start and end of Object Lifetime through custom annotation `@Create("Group")` and `@Release("Group")`, where "Group" is a convenient and flexible way to let user to associate objects with similar lifetime into a collection. Once `Release` custom annotation is retrieved, we immediately insert the `long` representation of the group information into a static hashset-like data structure (`GCIDMap`) in our `VerifierMain` component; this indicates that the specified group and all of its associating objects should be eligible for the next garbage collection. During the garbage collection process, we check if the objects from specified group is indeed released or not; if it is garbage collected, we conclude the specification has been satisfied, on the other hand, if the object is marked as active and promoted to live longer, then we report this as a violation of the specification in the form of a log. In order to do this verification process, we need to inspect every object that is marked as active; we do so by reading the Object Lifetime information from these objects, and compare what is contained in the `GCIDMap` which holds a list of to-be-released group ID. Note that, from the garbage collection site, group ID is embedded as `long` value; this can be mapped back to string representation using the same radix value, and makes it more intuitive to the user. When choosing appropriate data structure to hold to-be-released group ID, `THashMap` from GNU Trove, a library that provides high speed regular and primitive collections for Java, has been considered[13]. However, since GCIDMap is built as part of bootstrapping process, adding external classes to classpath during this process caused unknown exception.

The checking process is relatively straightforward in terms of ideas; however, the implementation requires us to have a good understanding into the garbage collection mechanism offered in Maxine VM. Maxine VM supports different types of garbage collection algorithms in `com.sun.max.vm.heap`. Particularly, we considered two types of garbage collection schemes, Semi-Space collection and Generational collection with stability and performance factors in mind.

Semi-space Garbage Collection

Semi-space Garbage Collection, also known as Copy Collection, is a garbage collection strategy to divide the usable heap into two regions; the two regions are generally named as "from-region" and "to-region". From-region is first used as free heap where object allocation takes place. Semi-space heap scheme maintains a "bump" pointer which points to the end of the last allocated memory, and when a new object is allocated, it bumps up the pointer to point to the updated end. This type of allocation is very fast because it involves only moving the pointer similar to the stack pointer. When the from-region is

almost full, memory reclamation happens. This involve of enumerating the roots (stack, main thread), and copying all live objects from from-region to to-region. Note that during this process, reference for objects are also updated as objects are moved from one memory region to another. Figure 5.4 shows general methodology of Semi-space collector.

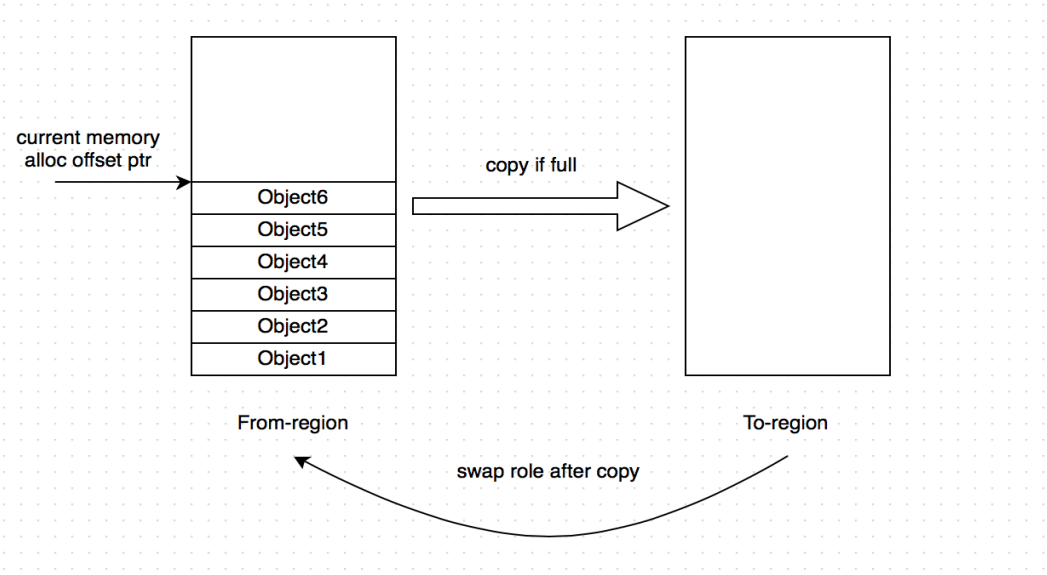

Figure 5.4: *Semi-space Collector*

This memory remapping process does add certain computational overhead to the GC but this process also ensures memory compaction; therefore, it not only reduces the fragmentation problem (not enough consecutive space to allocate object, and causes frequent GC), but also maintains good locality of reference, resulting in better cache hit ratios (faster execution). In Maxine VM, the reference remapping is done by `mapRef` method call, which deals with three different state of object reference:

1. Reference points to a not-yet-copied object in "fromSpace". The object is copied and a forwarding pointer is installed in the header of the source object in "fromSpace"

2. Reference points to a object in 'fromSpace' for which a copy in 'toSpace' exists. The reference of the 'toSpace' copy is derived from the forwarding pointer and returned

3. Reference points to a object in 'toSpace'. Return the reference.

Semi-space collection is the most stable garbage collection scheme in Maxine VM that is maintained since the first day Maxine project started. Semi-space collector outwin Mark-and-Sweep collection (also included in Maxine VM) which walks entire heap and has longer freeze time. However, due to the semi-space nature, Semi-space garbage collection requires twice as much the heap size of the other collection methods to keep the allocatable heap size the same. Further, since both "to" and "from" regions of the heap is manipulated during the copy process, potential page-fault might be higher.

Generational Garbage Collection

Generational collection is a garbage collection technique enabled in Maxine Generational Heap Scheme. The idea of Generational Garbage Collection is to divide heap into different generations, "nursery space" and "old generation space". The objects survived each garbage collection will get promoted to the next generation. Since heap is divided into different regions, it is possible apply different garbage collection method without locking the entire heap for new object allocation; typically, the young space will use garbage collection algorithm that allows fast collection (minor collection), and old generation employes full collection on both you space and old generation space. This garbage collection approach is based on the hypothesis that most of the objects die young, and if the objects are promoted to old generation, the chances of it becoming garbage in short time is low[3]. Since full collection involves determining the safe point and lock the heap which reduces the Java application performance, this garbage collection scheme enables less frequent full garbage collection, and more effective use of heap. In Maxine VM, generational collection is governed by `GenSSHeapScheme`, and it uses two generations: young generation and old generation. The young generation is a simple bump allocator nursery space without any aging mechanism, and the old generation space is organized into two semi-space which swaps its role during full collection. The heap layout is shown in Figure 5.5.

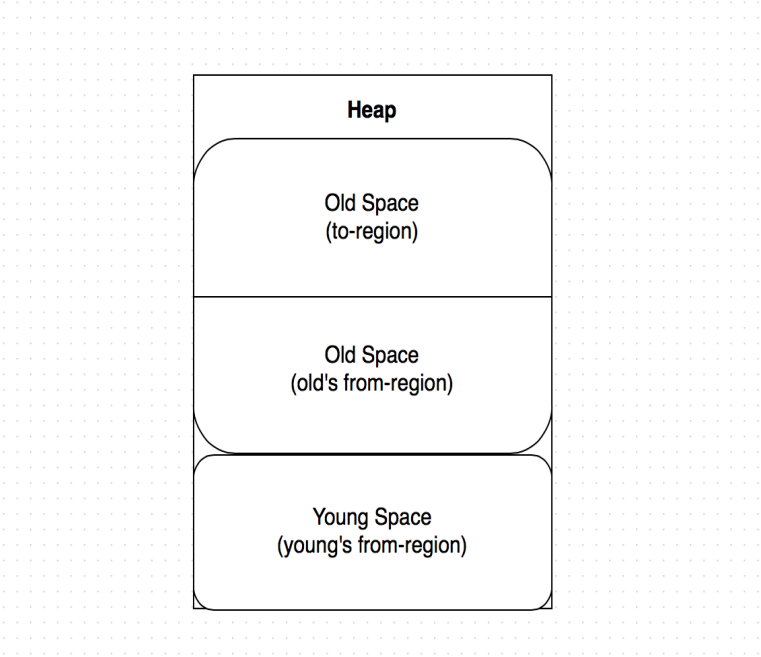

Figure 5.5: *Generational Heap Layout*

Generational heap space can also be defined in the following formula, and configured by user[25]:

$$M = YS + 2OS \tag{5.1}$$

where M is the heap size specified using the `-Xm` option; YS is young generation; OS is tenured generation. It is possible to configure percentage of young space, YP with respect to the entire heap through `-XX:YoungGenPercent` option, and effective heap size can be calculated as:

$$H = M/(2 - YP) \tag{5.2}$$

In Maxine VM, `GenSSHeapScheme` uses two types of garbage collection operation: minor collections, which only collect the young generation, and full collections, which collect both young and old generations. Garbage collection, for both generations, employs a Copy collection-like algorithm to evacuate live objects from the from-region into the to-region.

The main difference between the young generation and old generation collections is the choice of from-space, and the root set. The from-region in young generation is the young space, and the from-region in the old generation is the from-region of the old generation. Both young and old space collection use the old generation's to-region as their to-region. The old generation collection swaps the old generation's from-region and to-region, whereas a young generation collection doesn't; the young space is re-used as the young allocation space after collection. The root set for a young space collection is the same as that of the old space plus additional roots specified by a remembered set through card table. Card table is a data structure containing array of bits. Each bit indicates if a given range of memory in old generation that contains a write to a young generation object. Every update to a reference field of an object ensures that the card containing the updated reference field is marked dirty by setting its entry in the card table to the appropriate value. Maxine VM uses `CardTableRSet` class to implement card marking, which allows mapping from address to card table entry.

Since Semi-space collection in Maxine VM is more stable than Generational collection, we decided to intercept on Sesmi-space collection for our verification process. We understand that Semispace collection results in more frequent garbage collection when heap size is small; however, considering the robustness and stability goal we want to achieve with our runtime verifier, we decided to focus our implementation with Semi-space collector. In addition, as been articulated in Wimmer et al.[56], by choosing larger heap size, performance difference between Semi-space collector and Generational collector can be reduced. As we described in previous paragraphs, Semi-space collector copies survived objects to to-region of the heap. During this process, the reference to the survived objects also has to be remapped. From inspecting `mapRef` method in `SemiSpceHeapScheme`, we determined that `toRef` reference variable obtained from `getFrowardRef` method is the reference of survived objects. We used this reference variable as input parameter to our read header field to obtain the Object Lifetime information written into the header when this object is created. Note that, during garbage collection process, no new objects can be allocated in the heap, so we adjusted our reading mechanism for object header access, so there is no local variable used. Since semispace collector's `mapRef` method is also called outside of garbage collection for defragmentation, we also set lock to ensure that we check object header of an object just when `mapRef` method is called during the garbage collection process; we set the lock in `adviceBeforeGC`, and unset it in `adviceAfterGC`. The annotated group value is of `String` type, and we have converted it to `long` before writing into the header; we are certain that for any objects that has its lifetime information specified via annotation would have group value greater than zero in its header field (`Long.parseLong(string)` uses 32 as radix base). The system related objects have their misc header filed initialized to zero,

and unassigned object has negative misc header value. Therefore, we also filter the read object group value by only looking at objects with group value greater than zero during the garbage collection process, so we do not need to check every survived object against the specified to-be-released group value. In this approach, we retrieve survived object's Object Lifetime information during garbage collection process, and verify against what user has specified in `Release` annotation. We tested our runtime verifier with example client code, and successfully identified the junk object using Object Lifetime Specification.

5.4 Alternative Considerations: RVM

In this section, we present our work done with Jikes RVM, an open source meta-circular VM that we used as an alternative implementation with our runtime verification for Object Lifetime Specification study. During our implementation phase, we noticed a red stack overflow problem when running test program with VMA enabled image. After investigating various sources, and doing elimination testings, we determined that this is a bug in Maxine VMA bootsimage. We proceeded in an attempt to replace the existing advising mechanism in Maxine VM; however, we determined that the advising mechanism for Maxine VM is implemented in bytecode level which means we do not have access to the advising trigger point in Maxine source code. This posed significant difficulty for us to continue our implementation with Maxine VM. We reported this red stack overflow problem to Oracle Lab's Maxine Project team, and while waiting for their solution, we decided to look for an alternative VM for our implementation. With available options, we evaluated each JVM against preliminary criterions that we deemed to be compulsory for a JVM (Table 5.3).

JVM	**Meta-Circularity**	**JDK Compatibility**	**Documentation**	**EasytoModify**
Jikes RVM	Yes	High	Good	Yes
HotSpot	No	N/A	Good	No
VMKit	No	Partial (J3)	Moderate	Moderate
SableVM	Yes	Partial (not with new JDK)	Moderrate	Moderate

Table 5.3: Alternative JVM Selection

We investigated four open source JVM, including Jikes RVM, HotSpot, VMKit from LLVM and SableVM. The four major criterions that we evaluated against alternative VMs

are meta-circularity, JDK compatibility and user documentation and if it is modifiable. After reading the design documentation for each VM, We determined that Jikes RVM is the potential alternative that should be studied further. The main advantage is that Jikes RVM has been widely used by academia for garbage collection research due to its "easy-to-modify" Memory Manager Toolkit (MMTk), and has been proven to be "modifiable" in certain aspect of the research. Since it has been studied and applied in research, Jikes RVM also have good research mailing list which one can post discussion thread. Furthermore, Jikes RVM is compatible to the up-to-date JDK, and can run major benchmark to evaluate its performance. With basic understanding of the characteristics of Jikes RVM, we started our runtime verifier implementation on RVM sswhile waiting for the response from Maxine VM team.

5.4.1 Implementations

As being discussed earlier, Jikes RVM provides a flexible open testbed to prototype virtual machine technologies and experiment with a large variety of design alternatives. Jikes RVM is also meta-circular VM that provides ease of portability, and a good integration of virtual machine and application resources such as objects, thread and operating system interfaces. Our implementation with Jikes RVM started from determining the lifetime of Jikes RVM bootstrap sequence. We instantiated our runtime verifier in VM class in `org.jikesrvm` package after the VM finishes booting. We than modified object layout in `MemoryManager` class in `org.jikesrvm.mm.mminterface` package. Specifically, we increased header size by an extra Word; changed offset of `Misc` header; added a read and write utility support for added `Word` filed in `Misc` header. We then proceeded to implement even handler for the supported event callback from Jikes RVM. Annotation extraction mechanism is also implemented within each method callback handler to retrieve Object Lifetime Specifications. Jikes RVM also has a simplified mirror object for Method object, that is similar to Maxine's `methodActor`. From this mirror object, we can get annotation using reflection.

While implementing the callback handlers, we realized certain limitations with Jikes RVM. Unlike Maxine VM which has comprehensive support in different VM events, Jikes RVM does not have full event notification support for what we need. For instance, Jikes RVM does method invoked callback only once. Jikes RVM's "method invoked" advice is done when the method is first compiled, which means that when same method is called again, it will not issue advice again. In this case, we will have cases where Object Lifetime Specification does not get stored into an object. Although it is possible to force method recompile every time by setting certain flags, this will introduce performance problem to the Jikes RVM, and diminish the purpose of compiling the bytecode at runtime. Another

limitation is that Jikes RVM does not have "New object created" event advice so writing group information into object header filed is impossible in source code level. We studied the possibility of using bytecode rewriting strategy, and determined that we cannot instrument bytecode to directly access object header since it is not a "field" in the Java sense. To access object header field from bytecode we will need to add some native routines, or intercept existing "magic" routines in Jikes RVM from bytecode to access the header field; this is because native method would go through JNI like other JVM, and "magic" methods are recognized by all the bytecode compilers in Jikes RVM, and the compiler generates code for them directly. We concluded that in order to realize what we wish to achieve with runtime verifier, we need to intercept the Jikes RVM compilation process and add additional abstraction layer in bytecode level. This will potentially change Jikes RVM's internal behavior, and is not what we want as stated in our runtime verifier design goal. With these limitations and design belief we attach to our runtime verifier design, we decided to stop our implementation attempt with Jikes RVM.

Chapter 6

Experiment and Analysis

We described our design principle and implementations of the runtime verifier in previous chapter. We have articulated our design and implementations with respect to each component for the runtime verifier. We designed our tool to be light-weight based on lean design principle, and fault tolerant in handling exceptions that occur inside of the runtime verifier. We also implemented our tool in a isolated way through packaging and callbacks so that we do not change or alter the internal state for neither JVM nor the Java program under the test. We have used different data structures to dispatch and store annotated user specifications based on different annotation type, and verified Object Lifetime information with actual JVM events. Overhead is expected with our runtime verifier, but we try to control it through design choices and implementation strategies so that our runtime verifier have less influence to the application performance.

In this chapter, we evaluate our runtime verifier tool for its accuracy in detecting unintentional object retentions through a example program; we particularly test different Object Lifetime Specification semantics arise from different inter-procedural method calls. We also conduct performance experiments with open source benchmarks [5]to determine the overhead associated with the runtime verifier; to better understand overhead associated with major component of our runtime verifier, we isolate component based on process flow, and run benchmarks individually. We also try to draw performance implication by comparing the performance overhead of our runtime verifier on Maxine VM with OpenJDK, and Maxine VM with C1X optimizing compiler.

6.1 Accuracy of Runtime Verifier

The goal of our runtime verifier is to prove the applicability of collaborative Object Lifetime Specification approach in determining unused object retention. We enables the user to specify Object Lifetime Specification in the form of meta-data which keeps the internal state of the user program untouched but allows us to do runtime verifications with it. We implemented our runtime verifier with JDK compatible JVM, Maxine VM so that we can apply our runtime verifier on real-world Java programs. More importantly, the JDK-compatibility allows us to determine the runtime performance overhead of our runtime verifier by running existing benchmarks, so we can evaluate and improve the overhead associated with our tool to make it more appropriate for the use in Prototype Verification Testing and performance testing of the user program.

To evaluate the accuracy in detecting junk object, we considered different semantics in using our custom annotations; the currently supported custom annotations in verifying Object Lifetime Specification is shown Table 6.1 (annotation usage example is provided in Section 4.1).

Annotation Type	**Semantic Meaning**
@Create	Start of object lifetime; associate instanced objects to a group
@Release	End of object lifetime; objects associated to a particular group should be reclaimed

Table 6.1: Custom Annotations

Based on the meaning of our annotation, we developed four generic structures for the possible use cases, and tested the correctness of our verifier with respect to the semantics of intended meaning of the annotated Object Lifetime Specification. Figure 6.1 below illustrates these four cases.

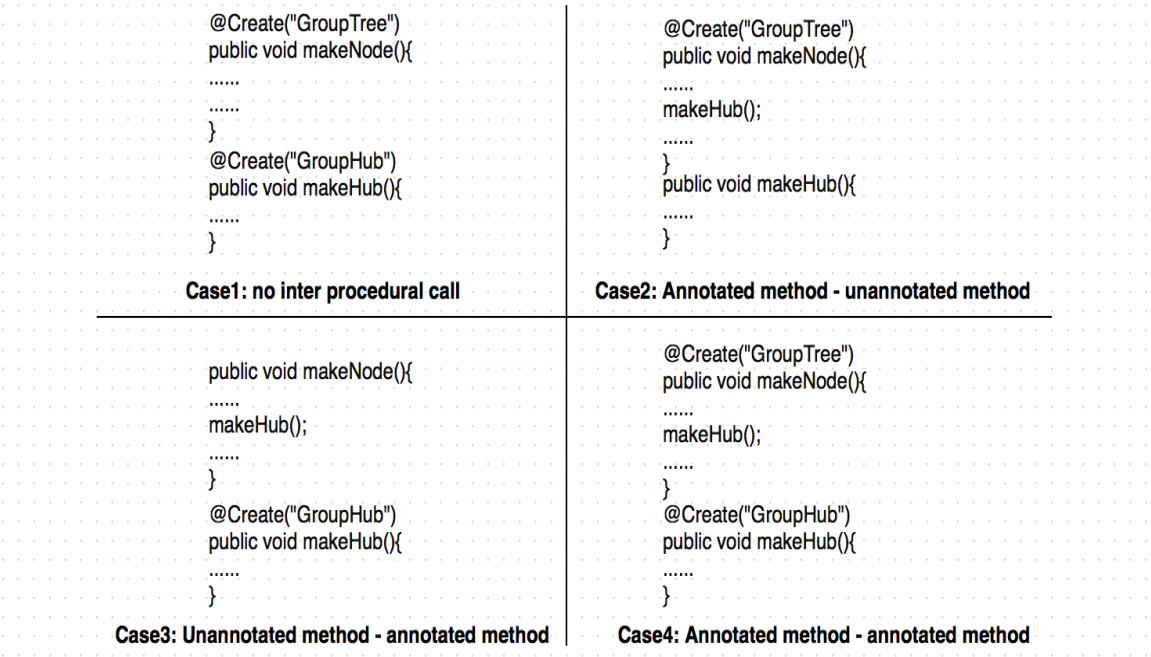

Figure 6.1: *Four Case Scenarios*

The first case is when a method annotated with `Create` is stand-alone, and there is no inter-procedural operations (i.e. no nested method calls). In this case, the Object Lifetime information should be effective only to the objects instantiated in this method. The second case is when there is an "un-annotated" method invoked in an annotated method; in this case, annotated Object Lifetime Specification should be transitive, and therefore effective to both objects instantiated in the annotated method and also to objects instantiated in method invoked inside of the annotated method. The third case is an unannotated method calls an annotated method; in this case, objects instantiated outside the annotated method will not carry Object Lifetime Specification, and objects created inside the annotated method will carry the annotated Object Lifetime information. Since the goal of dynamic runtime flow is to deviate the Object Lifetime information from the stack, objects created outside the inner annotated method and onward, will carry Object Lifetime Specification specified by the inner method's annotation.The fourth case is when an annotated method calls an annotated method within its method body. Similar to previous case, objects created inside the inner annotated method and onward will inherit Object Lifetime Specification specified by the inner method's annotation; any objects created before

invoking the inner annotated method will carry Object Lifetime Specification specified by the outer method's annotation. We propose the following relational equations to illustrate the Object Lifetime Specification propagation for respective cases; *LifeSpec* means Object Lifetime Specification, *M* indicates the method, and double arrow and its footnote denote the propagation of Object Lifetime Specification and its owner.

$$Case1 : LifeSpec\{M_1\} \gg_{m1} M1_{exit} \tag{6.1}$$

$$Case2 : LifeSpec\{M_1\} \gg_{m1} M_i \gg_{m1} M1_{exit} \tag{6.2}$$

$$Case3 : M_1 \rightarrow LifeSpec\{M_i\} \gg_{m_i} M1_{exit} \tag{6.3}$$

$$Case4 : LifeSpec\{M_1\} \gg_{m1} LifeSpec\{M_i\} \gg_{m_i} M1_{exit} \tag{6.4}$$

Based on above mentioned four cases, we wrote test programs, where we deliberately left object reference dangled in method annotated by Release custom annotation. We run the sample client codes with our runtime verifier under Maxine VM, and successfully determined junk objects which is not nulled in the method that is annotated with `Release`. Note that, in order to verify Object Lifetime specification, we require the garbage collection process to take place; this is because our verification engine requires input (survived object reference) from the garbage collector at runtime. To create an environment that potentially suggest to the Maxine VM that garbage collection should happen, we created dummy objects to increase the memory footprint. It should be noted that, industrial JVM does not give user the ability to force garbage collection to happen at a particular point; Java runtime analyzes the current heap usage and predict the next heap allocation to determine if garbage collection should take place. This means that we can not detect the junk object at the instance it is created. However, we assume that garbage collection will eventually take place at least once for non-trivial Java program, so we can detect the junk object during the garbage collection. Compared to memory monitoring tools that requires sequence of heap snapshots for analysis and identification of Java memory leak, we believe our runtime verifier can detect the junk object faster. An example code is shown in the Figure 6.2.

```
class Person{
   private String firstName;
   private String lastName;
   private int age;
   public Person ted;
   public Person mat;
   public Person jon;

   public Person(String firstname, String lastname, int a){
      firstName=firstname;
      lastName=lastname;
      age=a;
   }
   public Person(){
      firstName="A";
      lastName="A";
      age=0;
   }
   public String getName(){
      return (firstName+lastName);
   }
   @Create("Group11")
   public void allocatePerson(){
      System.out.println("-------allocatePerson_a()");
      allocatePersonInnerNew();  //calls another method, (Case) 2
      ted = new Person("Ted", "Ted", 21);
      ted.getName();

   }
   public void allocatePersonInnerNew(){
      System.out.println("-------allocatePerson_aInnerNew()");
      jon = new Person("Jon", "Jon", 21);
   }
   @Create("Group12")
   public void allocatePersonNew(){
      System.out.println("-------allocatePerson_aNew()");
      mat = new Person("Mat", "Mat", 21);
   }
   @ReleaseA("Group11")
   public void deallocatePerson(){
      System.out.println("-------deallocatePerson_a()");
      ted=null;
      //jon = null; ---> Junk object!
   }
}
```

Figure 6.2: *Example Client Code*

```
public class Test {
   public static void main(String[] args){
      System.out.println("\n==========Begin Test==========\n")
      Person p = new Person("Matthew", "Ma", 26);  //Person class from GCTest1
      String str= p.getName();
      System.out.println(str);
      p.allocatePerson(); //Group11 created
      process.run();
      p.deallocatePerson(); //objects in Group 11 should be reclaimed
      //create dummy objects
      int numObject = 0;
      List<Person> list = new ArrayList<Person>();
      for(int i = 0; i< 500; i++){
         list.add(new Person());
         numObject = numObject + 1;
      }
      list = null; //list and its containing object is eligible for GC
      System.out.println("Created test object which is garbage collectable: "+ numObject + " Person objects");
      p.allocatePersonNew(); //Group 12 created
      System.gc(); //after Group11's associated objects should be garbage collected by specification
      p.allocatePersonInnerNew();
      System.out.println("=============End Test============\n");

   }
}
```

Figure 6.3: *Example Client Code Main*

6.2 Performance Analysis for Runtime Verifier

Runtime verification will always add performance overhead to the application; the purpose of this section is to understand the performance overhead, analyze its roots and improve the performance of our runtime verification process. In this section, we show the performance results and analysis of our runtime verifier evaluated with open source benchmarks. As we stated earlier, the goal of our runtime verifier is to prove the applicability of collaborative Object Lifetime Specification approach in determining unused object retention. We implemented our runtime verifier with JDK compatible JVM, Maxine VM so that we can determine the runtime performance overhead of our runtime verifier by running open source benchmarks.

We evaluated our runtime verifier implemented on Maxine VM with DaCapo Benchmark, a tool for Java benchmarking consists of a set of open source, real world applications with non-trivial memory loads [5]. We chose DaCapo as our benchmarking tool because it not only includes complex static and dynamic metrics but also contains test programs to maximize the coverage of application domains and application behavior. Mckinley et al. from their experimental analysis, states that DaCapo benchmark has much richer code complexity, class structures, and class hierarchies that real life Java software application encompasses[8]. Our evaluation strategy distinguishes the performance overhead between each major functional component of our runtime verifier; we evaluate the overhead associated with T1X compilation and advice callbacks, annotation retrieval, and garbage collection interception.

We set `SemiSpaceHeap` Scheme with minimum heap size 256Mb and maximum heap size 2Gb, and report the average of four executions of each program, with a warmup sequence of two iterations. Note that some test in DaCapo benchmark could not be run due to deoptimization bugs in Maxine VM, and has been taken off. We chose maximum heap size to be 2Gb because fop and eclipse test run out of memory under 1Gb, and `SemiSpaceHeap` scheme only provide half of the heap size as effective allocation heap. Each test's description is provided in Appendix B.

We conducted the experiment on Kubuntu 64-bit Linux, Intel Core i7-2600K 3.4GHz with 15.7Gb of main-memory. Figure 6.4 shows the benchmarking results. Time variable in Logarithmic scale is provided to illustrate the comparison.

Figure 6.4 shows benchmarking results measured in milliseconds for Maxine VM with normal boot image versus Maxine VM using VMA-enabled boot image; both of the Maxine VM use `SemiSpace` Heap scheme as garbage collection model. There are two main

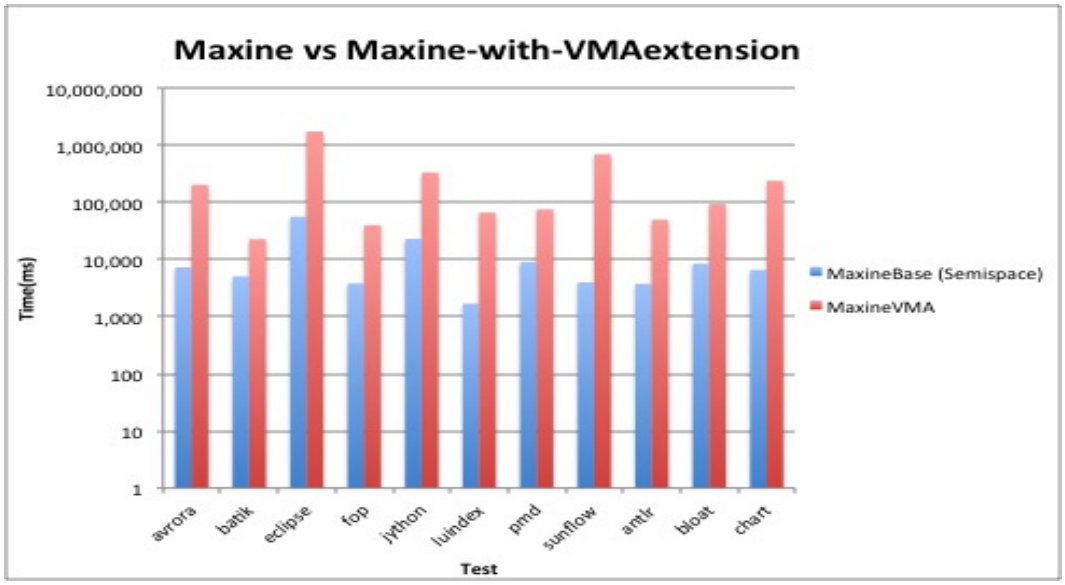

Figure 6.4: *Maxine VMA Performance in Logarithmic Scale*

differences between the two presented Maxine VM. First, MaxineVMA is an experimental extension that does advising for all the bytecodes execution. Further, since bytecode advising is only limited to code generated by the template JIT compiler, MaxineVMA uses T1X template-based compiler instead of the normal C1X compiler which supports recompilation for optimization. From the figure, we observe approximately 20 times performance slow down associated with Maxine VM with VMA extension compared to normal Maxine VM. We deem the major slow down comes from the different compiler used for JIT optimization; this is because compared to C1X compiler, T1X compiler's performance is expected to be much slower, as it is considered to be baseline compiler where no optimization is done[33]. Furthermore, there is overhead associated with runtime advising mechanism for bytecode execution for advice callback-enabled VM image. Comparing with OpenJDK1.7, Maxine VMA shows about 33 times performance slow down.

From closer inspection of the experimental data, we see significant performance overhead associated with `eclipse`, `luindex` and `sunflow`. Eclipse test executes some of the non-GUI `jdt` related performance tests for the Eclipse IDE, and `sunflow` test renders a set of images using ray tracing[5]. Benchmark test `luindex` displays inconsistent results, and some times fail for a particular file-not-found exception, so it will be excluded from our discussion for further analysis. After investigating the `eclipse` test, we learned that

`eclipse` and `sunflow` test have more *Weighted Methods per Class* than other tests, where weight is assigned to 1. WMC indicates the number of declared methods in a class; therefore, the larger the WMC, the program provides more behavior. Further, eclipse has the largest *Response for a Class*, which is a measure of the number of different methods that may execute when a method is invoked. Since one of VMA extension's major addition to the Maxine VM is the VM event advising mechanism (such as each method invoked and classloaded), the benchmarks results for `eclipse` and `sunflow` makes sense as it contains more methods and code complexities. To evaluate the average overhead on non-uniform data, we used geometric mean, an average calculation used to indicate the central tendency of a set of numbers by using the product of their values.

$$\Pi(X_i)^{\frac{1}{n}} \tag{6.5}$$

The geometric mean indicates that the averaged (centeral tendency) performance overhead associated with VMA-enabled Maxine VM is about 20 time more than the regular Maxine VM.

In next figure (Figure 6.5), we present the performance result comparisons for VMA-enabled Maxine VM, AnnotationExtraction-enabled Maxine VM, and AnnotationExtraction and GarbageCollection intercepted Maxine VM, where the last case is the full components of our runtime verifier (they all use VMA-enabled bootimage).

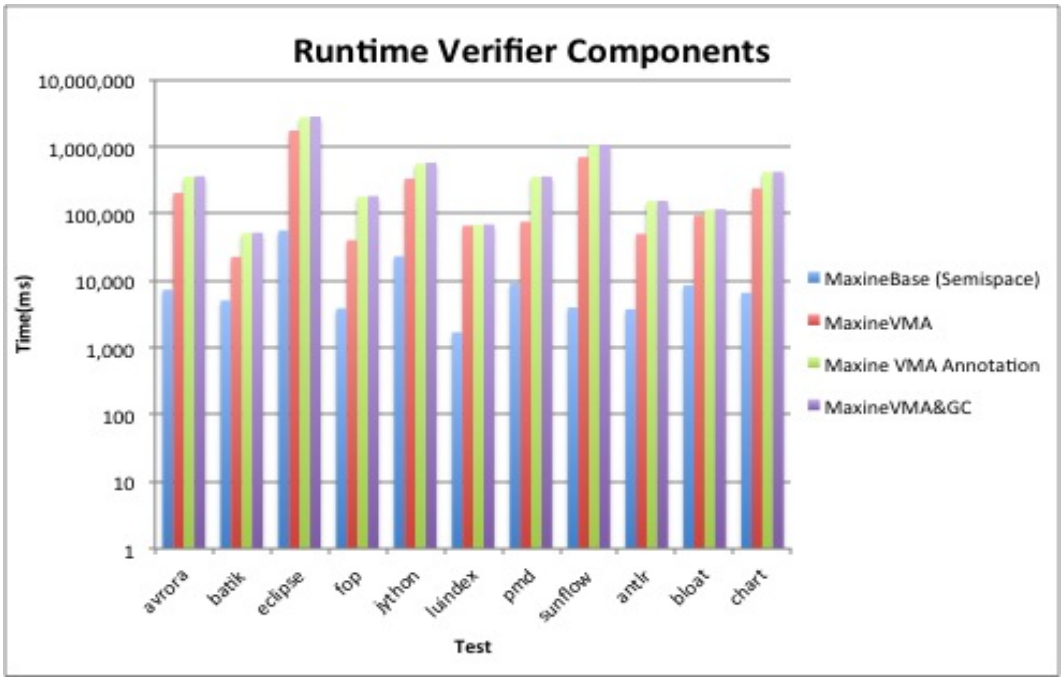

Figure 6.5: *RuntimeVerifier Performance with Respect to Different Components*

Based on experimental results, we found there are performance slow down associated with annotation extraction. Overhead with annotation extraction was also apparent for `eclipse` and `sunflow` test where Weighted Mehods per Class and Response for a Class are large. Since we do runtime processing of the annotation to store annotated Object Lifetime information, we check every method invoked to see if there is annotation present. The overhead associated to this accumulates, and resulted in the performance slow down that we see from the experiments. There is relatively large overhead increase observed in `fop` test, which takes an XSL-FO file, parses it and formats it and generates a PDF file. `Fop` test has relatively large coupling between object classes[8], which means that interaction between objects are substantially more complex. We could not directly infer this fact with overhead associated to annotation extraction, but we would think that checking if annotation is present in tiered method call from different class based on reflection would be more complex as it tries to find the class associated with it. Another test that showed bigger increase in the overhead is `pmd` test; this is a test that analyzes a set of Java classes for a range of source code problems[5]. `pmd` has a large score in *Lack of Cohesion in Methods* (LCOM); LCOM counts the methods in a class that are not related through the sharing of some of the fields [8]. Large LCOM value indicates that method in a class is relatively inde-

pendent; in other words, each method have their own functional role. Therefore, number of invocation on different methods has to increase to perform certain functional task; this will increase the amount of annotation presence check for `pmd` test. The geometric mean for the performance slow down between VMA-enabled Maxine VM to VMA-enabled and annotation component included Maxine VM is approximately 2 times. The last major component added as part of our runtime verifier is the interception of garbage collection process. The performance overhead is seen to be increased about 15% after adding this component. The overall performance overhead compared to VMA-enabled Maxine VM is 2.1 times. We also conducted same benchmark evaluation on OpenJDK1.7, and we determined that the overall performance overhead of our runtime verifier is about 58 times more than OpenJDK1.7.

Test	**openJDK1.7**	**Runtime Verifier**	**Delta**
Avrora	5310	361781	68.1
Batik	2866	52182	18.2
Eclipse	35757	2819128	78.9
Fop	2075	183651	88.5
Jython	9285	576189	62.1
Luindex	1585	69105	43.6
Pmd	4310	358723	83.2
Sunflow	2521	1072904	425.6
Antlr	2034	155981	66.8
Bloat	3865	116774	30.2
Chart	4120	421769	98.5
		Geometric Mean	58.4

Table 6.2: OpenJDK1.7 vs Runtime Verifier Performance Comparision

From the experiment, we concluded that major overhead results from Maxine VMA extension, which caused more than 30 times performance slow down. Since we require bytecode advising for our runtime verification, overhead associated with bytecode advising and T1X compilation (bytecode advising only works with T1X template-based baseline compiler) is unavoidable. However, when comparing with OpenJDK1.7, there is another factor that can have big influence to the performance: garbage collection. The type of

garbage collection OpenJDK1.7 uses is multi-generation collector; while we used semi-space collector. As being discussed in previous chapter, we chose SemiSpace heap scheme to intercept to ensure functional stability with the runtime verifier, while sacrificing the performance. By experimenting with different garbage collection strategies in Maxine VM, we see performance gain for Generational Collection to be as follows (Figure 6.6).

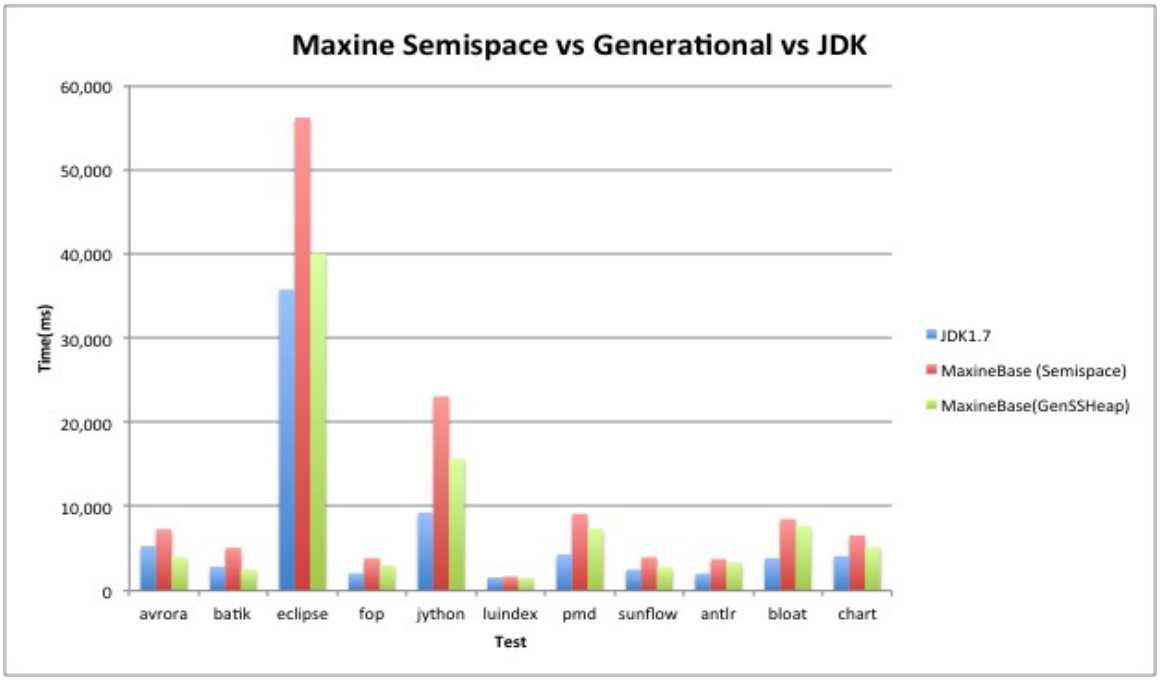

Figure 6.6: *JDK1.7 vs Maxine VM SemiSpace GC vs Maxine VM Generational GC*

From the figure, it can be seen that using Generational collection results in much lower performance slow down for Maxine VM compared to Semispace collection. Our analysis shows that, Maxine VM with Semispace GC results in 80% performance slowdown, while Maxine VM with Generational GC cause only 27% performance slowdown in averrage, and even better performance than OpenJDK1.7 for some tests. To reduce the overall performance overhead associated with current implementation, we revisited Generational Garbage Collection Scheme in Maxine VM. We inserted debug statements into classes in `com.sun.max.vm.heap.gcx` package, such as `GenSSHeapScheme` and `Evacuator` class. We particularly looked for the reference update and forward reference mechanism, and intercepted Generational Collector to retrieve the Object Lifetime Specification for evacuated

objects. We rerun the same performance benchmark and compared the performance overhead between our runtime verifier with SemiSpace collector and the case with Generational collector. We obtained the following results shown in Figure 6.7.

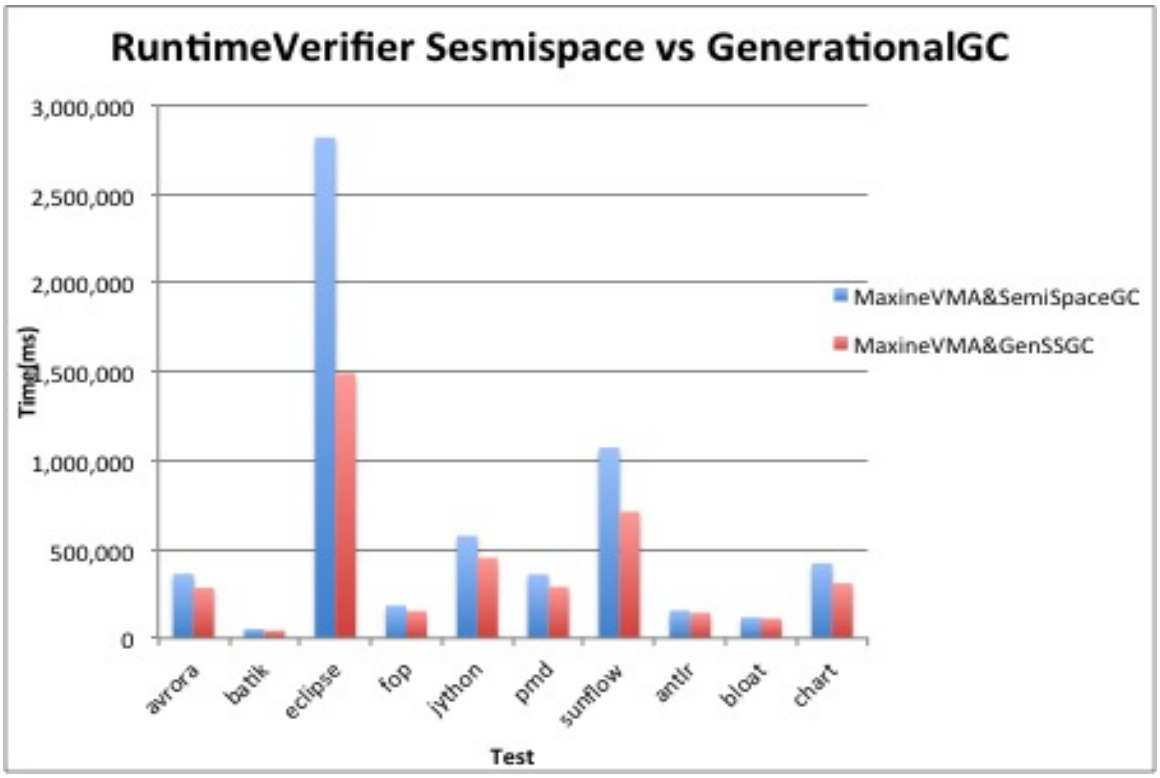

Figure 6.7: *MaxineVMA SemiSpace GC vs MaxineVMA Generational GC*

The result shows performance improvement for up to 47% (eclipse test), and an average of 22% performance gain. To understand better about the performance gain with Generational garbage collection, we looked for quantitative support to analyze the effect of garbage collection to the performance of the application. From our investigation, we determined that reducing the GC pause time in multi-threaded program can have significant impact on the application performance. An experiment (Appendix B.1) presented in an Oracle GC Tuning article shows that if an application spends 1% of its execution time on garbage collection, it will loose more than 20% throughput on a 32-processor system; further, if the pause time is increased to 2%, another 18% performance slowdown is observed [39]. Some tests in DaCapo benchmark balance the load onto 8 threads for the

test machine; there are four processors on the test machine, so we believe that the effect of increasing the GC pause time is represented in the region circled in red, shown in Appendix B.1. From the figure presented in Appendix B.1, we see that the influence of 1 % garbage collection pause time can be about 7% performance slow down; additional 1% pause time increase can reduce the throughput by 7%. Since Generational Garbage collector incur less pause time than Semi-space collector, we infer that the performance improvements with Generational garbage collection is mainly contributed by less pause time.

Comparing to the OpenJDK, MaxineVMA with Generational Garbage collection results in 45 times performance slowdown which is much better than the 58 time slowdown that comes with previous implementation with Semispace collector (previously, we have also determined that overhead associated with VMA extension itself is a factor of 33).

Chapter 7

Related Work and Future Works

In previous chapters, we described our design principle, implementation and experimentation of our runtime verifier. We have discussed our idea of using Object Lifetime Specification to uncover unused object retention in Java application; we have introduced Object Lifetime concept, grouping and collaborative way to specify Object Lifetime through annotation. We have also proceed detailed description on our design approach, tool architecture and implementation of our runtime verifier with Maxine VM; we discussed requirements, control flow, component design and implementation details for each component of our tool using figures and sample code. We have conducted performance experiments using our tool, running benchmarks, and evaluated our tool performance with respect to OpenJDK and Maxine VM without our runtime verifier.

In this chapter, we discuss the limitations and implications of our tool and review related research from the academia; based on these study, we articulate a direction for our next step future work to improve our runtime verifier in terms of usability, accuracy and performance.

7.1 Limitation Analysis and Future Works

Based on experimental results, we found certain performance limitations associated with our runtime verifier. The overhead mainly comes from three source, VMA extension, annotation extraction and garbage collection. In this section, we describe limitations we observed, analyze associated issues with it, and discuss potential improvements that should be studied in the future.

7.1.1 VM Event Handling

Event listener corresponds to VM advising mechanism that we used. From tiered analysis, we found that advice callback-enabled Maxine VM performs twenty times slower than regular Maxine VM. Repetitive and frequent method invocation causes more frequent advise callback which triggers a series of conditional checks in our callback handlers, accumulating overhead from each procedure.

Currently, we use VM level bytecode advising mechanism provided by Maxine VM's VAM extension; VMA-enabled boot image uses the T1X compliler, a baseline template-based compiler, and does not work with C1X, an optimizing compiler that regular Maxine VM uses. Therefore, there is no optimizations done to the bytecode to generate platform specific instruction sets. We believe this has greater impact to the application performance on top of advice callbacks invoked on every method call. In the future, we would like to explore the possibility of using C1X, and determined how we can elaborate existing bytecode advising mechanism to it. We have investigated the possibility of redirecting bytecode advising to another abstraction layer which communicate with C1X compiler. However, we found that bytecode advising mechanism and C1X is implemented in different project, and current Maxine VM bootimage generator does not allow inclusion of both. Therefore, we believe, the solution to substitute T1X should be sought after with Maxine Project Team. The major difficulty for creating bytecode advising framework with C1X compiler is that C1X does more than template lookup; it introduces XIR intermediate representation, and we do not have complete knowledge and design specifications of what layers of transformation we need to consider in order to develop a C1X supported advising mechanism by ourselves. Based on our investigation, we found there is a visualization tool called Java HotSpot client compiler visualizer[37] which helps the introspection of intermediate representations from bytecode parsing to machine code generation process. We believe this tool is worth investigating in solving the above posed problems in the future.

7.1.2 Annotation

From the benchmark evaluation, we have also seen certain amount of overhead associated with runtime annotation processing. Currently, annotation is the only method we use to allow user to specify Object Lifetime information in the program. This is a performance limitation because for every method call, we check if there is annotation present. We have considered different level of annotation, such as class level. However, class level annotation does not allow us to distinguish object allocated in different context (i.e.. different group),

therefore, restrict the expressiveness of our Object Lifetime specification. Further, method level annotation allows us to deviate from the stack frame while class level annotation does not. Since our research goal is to design a collaborative system for runtime verification, annotation, as an user input, is necessary. Therefore, the amount of overhead associated with annotation processing is inevitable. However, we think it should be possible to add another intermediate step, so that we do annotation processing at compile time, which might reduce the runtime overhead that comes from annotation processing.

An alternative approach, that is worth study in the future is to design an annotation compiler, which will parse user annotation during the compile time of Java source code. The annotation compiler should be designed, such that it retrieves annotated Object Lifetime specification from the source code, and embed this information as part of the generated bytecode. The annotation compiler must be able communicate with Java compiler from the JDK and also Maxine VM, so the object's lifetime specification is integrated into the generated bytecode. In this way, runtime verification would only consist of VM event advice handling and garbage collection interception, which would reduce the overall runtime overhead.

7.1.3 Selection of JVM

For this thesis, we have integrated our runtime verifier into Maxine VM, a meta-circular VM developed by Oracle Lab. Maxine VM is well designed in many aspects (refer to Chapter 5); however, there are a few considerations that should be understand for future work of runtime verifier.

Meta-Circularity: Firstly, meta-circular VM's bootstrap process is very complicated and hard to diagnose if low level exception happens. We have run into problem not being able to run VMA-enabled Maxine VM with test handler due to JDK dependence problem of VMA extension, which produced red stack overflow, that pointed us to debugging into native code trap.c. The debugging process was rather complicated which involved inspecting platform environment, Java environment (we noticed that some Maxine source code do not work well with JDK 1.6) and `VMAConfiguration` code. We reported this problem to our Maxine project contact, and we were informed that the solution to this problem is to set `+VMAXJDK` flag to exclude VMA advice mechanism for JDK related calls; in addition, older version of the Maxine source code should be used as it is considered to be more stable with VMA extension.

Bytecode to Machine Level Instruction: JVM generally has two choice of generating platform specific instruction set from bytecode: interpretation or JIT compilation.

Interpretation requires the "dictionary-like" lookup every time a bytecode is visited, and translated to machine code; while JIT compiler enables the optimization by compiling the frequently visited bytecode "hotspot" into highly tuned machine code (refer to Chapter 3). Although there is performance decrease with interpreter-only JVM, it greatly simplifies implementing custom VM event advising structure. Since Maxine VM does not have interpreter, our choice of implementing VM event listener was limited to using VMA extension which is not designed specifically for observing object lifetime related VM event.

Garbage Collection: Garbage collection is an important factor that influences application's overall performance. Different garbage collection algorithm results in garbage collection to take place at different frequency and at different part of the heap (refer to Chapter 3), therefore introducing different amount of pause time to the application. Maxine VM offers different garbage collection algorithms governed by `HeapScheme`. Of the provided heap schemes, we investigated two advanced garbage collection schemes, SemiSpaceHeap Scheme and Generational Heap Scheme. We applied our implementation to `SemiSpaceHeap` scheme because Semi-space collector is the most stable collector in Maxine VM (recommended by our contact from Maxine team). SemiSpace Heap scheme requires bigger heap space and it is less effective compared to Generational collection as we shawn in the performance experiment.

From the experiment, we saw major overhead resulted from VMA extension which is required for our runtime analysis. This overhead has correlation with our JVM selection; however, to reduce overhead resulted from the Maxine VM itself, we also intercepted Generational collector to improve the overall performance. One problem associated with this implementation is that, some time our runtime verifier reports false positive cases for junk objects. For instance, an object is nulled in the method annotated with `@Release` but after the minor collection, we still get a report saying this object is retained as junk. Our investigation shows that this results from issues with incremental garbage collection where minor collection retains dead object live owing to the conservative set of roots that is used. Another problem we observed is, during benchmarking, verifier with Generational collection scheme crashed more often than Semi-space implementation. Notably, `luindex` test wasn't able to run for runtime verifier using Generational collection. These problems should be addressed in the future.

7.2 Related Work

In this thesis, we presented our research work in detecting Java memory leak using Object Lifetime Specification. We have been inspired from numerous works done by other

researchers in the community in terms of concepts and methodology in detecting unused object retention for Java language. We provided brief discussion on related works when articulating the challenges associate with detecting memory leak in Java in Chapter 2. In this section, we review the representative works for the contributions they made as a conclusion to this chapter.

7.2.1 Object Lifetime Specification

At the foremost, our work would not be possible without Object Lifetime Specification invented by Professor Derek Rayside, who is also my advisor. Dr. Rayside provided the direction of having object's lifetime specified as a property, and determine unused object retention from the aspect of verification of satisfiability at runtime. Object Lifetime Specification associates objects that has similar lifetime into a Group upon creation time, which effectively reduces the amount of work required to do in recording specifications into each objects. Object's end time is effectively been considered as when it is getting garbage collected, and this is the exact place we verify the specified Object Lifetime Specification with actual JVM object reclamation events.

7.2.2 Profiling

Profiling is an important method used to understand better about object lifetime behavior for optimization, memory leak detection and developing new garbage collection algorithm. Particularly we looked through profiling techniques that helps us uncover object lifetime, usage and ownership. Hertz et al. proposed Merlin object lifetime algorithm which efficiently computes object lifetimes by time-stamping live objects when they lose an incoming reference and later uses the timestamps to reconstruct the time at which the object became unreachable. Merlin provide precise profiling information, however, since Merlin's execution time is proportional to the total allocations plus the number of times each object loses an incoming reference, it introduces significant amount of overhead[18]. Since it is particularly challenging to obtain object lifetimes both precisely and efficiently, techniques, such as Resurrector has been developed to enable tunable profiling strategy based on mutator execution that explores the middle ground between high precision and high efficiency[57]. Rayside et al. proposed Object Ownership profiling, the first heap profiling technique to report a hierarchy annotated with quantitative information about time and space[52]. There are other more progressive profiling techniques that target to eliminate redundant objects through profiling, such as Object equality profiling[31], which tries

to discover opportunities for replacing a set of equivalent object instances with a single representative object to save space. Statistical profiling technique that traces program allocations/ frees to construct a heap model, and uses adaptive profiling infrastructure to monitor loads/stores to these objects has also be proposed and implemented[17].

7.2.3 Memory Leak Detection

Memory leak detection in Java involves wide range of research, such as in lifetime approximation, object usage prediction/traces and others. Object lifetime approximation is developed as an alternative to reduce the cost of producing the garbage collection traces; it intends to model the object allocation and object death behavior of actual programs. The simplest application of object lifetime to quantitative analysis is reference counting, which associates a count of incoming references with each object, and use that as liveness check of objects[10], Stefanovic et al described mathematical functions that model object lifetime characteristics based on the actual lifetime characteristics[55]. Hirzel et al. shows the quality of the results of a reachability-based leak detection is largely dependent to the type and liveness accuracy of its reachability traversal[19]. To approximate object usage, Dufour et al. uses a blended escape analysis to characterize and find excessive use of temporary data structures; by approximating object lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the "focused" area[12]. Research that targets to assist releasing garbage objects has also been proposed, such as Free-me, a static technique that identifies when objects become unreachable, and inserts calls to free garbage objects[15].

7.2.4 Existing Java Memory Usage Monitoring Tools

Existing tools for Java memory leak detection is also studied during the course of implementation. Through this process, we gained insight on expected performance criterion and leak detection effectiveness. Jconsole[51] is a monitoring tool which uses extensive JMX instrumentation of the Java virtual machine to provide information on performance and resource consumption of applications running on the Java platform. One can attach Jconsole to the test application, and Jconsole will show the memory consumption at different memory pool from the Memory Tab of the GUI. VisualVM is a monitoring tool from JDK where user can set profiler to record allocation stack traces, and take heap snapshot of object for comparing memory usage[40]. JMap is also useful tools that can be used to generated and analyze heap dump file which can be an input to VisualVM; JMap can print

a memory map tree for all objects for a given process[41]. Eclipse software foundation also offers Memory Analyzer (MAT) which is used to analyze productive heap dumps with objects, and calculate retained sizes of objects to determine leaked object[42].

Chapter 8

Conclusion

In this thesis, we presented our work of Virtual Machine-assisted Collaborative Junk Object Detection using Object Lifetime Specification. We developed a runtime verifier that allows user to specify Object Lifetime Specification in the form of annotation in Java program, and verify it against actual VM events that takes place during the program execution. We implemented our runtime verifier on top of Maxine VM, a JDK compatible meta-circular Virtual Machine developed by Oracle Lab. We modified Java Object Header Layout to allow insertion of extra state-analysis Word to record Object Lifetime information. We also implemented runtime annotation extraction based on reflection to retrieve user annotated Object Lifetime information when a method is invoked. Finally, we intercepted Semi-space collector to verify Object Lifetime Specification for survived objects to determine if it belongs to unintentional object retention.

We evaluated our runtime verifier against different Object Lifetime Specification semantics for the correctness, and also experimented our tool with DaCapo benchmark to determine the associated performance overhead associated to each component of our tool. The test results shows that our tool correctly identified junk objects under different Object Lifetime Specification semantics. The benchmarking process showed that our tool adds up to 58 times slow down to the program compared to OpenJDK 1.7. The major overhead comes from VMA extension which is a functional restriction by the Maxine VM. To reduce this major overhead, we revisited Maxine Generational Heap scheme to see if we can reduce the amount of overhead by using industrial-strength garbage collection technique, Generational garbage collection. Our new implementation shows about 22% performance improvement in average, and up to 45% improvements for some of the tests. Research comes with limitations and sparks the motivation for future improvement; we analyzed the

current limitation of our work, and suggested possible course of studies in improving the overall performance and accuracy of our runtime verifier.

Through this research, we proved that Collaborative JVM-assisted memory leak detection is possible with Object Lifetime Specification. We implemented a runtime verifier on a JDK-compatible JVM by elaborating with JVM technologies. We determined the runtime overhead of the runtime verifier by experimenting and running DaCapo benchmarks. We showed that our approach can be potentially used to detect Java unintentional object retention during Prototype Verification Test.

APPENDICES

Appendix A

Maxine VM

A.1 Maxine VM Startup Sequence

```
@m22ma Debug: DirectReferenceScheme @PRIMORDIAL
@m22ma Debug: XOhmLayoutScheme @PRIMORDIAL
@m22ma Debug: ThinInflatedMonitorScheme @PRIMORDIAL
@m22ma Debug: GenSSHeapScheme @PRIMORDIAL
@m22ma Debug: JavaRunScheme @PRIMORDIAL
@m22ma Debug: DirectReferenceScheme @PRISTINE
@m22ma Debug: XOhmLayoutScheme @PRISTINE
@m22ma Debug: ThinInflatedMonitorScheme @PRISTINE
@m22ma Debug: GenSSHeapScheme @PRISTINE
@m22ma Debug: JavaRunScheme @PRISTINE
@m22ma Debug: DirectReferenceScheme @STARTING
@m22ma Debug: XOhmLayoutScheme @STARTING
@m22ma Debug: ThinInflatedMonitorScheme @STARTING
@m22ma Debug: GenSSHeapScheme @STARTING
@m22ma Debug: JavaRunScheme @STARTING
@m22ma Debug: DirectReferenceScheme @RUNNING
@m22ma Debug: XOhmLayoutScheme @RUNNING
@m22ma Debug: ThinInflatedMonitorScheme @RUNNING
@m22ma Debug: GenSSHeapScheme @RUNNING
@m22ma Debug: JavaRunScheme @RUNNING
---Creating ObjectLifeTimeVerifier_Main
```

```
2014-06-20 12:36:03.016 : ObjectLifeTimeVerifier_Main-Test
---still alive......
2014-06-20 12:36:03.017 : ObjectLifeTimeVerifier_Main-Test
---ObjectLifeTimeVerifier_Main finalize(). GC takes place
@m22ma Debug: DirectReferenceScheme @TERMINATING
@m22ma Debug: XOhmLayoutScheme @TERMINATING
@m22ma Debug: ThinInflatedMonitorScheme @TERMINATING
@m22ma Debug: GenSSHeapScheme @TERMINATING
@m22ma Debug: JavaRunScheme @TERMINATING
```

A.2 T1X Non-optimizing Compiler

The description of T1X compiler is adopted from Maxine Project Wiki [45].

T1X is template-based baseline compiler and is Maxine's first line of execution (Maxine has no interpreter). As such, it's primary goal is to produce code as fast as possible. Code quality is of secondary concern. It also closely matches the JVM specification's execution models. That is, the JVM operand stack and local variable variable array is modeled directly. This makes it suitable for implementing bytecode level debugging as well being the execution mode the de-optimization process uses as its end target.

Having the templates written in Java makes modifying or extending the compiler fairly easy. More importantly, it also means the compiler is very portable and it mostly relies on the optimizing compiler. It performs very little direct machine code generation.

A.3 Before Filtering JDK-related Method Callback

```
@m22ma Debug:  VMAJavaRunScheme @RUNNING
@m22maDebug--- in myVerifierVMEventHandler initialize()
@m22maDebug---Creating ObjectLifeTimeVerifier_Main
@m22maDebug--- in adviseBeforeInvokeVirtual(): void println(String)
@m22maDebug--- in adviseAfterNew(): Person
@m22maDebug--- in adviseBeforeInvokeVirtual(): String getName()
@m22maDebug--- in adviseAfterNew(): java.lang.StringBuilder
@m22maDebug--- in adviseBeforeInvokeVirtual(): StringBuilder append(String)
```

```
@m22maDebug--- in adviseBeforeInvokeVirtual(): StringBuilder append(String)
@m22maDebug--- in adviseBeforeInvokeVirtual(): String toString()
@m22maDebug--- in adviseBeforeInvokeVirtual(): void println(String)
@m22maDebug--- in adviseBeforeInvokeVirtual(): Method getMethod(String, Class[])
@m22maDebug--- in adviseBeforeInvokeVirtual(): Method getMethod(String, Class[])
@m22maDebug--- in adviseBeforeInvokeVirtual(): Method getMethod(String, Class[])
@m22maDebug--- in adviseBeforeInvokeVirtual(): Method getMethod(String, Class[])
@m22maDebug--- in adviseBeforeInvokeVirtual(): Method getMethod(String, Class[])
@m22maDebug--- in adviseBeforeReturn(): Class
@m22maDebug--- in adviseBeforeReturn(): RetentionPolicy
@m22maDebug--- in adviseBeforeInvokeVirtual(): Method getMethod(String, Class[])
@m22maDebug--- in adviseBeforeInvokeVirtual(): Method getMethod(String, Class[])
@m22maDebug--- in adviseBeforeInvokeVirtual(): Method getMethod(String, Class[])
@m22maDebug--- in adviseBeforeInvokeVirtual(): Method getMethod(String, Class[])
@m22maDebug--- in adviseBeforeInvokeVirtual(): Method getMethod(String, Class[])
@m22maDebug--- in adviseBeforeReturn(): Class
@m22maDebug--- in adviseBeforeInvokeVirtual(): void allocatePerson_a()
@m22maDebug--- in adviseBeforeInvokeVirtual(): void println(String)
@m22maDebug--- in adviseAfterNew(): Person
@m22ma Debug: VMAJavaRunScheme @TERMINATING
```

A.4 After Filtering JDK-related Method Callback

```
@m22ma Debug:  VMAJavaRunScheme @RUNNING
@m22maDebug--- in myVerifierVMEventHandler initialize()
@m22maDebug---Creating ObjectLifeTimeVerifier_Main
@m22maDebug--- in adviseBeforeInvokeVirtual(): void println(String)
@m22maDebug--- in adviseAfterNew(): Person
@m22maDebug--- in adviseBeforeInvokeVirtual(): String getName()
@m22maDebug--- in adviseBeforeInvokeVirtual(): void println(String)
@m22maDebug--- in adviseBeforeInvokeVirtual(): void allocatePerson_a()
@m22maDebug--- in adviseBeforeInvokeVirtual(): void println(String)
@m22maDebug--- in adviseAfterNew(): Person
@m22ma Debug: VMAJavaRunScheme @TERMINATING
```

Appendix B

Dacapo Benchmark

B.1 Benchmarks

avrora: simulates a number of programs run on a grid of AVR microcontrollers
batik: produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik
eclipse: executes some of the (non-gui) jdt performance tests for the Eclipse IDE
fop: takes an XSL-FO file, parses it and formats it, generating a PDF file.
jython: inteprets a the pybench Python benchmark
luindex: Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible
pmd: analyzes a set of Java classes for a range of source code problems
sunflow: renders a set of images using ray tracing
antler: parses one or more grammar files and generates a parser and lexical analyzer for each
bloat: performs a number of optimizations and analysis on Java bytecode files
chart: uses JFreeChart to plot a number of complex line graphs and renders them as PDF

B.2 Garbage Collection Tuning

The following figure shows the effect of GC Pause time to the throughput rate of a Java application with respect to the different number of processors.

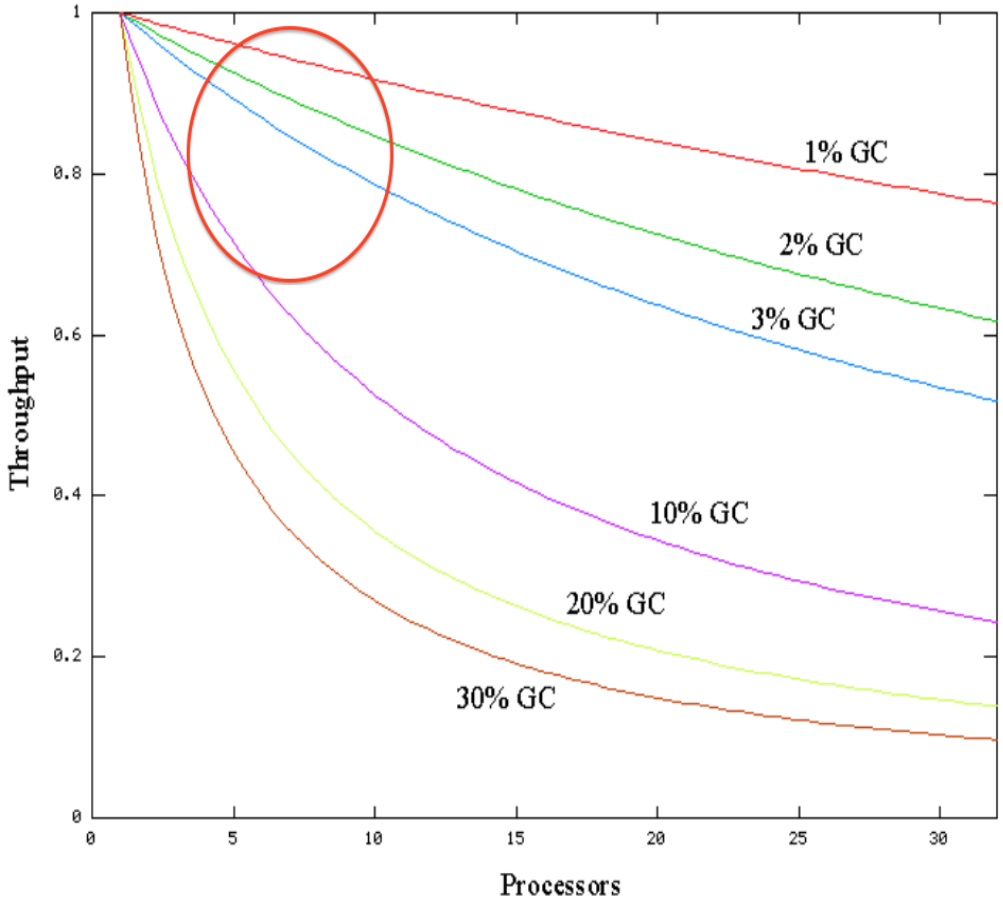

Figure B.1: *Impact of GC Pause Time to Throughput Rate with Different Number of Processors*

References

[1] Android. Activities. http://developer.android.com/guide/components/activities.html, 2014.

[2] P. Dibble. B. Brosgol, J. Gosling. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[3] Henry G. Baker. Infant mortality and generational garbage collection. *SIGPLAN Not.*, 28(4):55–57, April 1993.

[4] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 187–196, New York, NY, USA, 1993. ACM.

[5] DaCapo Benchmark. The Benchmarks. http://www.dacapobench.org, 2009.

[6] Zev Benjamin. Runtime verification of object lifetime specifications. Master's thesis, Massachusetts Institute of Technology, 2009.

[7] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA, 2004. ACM.

[8] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and

analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[9] J. Bloch. *Creating and Destroying Objects*. Effective Java. Prentice Hall, New Jersey, second edition, 2008.

[10] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, December 1960.

[11] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[12] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 59–70, New York, NY, USA, 2008. ACM.

[13] Rob Eden. The GNU Trove Library. http://trove.starlight-systems.com, 2012.

[14] David Evans. Static detection of dynamic memory errors. *SIGPLAN Not.*, 31(5):44–53, May 1996.

[15] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. Free-me: A static analysis for automatic individual object reclamation. *SIGPLAN Not.*, 41(6):364–375, June 2006.

[16] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. Free-me: A static analysis for automatic individual object reclamation. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 364–375, New York, NY, USA, 2006. ACM.

[17] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. *SIGPLAN Not.*, 39(11):156–164, October 2004.

[18] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. Generating object lifetime traces with merlin. *ACM Trans. Program. Lang. Syst.*, 28(3):476–516, May 2006.

[19] Martin Hirzel, Amer Diwan, and Johannes Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Trans. Program. Lang. Syst.*, 24(6):593–624, November 2002.

[20] Hajime Inoue, Darko Stefanovic, and Stephanie Forrest. On the prediction of java object lifetimes. *IEEE Trans. Comput.*, 55(7):880–892, July 2006.

[21] Ronald Cohn Jesse Russell. *Observer Pattern*. Book on Demand, 2012.

[22] Oracle Lab. The Maxine Project: Actor classes in the Maxine VM. https://wikis.oracle.com/display/MaxineVM/Actors#Actors-MethodActor, 2011.

[23] Oracle Lab. The Maxine Project: Object representation in the Maxine VM. https://wikis.oracle.com/display/MaxineVM/Objects#Objects-OHMlayout, 2011.

[24] Oracle Lab. The Maxine Project: Object representation in the Maxine VM. https://wikis.oracle.com/display/MaxineVM/Objects#Objects-HOMlayout, 2011.

[25] Oracle Lab. Generational Heap Scheme. https://wikis.oracle.com/display/MaxineVM/Generational+Heap+Scheme, 2012.

[26] Oracle Lab. The Maxine Project: Dynamic Extension. https://wikis.oracle.com/display/MaxineVM/Boot+Image#BootImage-StaticExtension, 2012.

[27] Oracle Lab. The Maxine Project: VM Boot Image. https://wikis.oracle.com/display/MaxineVM/Boot+Image#BootImage-StaticExtension, 2012.

[28] Oracle Lab. The Open Source Maxine VM Project. https://wikis.oracle.com/display/MaxineVM/Home, 2013.

[29] Oracle Lab. Virtual Machine Level Analysis. https://wikis.oracle.com/display/MaxineVM/Virtual+Machine+Level+Analysis, 2013.

[30] Oracle Lab. The Maxine Project: C1X Compiler. https://wikis.oracle.com/display/MaxineVM/C1X, 2014.

[31] Darko Marinov and Robert O'Callahan. Object equality profiling. *SIGPLAN Not.*, 38(11):313–325, October 2003.

[32] Bernd Mathiske. Systems programming in the maxine vm: how to enable it and how to get around it. PPPJ08 Principles and Practice of Programming in Java, sep 2008.

[33] Oracle. How much optimization does the baseline compiler do. https://wikis.oracle.com/display/MaxineVM/FAQ, 2012.

[34] Oracle. JVM Tool Interface Agent Command Line Options. http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#starting, 2012.

[35] Oracle. http://www.oracle.com/technetwork/java/intro-141325.html, 2014.

[36] Oracle. Annotation Basics. http://docs.oracle.com/javase/tutorial/java/annotations/basics.html, 2014.

[37] Oracle. C1 Visualizer. https://java.net/projects/c1visualizer, 2014.

[38] Oracle. Defining Client Access with Interfaces. http://docs.oracle.com/javaee/1.4/tutorial/doc/EJBConcepts6.html, 2014.

[39] Oracle. Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning. http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html, 2014.

[40] Oracle. Java VisualVM. http://docs.oracle.com/javase/6/docs/technotes/guides/visualvm/, 2014.

[41] Oracle. jmap - Memory Map. http://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html, 2014.

[42] Oracle. Memory Analyzer. http://www.eclipse.org/mat/, 2014.

[43] Oracle. Object Layout. https://wikis.oracle.com/display/MaxineVM/Objects#Objects-ObjectLayout, 2014.

[44] Oracle. Predefined Annotation Types. http://docs.oracle.com/javase/tutorial/java/annotations/predefined.html, 2014.

[45] Oracle. T1X: A Template-based Baseline Compiler. https://wikis.oracle.com/display/MaxineVM/T1X, 2014.

[46] Oracle. The History of Java Technology. http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html, 2014.

[47] Oracle. The Java Timeline. hhttp://oracle.com.edgesuite.net/timeline/java/, 2014.

[48] Oracle. The Structure of the Java Virtual Machine. http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html, 2014.

[49] Oracle. Thread Management. http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html, 2014.

[50] Oracle. Understanding Memory Management. http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html, 2014.

[51] Oracle. Using JConsole. http://docs.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html, 2014.

[52] Derek Rayside and Lucy Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 194–203, New York, NY, USA, 2007. ACM.

[53] Niklas Röjemo and Colin Runciman. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. *SIGPLAN Not.*, 31(6):34–41, June 1996.

[54] Esen Sagynov. The Principles of Java Application Performance Tuning. http://java.dzone.com/articles/principles-java-application, 2013.

[55] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. On models for object lifetime distributions. *SIGPLAN Not.*, 36(1):137–142, October 2000.

[56] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, January 2013.

[57] Guoqing Xu. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. *SIGPLAN Not.*, 48(10):111–130, October 2013.