A Profiling Tool for Exploiting the use of Packed Objects in Java Programs

by

Umang Umesh Pandya

Bachelor of Engineering, University of Mumbai, 2007

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

Master of Computer Science

In the Graduate Academic Unit of Faculty of Computer Science

Supervisor(s): Kenneth B. Kent, Ph. D, Faculty of Computer Science
Eric Aubanel, Ph. D, Faculty of Computer Science

Examinining Board: Gerhard Dueck, Ph. D, Faculty of Computer Science, Chair
David Bremner, Ph. D, Faculty of Computer Science
Yves Losier, Ph. D, Mechanical Engineering

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

January, 2015

©Umang Umesh Pandya, 2015
Abstract

Packed objects is an experimental feature in the IBM J9 Virtual Machine. Packed objects can be used to gain greater control over the layout of objects in memory. Applications which use packed objects have greater flexibility when they work with memory structures that are not in Java code. The purpose of developing this profiling tool is to assist software developers in determining how much performance and/or memory gain they can achieve if they switch to packed objects from standard Java objects. This switch can result in reduced memory consumption by the application since the use of packed objects in Java applications can reduce the amount of memory required to store objects and it also increases the efficiency of caching. If the application is accessing a lot of native data using JNI method calls, then use of the packed objects will eliminate marshaling/unmarshaling of native data into Java objects and thus eliminating redundant data copying, which is a chief limitation of existing methods used to access native data in Java applications such as the JNI API, and the Java New I/O API. The correctness and usefulness of the profiling tool is evaluated by running several test programs and benchmarks. Based on the results generated by the profiling tool for the benchmarks, the programs were modified to use packed objects instead of standard Java objects, and a performance gain was noticed in terms of reduced memory consumption by the program that allocates large array objects of non-primitive data types and by the program that uses the JNI functions to access arrays of primitive data types from the native side.
Dedication

I dedicate this thesis to my grandmother and my parents for their endless love, support and encouragement.
Acknowledgements

It's a pleasure to thank all the people who have helped me during my Master studies at the University of New Brunswick. I would like to thank my parents - Umesh Pandya and Usha Pandya, for their moral and financial support. Without their assistance I would not have been able to enroll in a graduate program at the University of New Brunswick and come to Canada.

I would like to thank my supervisors Kenneth Kent and Eric Aubanel, who have supported me throughout my thesis with their patience and immense knowledge. A special thanks goes to Karl Taylor from the IBM Ottawa Lab, who has been crucial in all stages of the development of this thesis and a constant source of inspiration. Furthermore, I would like to thank all my colleagues at the IBM CASA lab especially Aaron Graham, for their continuous support. Without their help this thesis would not have been possible.

I also gratefully acknowledge the financial support from the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program.

Finally, I would like to thank all my friends in Fredericton and back home in India for their continuous support and encouragement.
Umang Umesh Pandya
Table of Contents

Abstract ii
Dedication iii
Acknowledgments iv
Table of Contents ix
List of Tables xi
List of Figures xii
Abbreviations xiii

1 Introduction 1
   1.1 Motivation .............................................. 1
   1.2 Research Objectives ...................................... 2
   1.3 Thesis Organization ...................................... 2

2 Basics 4
   2.1 Packed Objects ........................................... 4
       2.1.1 Packed Object Overview ............................. 4
           2.1.1.1 Object Overhead ............................... 7
           2.1.1.2 Object Locality ............................... 9
       2.1.2 Packed Object Data Model .......................... 10
4.1.4.3 Set<PrimitiveType>ArrayRegion Routines 60
4.1.4.4 GetPrimitiveArrayCritical 61

5 Evaluation 63
5.1 Approach 63
5.2 Evaluation Method for Individual Cases 65
5.2.1 Case 1 - Fields of Primitive Data Types 65
5.2.2 Case 2 - Array Objects of Non-Primitive Data Type 67
5.2.3 Case 3 - Objects of Non-Primitive Data Type 70
5.2.4 Case 4 - Data Exchange Between Java Heap and Native Memory via JNI 72
5.3 Benchmarking 75
5.3.1 Employee Payroll Management System 76
5.3.2 A Java Program to Compress/Decompress a String 77
5.4 Results and Inferences 79
5.4.1 Comparison of the Size of the Array Object 79
5.4.2 Comparison of the Size of the Classes 84
5.4.3 Comparison of the Execution Time 86
5.4.4 Comparison of Performance of the Standard Java Object with the Off-Heap Packed Object 89

6 Conclusion and Future Work 92

Bibliography 96

A 99
A.1 Agent Start-up 99
A.2 java.crw.demo library 109
A.3 JNI Function Interception functions 114
A.4 Obtaining Stack Trace Information .......................... 116
A.5 Command Line Options ........................................ 118
A.6 Detailed Calculations for Case - 2 Test Case .............. 119
A.7 Detailed Calculations for Case - 3 Test Case .............. 122
A.8 Test Case for the Case - 4 ................................. 123

Vita
# List of Tables

4.1 List of Functions for GetPrimitiveTypeArrayElements Routines ..... 54  
4.2 List of Functions for GetPrimitiveTypeArrayRegion Routines ..... 59  
4.3 List of Functions for SetPrimitiveTypeArrayRegion Routines ..... 60  

5.1 The Size of Individual Classes and Objects in the Standard Java Model 81  
5.2 Dominator Tree for java.lang.Thread @ 0x680034988 main Thread ..... 81  
5.3 The Size of Individual Classes and Objects in the Packed Object Data Model ............................. 82  
5.4 Dominator Tree for java.lang.Thread @ 0x7fd859cdf098 main Thread 83  
5.5 Comparison of the Sizes of the Employee[] Object and * PackedEmployee[] Object .......................... 83  
5.6 Size of the Employee [] Object and PackedEmployee[] Object Returned by the Profiling Tool ................. 84  
5.7 Comparison of the Sizes of the Individual Classes in the Standard Java Model Obtained from the Memory Analyzer and the Sizes Estimated by the Profiling Tool ............................. 85  
5.8 Comparison of the Sizes of the Individual Classes in the Packed Object Data Model Obtained from the Memory Analyzer and the Sizes Estimated by the Profiling Tool ............................. 86  
5.9 Comparison of the Execution Time in the Standard Java Model and the Packed Object Data Model ............................. 87
5.10 Comparison of the Execution Time in the Standard Java model and the Packed Object Data Model with the JIT Turned Off ........ 88

5.11 Comparison of the Process Physical Memory in the Standard Java model and the Packed Object Data Model .................. 90

5.12 Comparison of the Execution Time in the Standard Java Model and the Packed Object Data Model .................. 90
# List of Figures

2.1  Representation of an Array Object in the Standard Java Model  ... 5  
2.2  Representation of an Array Object in Packed Object Data Model  ... 6  
2.3  Representation of an Object in Standard Java Model  ... ... 9  
2.4  Representation of an Object in Packed Object Data Model  ... ... 9  
2.5  Fields of Primitive Data Types in Standard Java Model  ... ... 11  
2.6  Fields of Primitive Data Types in Packed Object Data Model  ... ... 11  
2.7  Example of Mixed Packed Object  ... ... 14  
2.8  Off-heap Packed Object  ... ... 15  

4.1  Flow Diagram of Sequence of Events when a New Array is Allocated  ... ... 32  
4.2  Flow Diagram for Detecting a Cycle  ... ... 44  
4.3  Flow Diagram of Sequence of Events When a New Object is Allocated  ... ... 48  
4.4  Flow Diagram for Adding an Array in the List  ... ... 58  

5.1  Class Diagram of the Employee Payroll Management System  ... ... 77  
5.2  Class Diagram of the Employee Payroll Management System using Packed Objects  ... ... 78
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>JVMTI</td>
<td>JVM Tool Interface</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>BCI</td>
<td>Bytecode Instrumentation</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machine</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>MB</td>
<td>Megabytes</td>
</tr>
<tr>
<td>ms</td>
<td>milliseconds</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Packed objects [1][2][3] is an experimental feature in the IBM J9 Virtual Machine. Packed objects can be used to gain greater control over the layout of objects in memory. Applications programmed using packed objects have greater flexibility when they work with memory structures that are coded in other languages than Java [4]. Packed objects can also optimize certain Java objects to reduce their size and improve the efficiency of access.

Memory that is allocated outside the Java heap is called native data. Many Java applications have to deal directly with native data. There are certain disadvantages associated with the existing methods used for accessing native data in Java applications, mainly marshalling/unmarshalling of native data into Java objects which results in redundant data copying. Using packed objects for native data access enable us to model native data structures in Java code. This Java model overlays directly onto the native memory, avoiding the costly operations of copying data to and from Java code for each data access operation.
Packed objects can be used in Java code to reduce the amount of memory required to store certain objects namely arrays. In the packed object data model, arrays occupy considerably less amount of space as packed arrays require minimum overhead. Also the elements of the packed array are stored in adjacent memory locations with no object headers intervening between them, which improves efficiency of caching.

1.2 Research Objectives

The purpose of developing this profiling tool is to assist software developers in determining where and how much memory gain they can achieve if they switch to packed objects from standard Java objects. This switch can result in reduced memory consumption by the application. Also if the application is accessing a lot of native data using JNI [5] method calls, then use of the packed objects will eliminate marshaling/unmarshaling of native data into Java objects and thus eliminating redundant data copying.

1.3 Thesis Organization

The thesis is organized into six chapters. Chapter 2 presents the background information required to understand the remaining parts of the thesis. This includes concepts and properties of Packed Objects. It also includes a brief introduction about the JVM Tool Interface (JVMTI)[13] and java_crw_demo library which is used for bytecode instrumentation purposes. Chapter 3 discusses the requirements of the project followed by the design decisions taken to meet those requirements. Chapter 4 explains the implementation details of the project. It starts with a brief overview of the overall program structure and followed by the implementation detail of the four cases. Chapter 5 presents the approach used for the evaluation of the profiling tool.
It includes evaluation methods used for individual cases as well as methods used for evaluating the overall performance of the profiling tool along with benchmarking results. Finally, chapter 6 looks at the possible future extensions of this research work.
Chapter 2

Basics

2.1 Packed Objects

2.1.1 Packed Object Overview

Conventionally with the Java programming language we use standard Java objects to implement different functionality as per our requirements. However, in certain situations the use of packed objects instead of standard Java objects enables a programmer to gain greater control over the layout of objects in memory.

In order to illustrate the difference between a standard Java object and a packed object, let us take an example of a packed array and see how its memory layout will be different from the standard Java array.

Example:

```java
public class A
{
    int x;
}
A[ ] arr = new A[4];
```
Figure 2.1: Representation of an array object in the standard Java model.

From the memory layout of the standard Java array, we can observe that the elements of the array are references that point to the actual values. This representation is good for the dynamic growth of the array; however, the elements of the array are scattered across the memory and it is not desirable from a caching point of view. Also, as a result of the array elements being references rather than actual values, this results in additional memory space requirements.

Now let's see how this array can be represented in the packed object data model and how it will be laid out in memory:

Example:

```java
@Packed
public final class A extends PackedObject {
    int x;
}
A[] arr = new A[4];
```

In the packed object data model, the elements of the array are embedded within the array and they are stored in contiguous memory locations, which results in improved
caching and also they require less memory space.

Applications programmed using packed objects have greater flexibility when they interact with native data (memory structures that are not in Java code).

Existing techniques used to access native data in Java applications are [6]:

1. **The Java Native Interface (JNI) API**: We can use JNI methods available in the JNI interface to copy data back and forth between native and Java data structures. There are also methods available which can leave the data entirely in native memory and read or write to it.

2. **The Java New I/O (NIO) API**: We can use this API to create java.nio.Buffer objects to interact with native memory.

3. **The sun.misc.Unsafe.class**: This class provides functions to read and write to arbitrary memory locations in native or Java code but the caller of these functions should have sufficient privileges.

These techniques are useful as they allow us to avoid writing our own native methods and the overhead of calling these methods. But there are some disadvantages associated with these existing techniques, such as:
- Marshaling/unmarshaling of native data into Java objects results in redundant data copying.
- In order to access native data structures, which consist of many fields using the above methods, we should know the offsets of each field and how to convert the binary representations into an equivalent Java format, which could be trivial for primitive data types, but for complex structures it would be a complicated task [6].

Using packed objects for native data access allows us to model native data structures, including arrays and nested structures, in Java code. This Java model overlays directly onto native memory, avoiding the costly operations of copying the data to and from Java code for each data access operation.

The use of packed objects in Java code allows us to reduce the amount of memory required to store objects and also increases the efficiency of caching.

2.1.1.1 Object Overhead

Every object in the JVM [14] has some metadata associated with it, which contains information like the java.lang.Class value represents the type of that object, the length of an array object, etc. The most common approach is to place this metadata at the start of the object, thereby creating an object header [6].

For a large or complex object, the size of the header is relatively insignificant. But for small objects, the size of the header can be a significant portion of the object's overall size. Consider a byte array that contains a single element:

```
byte [] a = new byte[1];
```
This object has only 8 bits of data, but the header must reference both the class (byte []) and the length of the array. So, if we consider a 32-bit architecture and assume that the size of the integer is 32 bits and size of the native pointer is 32 bits then it will take an integer (4 bytes) to represent the length of the array, the class reference requires another integer (4 bytes), so we end up with at least 8 bytes of metadata for a single 8-bit data value. On top of this, the JVM is likely to add additional bytes of padding to ensure that the subsequent objects in the heap starts on an aligned address. Therefore the total extra memory required for the 8-bit data value is quite significant. Every object allocated by the JVM has a similar overhead associated with it, so the more objects we have, the greater the effect is on the system resources [6].

The structure of Java arrays can exaggerate this overhead. For example, consider an array of Point objects that is used to describe coordinates in a two-dimensional space. An instance of Point object consists of two int values, x and y. Again if we assume the size of the integer to be 32 bits then each of the int values will require 32 bits, plus the object header. If we assume that the object header consists of only the class reference (again assuming the size of the native pointer to be 32 bits), then each Point instance consists of around 8 bytes of data and 4 bytes of extra overhead. Furthermore, if we consider an array of 10 Point objects then its header will be of 8 bytes (32 bit integer for representing the length of the array and 32 bit integer for the class reference) and its elements will be 10 object references (assuming 4 bytes each = 40 bytes in total). If each element of the array contains a unique Point object, then the total is 80 bytes of data with 88 bytes of additional overhead [6].

In the case of packed arrays, the elements of the array are packed within the array which is in contrast to a standard Java array, where the elements are references
that point to the actual values. So, if we create a packed array of Point objects then this array does not require 40 bytes of overhead, which is incurred by the 10 Point instances, or the 40 bytes of references to those instances. The only overhead required for this packed array is the header (4 bytes to represent the length of the array and 4 bytes for the class reference), and this overhead is the same for an array of 10 Points or 10,000 Points [6].

2.1.1.2 Object Locality

Modern hardware relies heavily on caching and prefetching to provide efficient data access [7]. Caching exploits the observation that the memory, that was accessed recently is most likely to be accessed in the near future, so keeping this recently accessed data in faster memory usually results in better performance. So in modern memory architectures, the data is cached in small blocks known as cache lines. The other observation that we can make is the data which is stored in sequence is most
likely to be accessed in the same sequence [6].

However, there is no guarantee that related objects will be close enough in memory to appear in the same block of cached memory. Some JVM configurations attempt to keep related objects close to each other in memory [10], but this result is not always possible. Even when the JVM is successful in placing related objects next to each other, these objects will be separated by the space required for object headers, possibly disrupting the benefit achieved by placing related objects next to each other. In the packed object data model, the fields of a data structure are embedded within that data structure, thus always placing the related objects in adjacent memory with no intervening object headers. This structure can therefore increase the efficiency of caching [6].

2.1.2 Packed Object Data Model

The packed object data model differs from the standard Java data model in the following ways.

2.1.2.1 Primitive data type fields

In the packed object data model, fields of primitive data types, such as byte or int, occupy the minimum amount of space necessary. For example, a field of type byte or boolean occupies a single byte, short or char occupies 2 bytes; int or float occupies 4 bytes and long or double occupies 8 bytes.

This model is in contrast to the generic Java object model where all primitive data type fields are stored in either 32-bits (for byte, short, int, float, char, and boolean data types) or 64-bits (for long and double data types). This model can be wasteful, especially when there are multiple fields of small primitive data types [6].
For example:

```java
@Packed
public final class RGBColour extends PackedObject {
   public byte red;
   public byte green;
   public byte blue;
}
```

In the generic Java object model, the memory layout of this class will be as follows:

```
0  4  8  12 15
   Header   red   green   blue
```

Figure 2.5: Fields of Primitive data types in the standard Java model.

But when we represent the RGBColour class with the packed object data model then red, green and blue objects will occupy only 1 byte each, as shown in figure 2.6.

```
0  4  5  6  7
   Header  r  g  b
```

Figure 2.6: Fields of Primitive data types in the Packed Object Data model.

### 2.1.2.2 Object type fields

A packed class can contain fields of non-primitive data types. A packed object that contains one or more instance fields of non-primitive data types (e.g. String) is known as a **mixed packed object** [6].
For example:

```java
@Packed
public final class Player extends PackedObject {
    /**
     * as it contains a field of type String, Player is an example
     * of mixed packed object
     */
    String name;
    int age;
    int average;
}
```

So, if we create an instance of class Player, then that object would be an example of a mixed packed object as it contains instance fields of primitive type (two ints `age` and `average`) and a non-primitive data type (`name`).

In a packed class, fields of packed type behave differently as an instance field of a packed type in a packed class does not result in a reference to another object, but instead the data fields of that type are embedded within the object. A field of this type is known as a **nested packed field**. The instance fields of non-primitive data types behave in the same way as in the standard Java model, as these fields contain a reference, to either the field value or null [6].

In the following example, the Line class is a packed class and consists of two fields, start and end, which are instances of packed class Point. So, the fields start and end are examples of nested packed fields, and an instance of the class Line will not be a mixed packed object.
Public final class Point extends PackedObject {
    /* Point consists of two fields of primitive type */
    int x;
    int y;
}

Public final class Line extends PackedObject {
    /* start and end are nested packed fields and Line is not a mixed packed object */
    Point start;
    Point end;
}

The fields of packed class Line are examples of nested packed fields and will be embedded within the Line object, and its layout in memory will look like:

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>start.x</td>
<td>start.y</td>
<td>end.x</td>
<td>end.y</td>
</tr>
</tbody>
</table>

Now let's look at a packed class that is of a type mixed packed object and how its layout will appear in memory.

For example:

Public final class Team extends PackedObject {
    /**
     * the player fields can be nested into Team, but since they
* are of mixed type, so the Team object also becomes
* a mixed packed object
*/
Player player1;
Player player2;
}

Figure 2.7: Example of mixed Packed Object.

All the data for the 2 Player fields are inlined, but the name fields are just pointers, whereas the values of age and average fields will be embedded within the Team object.

2.1.2.3 Arrays

Similar to the nested packed fields of mixed packed object, the elements of an array of a packed type are packed into the array. This is in contrast to a standard Java array, where elements of an array are references that point to the actual values.

For example, if we consider a packed array of 10 RGBColour objects, it uses only 30 bytes of data plus the overhead of the array (array object header and any padding added by the JVM). Whereas, an array of 10 RGBColour objects in the standard Java model would consist of 10 object references of 4 bytes each, plus 10 instances of the RGBColour class, each of which consists of 12 bytes of data plus an object
header. So, we can observe the benefit of data packing when we deal with a packed array [6].

2.1.2.4 Off-heap packed objects

We can create a packed object in native memory outside the Java heap, and these objects are known as **off-heap packed objects**. These objects are very small and mainly consist of a pointer to actual data [6].

If we create an instance of RGBColour in native memory as an off-heap packed object, it will look like this:

![Off-heap Packed Object](image)

**Figure 2.8: Off-heap Packed Object.**

2.2 JVMTI

We have used the Java Virtual Machine Tool Interface (JVMTI) for programming our profiling tool. JVMTI is a programming interface that allows software developers to create a software agent that can monitor and control Java programming language applications. It provides both a way to inspect the state and to control the execution of applications running in the JVM.
JVMTI is a two-way interface. A client of JVMTI is called an agent and can be notified of interesting occurrences through events. A JVMTI agent can also query and control the application through functions, either in response to events or independent of them [11].

Agents run in the same process as the application being examined and communicate directly with the JVM executing the application. This communication is through a native interface (JVMTI). The native in-process interface allows maximum control with minimum intrusion on the part of a tool. Typically agents are relatively compact and they can be controlled by a separate process that implements the bulk of a tool’s functionality without interfering with the target application’s normal execution.

JVMTI provides a wide range of functions for controlling and inspecting different aspects of the application, for example functions for memory management, thread management, heap management, class loading, object queries, method queries, etc. JVMTI also provides several events by which the JVM can notify the agent about interesting events such as classes loaded in the JVM, exceptions issued, objects allocated, objects freed, VM Start event, VM Initialization event, VM Death event, and data dump request. We have to set up event callback functions for every event that we use. These event callback functions can be used to perform bytecode instrumentation to collect information about user-allocated objects that cannot be collected using standard JVMTI events. JVMTI provides support for bytecode instrumentation, which is the ability to alter the JVM bytecode instructions which comprise the target program. Because the inserted agent code is standard bytecodes, the JVM can still run at full speed, optimizing not only the target program but also the instrumentation.
2.3 Bytecode Instrumentation

Bytecode instrumentation is a valuable technique for transparently enhancing the virtual execution environments of the program for purposes such as monitoring or profiling [15]. Bytecode instrumentation is a process where new functionality is added to a program by modifying the bytecode of a set of classes before they are loaded by the virtual machine [16]. There is a reason why functionality added by bytecode instrumentation is not implemented directly in the source code, because the main objective of the bytecode instrumentation process is not to add additional functionality per se, but to enhance a program temporarily during its execution to gather profiling data, monitor memory usage, etc.

2.3.1 java_crw_demo

The java_crw_demo library is a small C library that can be used to do very basic bytecode instrumentation (BCI) of class files. This is not an agent library but a general purpose library that can be used to perform some limited bytecode insertion [17].

The basic BCI that this library does includes [12]

1. On entry to the java.lang.Object init method (signature "(V)", an invokestatic call to tclass.obj_init_method(object); is inserted.
2. On any newarray type opcode (opcode for the new array object), immediately following it, the array object is duplicated on the stack and an invokestatic call to tclass.newarray_method(object); is inserted.
3. On entry to all methods, an invokestatic call to tclass.call_method(cnum,mnum); is inserted.
4. On return from any method (any return opcode), an invokestatic call to tclass.return
_method(cnum,mnum); is inserted.

The tclass is a tracker class written in Java, which contains instrumenting methods required to collect profiling data about events such as new object allocation, new array object allocation, method entry, method exit etc.
Chapter 3

Project Design

This chapter describes the overall design of this project. It begins with the requirements of this project in Section 3.1, which are based on the problem analysed in Chapter 2. This is followed by a brief description of the approach adopted to solve the problem discussed in Section 3.2. The hardware and software platforms required for the project is analysed in Section 3.3.

3.1 Requirements

The requirements of this project are formulated based on the data that the profiling tool needs to collect in order to analyse the Java program and also from the point of view of the Java programmer who will be using this profiling tool. The requirements are as follows:

- The approach must be transparent to the Java programmer.
- The VM changes must be kept to a minimum.
- The VM must continue to function as before as no changes were made to the VM code.
• The profiling tool should unintrusively collect the data about the Java application.

• The process of collecting the profiling data about the Java application should not cause the VM to crash.

• The profiling tool should evaluate the profiling data to generate results that emphasize the benefits of packed objects over standard Java objects.

It is important that the profiling tool complies with the Java Virtual Machine and Java Language specification, as it should not modify the behaviour of the JVM or any user program. Running the user program with the profiling tool will increase the overall execution time of the program, but since this exercise will be carried out once to get the profiling tool’s results, we can ignore it. The profiling tool should collect the data it needs for analysis without affecting the execution of the Java program. Furthermore, the programmer should not notice any of the changes that have been made except for the analysis presented by the profiling tool.

We have aimed to keep the changes made in the JVM to a minimum because of the dependency between different files of the source code of the JVM. Changes made in one file can create instabilities with other parts of the code, so we must be very careful. If changes are inevitable, then they are evaluated thoroughly for correctness before integrating them in the JVM code.

The profiling tool should not interfere with the execution of the user program in any way. The JVMTI agent runs in the same process and address space as the Java application being executed, and it communicates directly with the virtual machine executing the application to gather the data required for analysis. So, if the JVMTI agent contains a bug, it can easily crash a VM. It is important to free all the memory
allocated by the JVMTI agent, because if the agent is leaking memory then the VM will appear to have a memory leak. Also, if the JVMTI agent allocates too much memory then there is a possibility that a VM process might fail with an "out of memory" error message, so we have to be careful.

The profiling tool gathers the data it needs for analysis while the user program is executing. Once the program execution completes, the profiling tool should present meaningful data to the user, which should enable him/her to make an educated decision about switching to packed objects from standard Java objects. We have presented the results in the following ways:

- Comparing the approximate amount of memory occupied by the objects of non-primitive data type in the standard Java model to the packed object data model. This should give users an idea about how much memory they can save if they switch to packed objects.

- Listing the number of fields of type byte, boolean, char, and short in a class. Since fields of these primitive types occupy less memory in the packed object data model, users can figure out how much benefits they can achieve if they switch to packed objects.

- Listing the amount of data transferred between the Java heap and the native memory for JNI calls. Along with the amount of data, we also list theJNI method used, the Java method which called this JNI method, and the Java class which contains this Java method.

3.2 Approach

In order to meet the requirements described in Section 3.1, the general task of collecting the profiling data needs to be divided into several subtasks so that they can
be handled individually in a more efficient manner. After carefully studying the packed object data model explained in Section 2.1.2, we have identified the following four cases where switching to packed objects could result in some sort of performance gain:

- **Case 1 - Fields of Primitive Data Types:** In the packed object data model, fields of primitive data type, such as byte or int, occupy the minimum amount of space necessary. This model is in contrast to a generic Java object model where all primitive data type fields are stored in either 32-bits or 64-bits format. This model can be wasteful, especially when there are multiple fields of small primitive data type.

- **Case 2 - Array Objects of Non-Primitive Data Types:** The elements of a packed array are embedded into the array, which is in contrast to the standard Java array where the elements are references which point to the actual values. This approach results in a significant reduction in memory consumption, especially if the application uses large arrays or a large number of smaller arrays.

- **Case 3 - Objects of Non-Primitive Data Types:** A packed object that contains one or more instance fields of non-primitive data types (e.g. String) is known as a mixed packed object. In a packed class, fields of packed type behave differently, as an instance field of a packed type in a packed class does not result in a reference to another object, but instead the data fields of that type are embedded within the object. These fields are called nested packed fields and this property of packed objects is in contrast to the standard Java data model where if an object contains an object of another class as an instance field, then the field value contains a reference, or pointer, to the object of that class. The advantage of this property of packed objects is that nested packed
fields of a packed object reside next to each other in memory, which improves cache locality and results in faster data access.

- **Case 4 - Data Exchange between Java Heap and Native Memory via JNI:** Accessing native data in the standard Java data model is an expensive operation because of redundant data copying. The better option in these situations would be to use the off-heap packed objects, as the JNI methods to access the off-heap packed objects are now available.

We have used the appropriate JVMTI functions to collect the information about the classes as they are loaded in the VM [8]. This has allowed us to identify the candidates which display a specific pattern (behaviour) that matches with one or more of the four cases explained in Section 3.2. Once we identify a target class then we collect the relevant data for the relevant cases for further analysis. For case 1, we have to estimate the packability of a class by observing all fields of different primitive data types and sizes. Case 2 also involves estimating the packability of the class, but here we look for classes that uses large arrays or a large number of smaller size arrays. Since in Case 3 we are primarily dealing with fields of non-primitive data type (reference values) that are internal to another class, so we have taken visibility and reference data into account in order to determine the packability of a class. Collecting data for Case 4 required a different approach as we were trying to profile the data sent and received via JNI methods. This step involved instrumenting JNI APIs in such a way that it enabled us to monitor the data access. Once the necessary data is collected, an analysis of the application is presented to the user which will give information about the individual classes (classes which exhibited a pattern that matched one of the four cases) like number and size of fields of primitive and non-primitive data type, number and size of the arrays of non-primitive data types used by the Java program, the details about the JNI method calls used by the user program etc. This information will assist the user to make a decision whether
switching to packed objects from the standard Java objects would be beneficial for
their application.

Exactly how the profiling tool collects the data for the above four cases from the
user program will be explained in detail in Chapter 4.

3.3 Hardware

The JVM can run on various hardware and software platforms, so it was necessary
to select a setup which will be used for this research project. We have used the
computers supplied to the Center for Advanced Studies (CAS) laboratory by the
University of New Brunswick (UNB) and IBM. For this project we have used a
1.8 GHz 16 core/32 Thread Xeon, 4x E7520, Nehalem-based server. This server is
used for compiling as well as executing the entire project. The evaluation of the
project was also carried out on this server. This server can be accessed via a desktop
computer by using a Secure Shell (SSH) client like PuTTY [18].

3.4 Design Decisions

With the requirements for the profiling tool listed in Section 3.1 and the approach
adopted described in Section 3.2, we were finally able to take the critical design
decisions for this research project. The design decisions related to the software are
explained in this section.

3.4.1 Software

The hardware specified above features an Intel processor microarchitecture CPU,
therefore software compatible with this architecture is used in this project. The
hardware used by us runs on a CentOS (Community Enterprise Operating System)
Linux distribution. IBM is one of the Strategic Developer Members of Eclipse [19] and since Eclipse is widely used for large scale project development, it was decided to use the Eclipse IDE for this project.

We have used JVMTI for programming our profiling tool as JVMTI is a programming interface that allows software developers to create a software agent that can monitor and control Java programming language applications. JVMTI provides both a way to inspect the state and to control the execution of user programs running in the JVM. Also, JVMTI provides a wide range of functions and events for controlling and inspecting different aspects of the application as described in Section 2.2. Due to these reasons we have used JVMTI in our profiling tool to collect the data from the user application that it needs for analysis.

Since profiling tool uses different JVMTI functions to collect the data from the user program running on the JVM, we have used C programming language in this project as JVMTI is a native API written in C. Test programs which were used for initial evaluation of the profiling tool were written in Java.
Chapter 4

Implementation

This chapter explains the implementation details of the profiling tool. It starts with a detailed explanation of the implementation of all four cases in Section 4.1. Subsection 4.1.1 explains the approach used to find the fields of primitive data types in classes as they are loaded into the VM. Subsection 4.1.2 explains the approach used to estimate the size of the array objects of non-primitive data types in the Java heap. Similarly Subsection 4.1.3 explains the approach used to estimate the size of the objects of non-primitive data types in the Java heap. Finally, Subsection 4.1.4 explains the approach adopted to monitor the data exchange between the Java heap and the native memory using JNI APIs.

4.1 Implementation Details of Individual Cases

Before a JVMTI agent begins collecting the profiling data from the Java application running on the VM, the agent needs to be initialized. Also, all the registered JVMTI events must have a designated callback function to handle these events. The initialization steps to initialize the agent and the callback functions used are explained in detail in Appendix A.1.
4.1.1 Case 1 - Fields of Primitive Data Types

As explained in Section 2.1.2.1, the fields of primitive data types occupy a minimum amount of space in the packed object data model. This property of packed objects can be exploited specifically by fields of byte, boolean, char and short types, since they will occupy either 1 byte (byte and boolean) or 2 bytes (char and short), which is in contrast to the generic Java data model where they usually occupy 4 bytes. So here we are listing fields of byte, boolean, char and short types in a class. This can help the user to decide whether he/she sees fit to change a particular class to packed, if that class contains several fields of these primitive data types.

We have used JVMTI's `GetLoadedClasses()` to get all the classes loaded in the VM. `GetLoadedClasses()` returns an array of all the classes loaded in the VM.

Once we get an array of classes, we need a data structure to hold the class details. We have used the following data structure for this purpose:

```c
typedef struct {
    char *signature;
    int count;
    jfieldID list_fields[];
} ClassDetails;
```

This data structure is used to store the signature of the class, number of fields in the class and an array which stores the field IDs of all the fields in this particular class.

Once we have this array of classes, we go through each class in it and do the following:

1. Get the signature of this class using `GetClassSignature()` JVMTI function.
GetClassSignature() returns the JNI type signature and the generic signature of the class.

2. The next step is to get all the fields of this class. We have used GetClassFields() JVMTI function for this purpose. GetClassFields() returns a count of fields and a list of field IDs of a class. It returns only directly declared fields and not inherited fields.

3. Once we have the array which contains field IDs of all the fields of a class then we proceed to retrieve the field names and signature. Field ID is a number (jfieldID type) and is not very informative, whereas field name and signature are more useful for further processing. We have used the GetFieldName() JVMTI function for this purpose. GetFieldName() returns the name and signature of a field.

4. For our heuristic, static fields in the class are not useful as they belong to the class and not to the objects that are instantiated from the class. So for each field in the class, we find out the field modifiers to determine whether the field has a static modifier in it. We have used the GetFieldModifiers() JVMTI function for this purpose. GetFieldModifiers() returns the access flags of a field. The value of the access flags is a mask of flags used to denote access permissions to and properties of this field.

5. From all the non-static fields of the class, we have to identify fields of byte, boolean, char and short types. This is done by comparing the signature of the field obtained earlier, with the signature of these primitive data types and incrementing the corresponding counters when a match is found.
6. Finally, this information about the class, i.e. class signature and counts for the fields of type byte, boolean, char and short, needs to be stored, so that it can be used in the end while presenting the user with the final analysis of the user's Java application. For this purpose, we have used a list to store information about the classes, where each node in the list contains information about a particular class, including the class signature, and counts for the fields of type byte, boolean, char and short.

The pseudo code for the above heuristic is shown in Algorithm 4.1.

4.1.2 Case 2 - Array Objects of Non-Primitive Data Type

As explained in Section 2.1.2.3, the elements of an array of a packed type are embedded into the array. This behaviour is in contrast to a standard Java array, where elements of an array are references which point to the actual values. This property of packed arrays is only applicable to arrays of non-primitive data types, because the arrays of primitive data types behave in the same way as they do in the generic Java model. So we cannot gain any benefits from the arrays of primitive data types if we switch to packed objects. But for arrays of non-primitive data types, switching to packed objects can be useful, as packed arrays occupy less space in memory and the elements of packed arrays are placed next to each other in memory, which improves caching and the rate at which elements can be accessed from the arrays.

In this case we have tried to gather information about each array allocation (non-primitive data types) in the Java program. When an array is allocated, we calculate the approximate size it occupies in the Java heap in the generic Java model. We then calculate the approximate size it will occupy in the packed object data model
Algorithm 4.1

for each class in classes[] do
    get class signature
    get class fields
    for each field in fields_list_ptr[] do
        get field name and signature
        get field modifiers
        if (field.mod not static) and (field.sig == "B") then
            byte_count++
        else if (field.mod not static) and (field.sig == "Z") then
            boolean_count++
        else if (field.mod not static) and (field.sig == "C") then
            char_count++
        else if (field.mod not static) and (field.sig == "S") then
            short_count++
        end if
    end for
    insert the class signature and corresponding counts to the list
end for

and compare it with the non-packed value to see if the array object will occupy less memory in the packed object data model. By comparing these two values, the user can decide whether switching to packed objects for this particular class (class of the array object) would be useful or not.

For tracking the allocations of arrays, we need a different approach. We cannot use static analysis like we did in Case 1, because we do not know if an array, which is declared in a class, is instantiated once, a hundred times or never. A more effective
approach is to profile the array allocation sites directly. By successfully profiling the array allocation sites, we can obtain information about each array allocation and use this information to calculate the approximate size that the array object will occupy in the generic Java model and the packed object data model.

To achieve this, we need to perform bytecode instrumentation. After looking at several libraries that can perform bytecode instrumentation, we decided to use the java_crw_demo library to perform bytecode instrumentation. The java_crw_demo library is a small C library that can be used to do some basic bytecode instrumentation (BCI) of class files.

A detailed description of the java_crw_demo library and its functions along with an explanation about how it is used in our profiling tool is given in Appendix A.2. BCI is performed by the java_crw_demo library by inserting an invokestatic call to the static methods declared in the Tracker class. For arrays, whenever a newarray type opcode is encountered, an invokestatic call to the tclass.newarray.method() is inserted and the array object is passed to this method. We have defined a class - ProfilingTool.class, which acts as the Tracker class and contains the newarr() method that performs the instrumentation. The newarr() method in turn calls a native method _newarr() which collects the data about each array allocated of non-primitive data types that we require for further analysis.

These sequence of events are described in the flow diagram in Figure 4.1. So, whenever a new array is created, the newarr() method is called and the array object is passed as an argument to it. In order to find more information about this array, like the array class, length of the array, etc., we need to use functions provided by JNI and JVMTI. It is more convenient to use these functions inside a native function, so
the newarr() method calls a native function _newarr(). Inside the newarr() method, first we check whether the array object is of primitive or non-primitive type. Since arrays of primitive type behave in the same way in the packed object data model as
they do in the generic Java model, we call the `newarr()` native function only if the array object is of a non-primitive data type.

Estimating the size of the class of the array object is essential in further calculations. We cannot use JVMTI's `GetObjectSize()` function to get the size of the array objects of non-primitive data types, because for objects of non-primitive data types, `GetObjectSize()` does not return the actual size occupied by the object on the heap. The steps involved in calculations and their significance are explained in the following subsections.

4.1.2.1 Calculations

The size of a class depends on the individual size of its fields. The size of a field is platform dependent i.e. 32-bit VM or 64-bit VM and it is also dependent on the VM configuration. For the calculations shown in this section, we have assumed that the programs are being executed on a 32-bit VM. Also, we have assumed following sizes for the primitive data types and other parameters used in calculations.

**Generic Java Model**

byte, boolean, char, short, int, float = 4 bytes
long, double = 8 bytes
Object header size = 4 bytes
Array header size = 8 bytes
Object pointer size = 4 bytes

**Packed Object Data Model**

byte, boolean = 1 byte
char, short = 2 bytes
int, float = 4 bytes
long, double = 8 bytes
Packed object header size = 12 bytes
Packed array header size = 16 bytes
Object pointer size = 4 bytes

Non-Packed Calculations
In order to calculate the size of an object in the generic Java model, we need to calculate the following set of values:

• **DataSize(object)** - This value takes into account the individual size of all the fields of a class to which the object belongs.

• **ObjectSize(object)** - This value is obtained by adding the size of the object header to the DataSize(object) value.

• **TotalObjectSize(object)** - This value represents the total size occupied by the object on the heap and it is same as the ObjectSize(object) value.

In order to calculate the size of an array object in the generic Java model, we need to calculate the following additional values once we obtain the length of the array.

• **DataSize(object[])** - Since in the standard Java model, the elements of the array are references that points to the actual objects, this step accounts for the size of these pointers. This value is obtained by multiplying the size of the object pointer with the length of the array.

• **ObjectSize(object[])** - This value is obtained by adding the size of the array header to the DataSize(object[]) value.

• **TotalObjectSize(object[])** - This value represents the total size occupied by the array object on the heap. So, while calculating this value we have to take
into account the size of the array object i.e. $\text{ObjectSize(object[])}$ and the size of the objects which the actual elements of the array i.e $\text{TotalObjectSize(object)}$ multiplied by the length of the array.

So, if we have to calculate the size of an array object, first we will need to calculate the size of the class to which the array object belongs. The steps involved are:

$\text{DataSize(object)}$ = obtained by adding the size of each field of the class to which the array object belongs

$\text{ObjectSize(object)}$ = $\text{DataSize(object)}$ + size of the object header

$\text{TotalObjectSize(object)}$ = $\text{ObjectSize(object)}$

So, for example, if we have a Point class which contains two fields of int type, then these values will be calculated in the following way:

```java
class Point {
    public int x;
    public int y;
}
```

$\text{DataSize(Point)}$ = 4 bytes + 4 bytes = 8 bytes

$\text{ObjectSize(Point)}$ = 8 bytes + 4 bytes = 12 bytes

$\text{TotalObjectSize(Point)}$ = 12 bytes

But in the VM, total object size must be at least 16 bytes, so we add a padding of 4 bytes. So we get

$\text{TotalObjectSize(Point)}$ = 16 bytes

Once we have the $\text{TotalObjectSize(object)}$, we can proceed to calculate $\text{TotalOb-}
jectSize(object[ ]), i.e. total memory occupied by the array object on the heap. In
an array of non-primitive data types, the elements of the array are actually pointers
that point to the actual objects, so in the first step of the calculations we account
for these pointers.

DataSize(object[ ]) = size of the object pointer \* number of elements of the array

In the next step, we add the size of the array header to the DataSize(object[ ]).

ObjectSize(object[ ]) = DataSize(object[ ]) + size of the array header

Since in an array of non-primitive data types, the elements of the array are ac­
tually pointers that point to the actual objects, so in this step we account for the
objects which are the actual elements of the array. We add their sizes to the Object-
Size(object[ ]) to get the total size of the array object.

TotalObjectSize(object[ ]) = ObjectSize(object[ ]) + (TotalObjectSize(object) * 
number of elements of the array)

For example, if we have an array of Point class of length 10, then

DataSize(Point[10]) = 4 bytes \* 10 = 40 bytes

ObjectSize(Point[10]) = 40 bytes + 8 bytes = 48 bytes

TotalObjectSize(Point[10]) = 48 bytes + (10 \* 16 bytes) = 208 bytes

Packed Calculations

The second set of calculations gives us the size occupied on the heap by the array
object in the packed object data model. Again, we need to calculate the following
set of values to calculate the size of an object in the packed object data model:

- PackedDataSize(object) - This value takes into account the individual size
  of all the fields of a class to which the object belongs in the packed object data
model.

- **PackedObjectSize(object)** - This value is obtained by adding the size of the packed object header to the PackedDataSize(object) value.

- **TotalPackedObjectSize(object)** - This value represents the total size occupied by the object on the heap and it is same as the PackedObjectSize(object) value.

In order to calculate the size of an array object in the packed object data model, we need to calculate the following additional values once we obtain the length of the array.

- **PackedDataSize(object[])** - Since in the packed object data model, the elements of the packed array are actual objects rather than references to these objects, in this step we account for the size of the fields of these objects. In the packed array, the elements of the array are placed next to each other with no object header intervening between them, so while calculating this value we use PackedDataSize(object) value and not TotalPackedObjectSize(object) value.

- **ObjectSize(object[])** - This value is obtained by adding the size of the packed array header to the PackedDataSize(object[]) value.

- **TotalObjectSize(object[])** - This value represents the total size occupied by the packed array object on the heap, and in this case it will be same as the PackedDataSize(object[]) value.

So, if we have to calculate the size of an array object in the packed object data model, first we will need to calculate the size of the class to which the array object belongs. The steps involved are:

\[
\text{PackedDataSize(object)} = \text{sum of size (in packed object data model) of each}
\]
field of the class to which the object belongs

\[ \text{PackedObjectSize}(\text{object}) = \text{PackedDataSize}(\text{object}) + \text{Packed object header size} \]

\[ \text{TotalPackedObjectSize}(\text{object}) = \text{PackedObjectSize}(\text{object}) \]

So, again, if we take Point class as an example, then these values will be calculated in the following way:

\[ \text{PackedDataSize}(\text{object}) = 4 \text{ bytes} + 4 \text{ bytes} = 8 \text{ bytes} \]
\[ \text{PackedObjectSize}(\text{object}) = 8 \text{ bytes} + 12 \text{ bytes} = 20 \text{ bytes} \]
\[ \text{TotalPackedObjectSize}(\text{object}) = 20 \text{ bytes} \]

Another padding rule states that the total object size must be aligned to 8, so in this case the total object size will be:

\[ \text{TotalPackedObjectSize}(\text{object}) = 24 \text{ bytes} \]

Once we have the \( \text{TotalPackedObjectSize}(\text{object}) \), we can proceed to calculate \( \text{TotalPackedObjectSize}(\text{object}[\ ] \) i.e. the total memory occupied by the array object on the heap in the packed object data model.

The key difference between \( \text{TotalObjectSize}(\text{object}[\ ]) \) and \( \text{TotalPackedObjectSize}(\text{object}[\ ]) \) is that in the packed arrays, the elements of the arrays are embedded within the array. The fields of the array object elements are placed next to each other with no object header intervening between them. This is accounted for in the following way:

\[ \text{PackedDataSize}(\text{object}[\ ]) = \text{PackedDataSize}(\text{object}) \times \text{number of elements of the array} \]
The above calculation step simulates the effect of in-lining all the elements of the array.

In the next step, we add the size of the packed array header to the PackedDataSize(object[]).

\[
PackedObjectSize(object[]) = PackedDataSize(object[]) + \text{Packed array header size}
\]

Finally, we calculate the total size of the array object in the packed object data model, which would be the same as the PackedObjectSize(object[]).

\[
TotalPackedObjectSize(object[]) = PackedObjectSize(object[])
\]

For example, if we have an array of Point class of length 10, then

\[
PackedDataSize(Point[10]) = 8 \text{ bytes} \times 10 = 80 \text{ bytes}
\]

\[
ObjectSize(Point[10]) = 80 \text{ bytes} + 16 \text{ bytes} = 96 \text{ bytes}
\]

\[
TotalObjectSize(Point[10]) = 96 \text{ bytes}
\]

### 4.1.2.2 Approach Used to Estimate Size of the Class

The pseudo code for the approach used to estimate the size of the class is shown in Algorithm 4.2. The same approach is used to estimate the size of a class in the generic Java model and the packed object data model, the only difference being the default size of the data types in these models.

The first step in estimating the size of the class is to get a list of its fields using JVMTI's `GetClassFields()` function. Similar to Case 1, the fields of static type are not applicable to our heuristic as they belong to the class and not the objects instantiated from the class. We use `GetFieldModifiers()` JVMTI function to get
the modifiers of a field. After that, for all the non-static fields of the class, we find
their data type by checking their signature. The size of each field is determined and
added to the overall size of the class in the following way.

• If a field is an object of any of the eight primitive data types then the default
  size of the data type is added to the overall size of the class.
• If a field is an array of any of the eight primitive data types then we ignore it and move on to the next field. Since the arrays of primitive types behave exactly in the same way in the packed object data model as they do in the generic Java model, we do not consider them while calculating the overall size of the class as their effect will be cancelled out while calculating TotalObjectSize() and TotalPackedObjectSize(). We can determine the data type of the primitive arrays from the signature of the field, but it is difficult to determine the length of the array. This is because arrays can be declared either in a static way or in a dynamic way, as shown below:

```java
int[] a = new int[10]; \static declaration

int[] a; \dynamic declaration
a = new int[10]; \size defined somewhere else
```

Finding out the length of the array, which is declared in a dynamic way requires very high-level bytecode instrumentation. So, as the size of these arrays of primitive types will be equal in the packed object data model and the generic Java model, we have ignored their impact on the overall size of the class.

• If the field is a String object or a String literal, then we only consider the size of the pointer that points to the actual String object. The reason behind this logic is that the size of this String object will be the same in the packed object data model and the generic Java model. So, we ignore the size of the String object.

• If the field is an object of some class, in that case we need to estimate the size of this field class and add it to the overall size of the original class. This is done by making a recursive call to the functions used to calculate the size of a class in the generic Java model and the packed object data model. But we have to be careful before making this recursive call, especially in the following
situations:

1. If a class contains an object of itself as an instance field, then calling the functions used to calculate the size recursively will cause the program to enter an infinite loop. This is explained in the following example:

   ```java
   class LinkedListNode
   {
       LinkedListNode next;
       Object data;
   }
   ```

   The field count for this class will be 2 and their signatures will be:

   LLinkedListNode;
   Ljava/lang/Object;

   So, when we evaluate the signature of the first field, it will result in a recursive call to estimate the size of the LinkedListNode class, and again the same procedure will be repeated and the program will enter an infinite loop.

   To avoid this, we compare the signature of the class of the object with the signature of the class whose size we are estimating. If the object is of the same class, then we ignore it and move on to the next field.

2. We may encounter the following scenario when the program will enter an infinite loop again.

   ```java
   class A{
       private B b;
   }
   class B{
       private C c;
   }
   ```
So, while estimating the size of the class A, we obtain a list of its fields: LB; i.e. object of class B.

So, now we make a recursive call to the functions to estimate the size of the class B, then we will get a list of its fields: LC; i.e. object of class C.

So, now we make another recursive call to the function to estimate the size of the class B, which will give us a list of its fields: LA; i.e. object of class A.

This results in another call to the functions to estimate the size of the class A, but we are already in the process of estimating the size of class A, so we have now entered in a cycle which will cause the program to enter into an infinite loop.

To avoid these and other similar scenarios, we have implemented the following solution:

Initially, when a class is seen for the first time, we create a new node for it in the list. This node contains the signature of the class and the size of the class, which initially is set as -1. If the class does not contain a cycle, then we would be successful in calculating its size and updating it in the list. But if a class does contain a cycle, then its size will still be set to -1. So, while estimating the size of some other class, if it contains a field which is an object of some class, we check if this class has been seen before or not. If it has been seen before, then we check its size from the list. If the size is anything other than
Search the class in the list contains the classes seen so far

Has this class been seen before?

No

estimate the size of this class by making a recursive call to the functions used to calculate the size of the class

Yes

check the size of the class

Is class size -1?

No

get the size of the class and use it in the calculations

Yes

This class contains a cycle, keep its size -1 for future detections

Figure 4.2: Detecting a cycle.

-1, then we use that value for further calculations, otherwise we know that this class is part of a cycle and we ignore this field and move on to the next field.

- If a field is an array object of a non-primitive data type, then we ignore it and move on to the next field. As explained earlier, arrays can be declared in a static or dynamic way, as shown below:

```java
Point[] p = new Point[10]; //static declaration
```
To find out the length of the array, which is declared in a dynamic way requires very high-level bytecode instrumentation. So, in our heuristic we have not considered the impact of the arrays of non-primitive data types on the overall size of the class.

**Effect of Inheritance**

In order to improve the accuracy of our estimation of the size of the class, we have accounted for the impact a Superclass can have on the size of the subclass via inherited fields.

To include this impact, we have used JNI’s GetSuperClass() function. This function returns the object that represents the Superclass of the specified class. So whenever a new array object is allocated, it is passed to the `newarr()` along with the class (jclass) of the array object. We obtain the Superclass of the array object’s class using GetSuperClass(), and then call the functions used to estimate the size of the class to estimate the size of the Superclass and add it to the size of the array object’s class. Similarly, if a class contains an object of some class as its field, then we find Superclass of this object’s class and add its size to the size of the original class.

**Effect of Padding**

The padding rules used by us are:

- Total object size must be at least 16 bytes. So, once we calculate `TotalObjectSize(object)` and `TotalPackedObjectSize(object)`, we check if these values are at least 16 bytes, if not then we add the required number of bytes.
to make them equal to 16 bytes.

- Total object size must be a multiple of 8. So, once we calculate \( \text{TotalObjectSize(object)} \), \( \text{TotalPackedObjectSize(object)} \), \( \text{TotalObjectSize(object[ ])} \) and \( \text{TotalPackedObjectSize(object[ ])} \), we check if these values are multiple of 8, if not then we add bytes to these values accordingly.

- Object pointer slots must be naturally aligned. For example, a packed object that contains a byte and a string as its fields requires 8 bytes of data, i.e. 1 for the byte field, 3 to align the object pointer and 4 for the pointer which points to the String object. We go through the field list of the class and for every field that contains an object pointer, we check its preceding field and, if padding is required to align the object pointer, then we add the padding accordingly.

Once we get the values \( \text{TotalObjectSize(object[ ])} \) and \( \text{TotalPackedObjectSize(object[ ])} \), we add these values along with the signature of the array object in a list. This list will contain all the array objects of non-primitive data types allocated by the Java program along with the size they occupy in the generic Java model and the packed object data model. We have also included length of the array and a count for the array objects. This count value indicates how many times an array of this particular class of the same length was instantiated. So, for example, if we consider an array of Point class of size 10, then its entry in the list will look like:

```
[LPoint; , 208, 96, 10, 1
```

### 4.1.3 Case 3 - Objects of Non-Primitive Data Types

As explained earlier in Section 2.1.2.2, A packed class can contain fields of non-primitive data types. A packed object that contains one or more instance fields of non-primitive data types (e.g. String) is known as a **mixed packed object**. In a
packed class, fields of packed type behave differently as an instance field of a packed
type in a packed class does not result in a reference to another object, but instead
the data fields of that type are embedded within the object. A field of this type is
known as a **nested packed field**. The instance fields of non-primitive data types
behave in the same way as in the standard Java model, as these fields contain a
reference, to either the field value or null.

Due to these properties of the packed objects, their fields will be placed next to
each other in the memory which will increase the efficiency of caching and the rate
at which the data can be accessed. Also packed objects may occupy less space in
memory as compared to space occupied by the same object in the generic Java model.

In order to track the allocation of objects of non-primitive data types in a Java
program, we have to again use BCI to profile the object allocation sites directly.
**java_crw_demo** library provides a capability where on entry to the java.lang.Object
init method (signature "()V"), an invokestatic call to the static method **newobj()**
defined in the Tracker class **ProfilingTool** is inserted. In order to collect the informa-
tion about this object allocation, we need to use different JVMTI and JNI functions
which can only be called from native code. So we call a native function **.newobj()**
from the newobj() method and pass the newly created object as an argument to this
native function. This sequence of events is described in the flowchart of Figure 4.3.

The process of estimating the size of the class of the newly allocated object is similar
to the one we used while estimating the size of the class of the new array object. Our
aim is to find how much space is occupied by the object in the generic Java model
and in the packed object model, so that we can compare these two values to decide
if switching this class to packed can be useful or not.
Figure 4.3: Sequence of events when a new object is allocated.
4.1.3.1 Non-packed Calculations

The first set of calculations gives us the space occupied on the heap by the object in the generic Java model. The first step is to estimate the total size of the object. The steps involved are:

\[
\text{DataSize}(\text{object}) = \text{sum of the size of each field of the class to which object belongs to}
\]

\[
\text{ObjectSize}(\text{object}) = \text{DataSize}(\text{object}) + \text{Object header size}
\]

\[
\text{TotalObjectSize}(\text{object}) = \text{ObjectSize}(\text{object})
\]

So, for example, if we have a Color class which contains four fields of byte type, then these values will be calculated in the following way:

```java
class Color {
    public byte r;
    public byte g;
    public byte b;
    public byte w;
}
```

\[
\text{DataSize(Color)} = 4 \text{ bytes} + 4 \text{ bytes} + 4 \text{ bytes} + 4 \text{ bytes} = 16 \text{ bytes}
\]

\[
\text{ObjectSize(Color)} = 16 \text{ bytes} + 4 \text{ bytes} = 20 \text{ bytes}
\]

\[
\text{TotalObjectSize(Color)} = 20 \text{ bytes}
\]

But in the VM, the total object size must be aligned to 8, so we add a padding of 4 bytes. So we get

\[
\text{TotalObjectSize(Color)} = 24 \text{ bytes}
\]

4.1.3.2 Packed Calculations

The second set of calculations gives us the space occupied on the heap by the object in the packed object data model. Again, the first step is to estimate the total size
of the object in the packed object data model. The steps involved are:

\[
\text{PackedDataSize}(\text{object}) = \text{sum of the size (in packed object data model) of each field of the class to which the object belongs to}
\]

\[
\text{PackedObjectSize}(\text{object}) = \text{PackedDataSize}(\text{object}) + \text{Packed object header size}
\]

\[
\text{TotalPackedObjectSize}(\text{object}) = \text{PackedObjectSize}(\text{object})
\]

So, for example, if we consider the Color class shown above, then these values will be calculated in the following way:

\[
\text{PackedDataSize}(\text{object}) = 1 \text{ byte} + 1 \text{ byte} + 1 \text{ byte} + 1 \text{ byte} = 4 \text{ bytes}
\]

\[
\text{PackedObjectSize}(\text{object}) = 4 \text{ bytes} + 12 \text{ bytes} = 16 \text{ bytes}
\]

\[
\text{TotalPackedObjectSize}(\text{object}) = 16 \text{ bytes}
\]

Once we calculate the values \text{TotalObjectSize}(\text{object}) and \text{TotalPackedObjectSize}(\text{object}), we add these values along with the signature of the object in a list. This list will contain all the objects of non-primitive data types allocated by the Java program along with the size they occupy in the generic Java model and the packed object data model. We have also included a count for the objects, and this count value indicates how many times an object of this particular class was instantiated.

So, for example, if we consider an object of Color class, then its entry in the list will look like:

\[
\text{LColor; } , 24, 16, 1
\]
4.1.4 Case 4 - Data Exchange Between Java Heap and Native Memory via JNI

Existing methods used to access native data in the Java applications such as the JNI API, the Java NIO API etc. have certain limitations, mainly they require marshaling/unmarshaling of native data into Java objects during every access operation, and this marshaling/unmarshaling causes the change in format of the data. Also, the location of the data changes, which results in redundant data copying.

Using the packed objects for the native data access allows us to model native data structures, including arrays and nested structures, in Java code. This Java model overlays directly onto native memory, avoiding the costly operations of copying the data to and from the Java code for each data access operation.

Use of packed objects in Java code allows us to reduce the amount of memory required to store objects and also increases the efficiency of caching. Also, applications programmed using packed objects have greater flexibility when they interact with native data.

As explained earlier in Section 2.1.2.4, we can create a packed object in the native memory and these objects are known as off-heap packed objects. Java interacts with these off-heap packed objects, which are very small in size and consists of a pointer to the actual data that resides in the native memory. A JNI API which includes operations on packed objects is now available. This extended API includes support for the following operations [1]:

- Allocating a new instance of a packed objects or array that references native data.
- Writing to a nested packed field.
- Reading and writing elements in a packed array.
- Obtaining a direct pointer to the data of a packed object or array.

In this case, we are tracking the data exchange between the Java heap and native memory via the JNI API, for array access operations. For each array access operation, we are recording the number of bytes accessed/modified. The actual cost of these array access operations would be even higher because after accessing data from the native memory we still have to marshall this data into a Java object, whose memory footprint would be higher, which increases the overall cost of this operation. But if we use packed objects instead, we can avoid overheads of data copying onto the Java heap from native memory, because in the case of off-heap packed objects data lies in native memory. The remainder of this section explains the methodology used to track these array access operations using the JNI API.

JVMTI includes functions for JNI function interception. These functions are:

- SetJNIFunction Table
- GetJNIFunction Table

An example illustrating the use of these functions to redirect JNI function calls is included in the Appendix A.3.

These functions provide the ability to intercept and resend the Java Native Interface (JNI) function calls by manipulating the JNI function table.

We have focused our heuristic on the JNI APIs which are used for array access operations. We are interested only in the JNI functions which are used to access or
modify elements of an array of primitive data types from the native code. So we will be focusing on the following sets of JNI functions:

- Get<PrimitiveType>ArrayElements Routines
- Get<PrimitiveType>ArrayRegion Routines
- Set<PrimitiveType>ArrayRegion Routines
- GetPrimitiveArrayCritical

Now, let us look at the methods used to collect the profiling data from these functions one by one.

### 4.1.4.1 Get<PrimitiveType>ArrayElements Routines

It contains a family of functions that returns the body of the primitive array. The result is valid until the corresponding Release<PrimitiveType>ArrayElements() function is called [5]. The Table 4.1 lists all the JNI functions which are part of this family of functions.

Since the cbVMStart function (callback for JVMI_EVENT_VM_START JVMTI event) is called when the VM starts, inside this function we have directed calls to the JNI function to our instrumentation function. This allows us to collect the data that we need for analysis before redirecting the JNI calls to the original JNI functions.

Once the call is directed to our instrumentation function, we collect the data from the parameters which are passed to these functions from the native code. For example, consider the instrumentation function for GetIntArrayElements JNI function i.e. MyGetIntArrayElements. The following steps are performed in order to determine the number of bytes transferred through the call to this JNI function:
<table>
<thead>
<tr>
<th>Get&lt;PrimitiveType&gt;ArrayElements Routines</th>
<th>Array Type</th>
<th>Native Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetBooleanArrayElements()</td>
<td>jbooleanArray</td>
<td>jboolean</td>
</tr>
<tr>
<td>GetByteArrayElements()</td>
<td>jbyteArray</td>
<td>jbyte</td>
</tr>
<tr>
<td>GetCharArrayElements()</td>
<td>jcharArray</td>
<td>jchar</td>
</tr>
<tr>
<td>GetShortArrayElements()</td>
<td>jshortArray</td>
<td>jshort</td>
</tr>
<tr>
<td>GetIntArrayElements()</td>
<td>jintArray</td>
<td>jint</td>
</tr>
<tr>
<td>GetLongArrayElements()</td>
<td>jlongArray</td>
<td>jlong</td>
</tr>
<tr>
<td>GetFloatArrayElements()</td>
<td>jfloatArray</td>
<td>jfloat</td>
</tr>
<tr>
<td>GetDoubleArrayElements()</td>
<td>jdoubleArray</td>
<td>jdouble</td>
</tr>
</tbody>
</table>

Table 4.1: List of Functions for GetPrimitiveTypeArrayElements Routines

1. We have used `GetObjectSize()` JVMTI function for estimating the size of the array accessed using `Get<PrimitiveType>ArrayElements Routines`, because for arrays of primitive data types, `GetObjectSize()` returns the exact size of the array including the overhead.

2. Once we get the size of the array object, we proceed to get the signature of the class to which the array object belongs. We have used JNI's `GetObjectClass()` function to get the class (jclass type) of the array object, and then, we have used JVMTI's `GetClassSignature()` function to obtain the class signature.

3. In order to find the name of the Java method and the Java class from which this JNI call was made, we are using JVMTI's `GetStackTrace()` function. An example illustrating the use of this function to obtain the name of the Java method and the Java class is included in the Appendix A.4.
4. We need to track each array of primitive data type which is being accessed using JNI’s array access API. In order to achieve this, we need a reference for each array (to be created on the native side), which can be used as a unique identifier for the array. In JNI, we can create three kinds of references for objects:

1. **Local Reference**: Local references are valid for the duration of a native method call. They are freed automatically after the native method returns [5].

2. **Global Reference**: Global references do not allow the underlying object to be garbage collected. Global references must be explicitly disposed of by calling `DeleteGlobalRef()` [5].

3. **Weak Global Reference**: Weak global references are a special kind of global reference. Unlike normal global references, a weak global reference allows the underlying Java object to be garbage collected [5].

We have used JVMTI’s `NewWeakGlobalRef()` to create a weak global reference for every array object of primitive data types. Inside the instrumentation function we create a weak global reference for the array object. If the same array object is accessed by two different JNI functions, then it will have the same weak global reference. This is very useful, since our final result will be based on the individual arrays i.e. how many times an array was accessed, the array was accessed using which JNI function, the array was accessed from which Java method and Java class, and the total data accessed/modified from this array (in bytes) using various JNI functions.

For this purpose, we have used a data structure which is used to store information about each array accessed using JNI functions. Each node in the linked list represents an array object.

```c
struct array_ref{
    jobject ref;
    int count;
};
```
Each node contains information about the array objects such as weak global reference, the count - number of times this particular array has been accessed, the total number of data bytes transferred between the Java heap and the native memory while accessing/modifying this particular array and a pointer to a linked-list which stores information regarding each individual array access operation. The data structure used for the inner linked list is shown below:

```c
struct array_details {
    char *jni_method;
    char *java_method;
    char *java_class;
    int data_bytes;
    int count;
    struct array_details *innerLink;
}
```

Every time a particular array is accessed using JNI's array access API, a node is created in this inner list. Here, each node contains information like the JNI function used to access the array, the Java method from which it was accessed, the Java class name where this Java method was declared, the number of bytes transferred between the Java heap and the native memory during this specific JNI call, the count - number of times this array was accessed from this specific Java method using the same JNI function.

Inside our instrumentation functions for the array access API once we collect the
data we need for our results (number of bytes accessed/modified), we insert the array object in the list. But before inserting this array object in the list, we check if this array object is already present in it. If a node corresponding to this array object already exists in the list then we update the node with the data collected from the latest JNI function call used to access the array. As explained above, each node in the main list contains its own inner list. This list is used to store information for each separate access of a particular array. So while adding a node in this inner list, we check if this array was accessed from this Java method declared in this Java class using this specific JNI function before, if it is then we update the node elements rather than creating a new node. If this particular array is accessed for the first time then a new node is created in the inner list. This is explained in the flowchart in Figure 4.4. Once we collect the data we need for our analysis, we redirect the JNI function call to the original JNI functions.

4.1.4.2 Get<PrimitiveType>ArrayRegion Routines

Get<PrimitiveType>ArrayRegion routines contains the family of functions that copies a region of a primitive array into a buffer [5]. Below Table 4.2 lists all the JNI functions which are part of this family of functions.

We have directed calls to these functions to our instrumentation functions inside the cbVMStart function (callback for VMStart JVMTI event). Once the call is directed to our instrumentation function, we collect the data from the parameters which are passed to these functions from the native code. For example, consider the instrumentation function for the GetIntArrayRegion JNI function i.e. MyGetIntArrayRegion. The following steps are performed in order to determine the number of bytes transferred through the call to this JNI function:
1. While using Get<PrimitiveType>ArrayRegion routines, the user has to specify start - the starting index and len - the number of elements to be copied. So we have used these values to calculate the number of elements of the array that will be accessed as:

\[ \text{data} = \text{len} - \text{start} \]

In order to find the number of bytes that were accessed, we multiply data with the default size of the primitive data types.
Get<PrimitiveType>ArrayRegion Routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Array Type</th>
<th>Native Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetBooleanArrayRegion()</td>
<td>jbooleanArray</td>
<td>jboolean</td>
</tr>
<tr>
<td>GetByteArrayRegion()</td>
<td>jbyteArray</td>
<td>jbyte</td>
</tr>
<tr>
<td>GetCharArrayRegion()</td>
<td>jcharArray</td>
<td>jchar</td>
</tr>
<tr>
<td>GetShortArrayRegion()</td>
<td>jshortArray</td>
<td>jshort</td>
</tr>
<tr>
<td>GetIntArrayRegion()</td>
<td>jintArray</td>
<td>jint</td>
</tr>
<tr>
<td>GetLongArrayRegion()</td>
<td>jlongArray</td>
<td>jlong</td>
</tr>
<tr>
<td>GetFloatArrayRegion()</td>
<td>jfloatArray</td>
<td>jfloat</td>
</tr>
<tr>
<td>GetDoubleArrayRegion()</td>
<td>jdoubleArray</td>
<td>jdouble</td>
</tr>
</tbody>
</table>

Table 4.2: List of Functions for GetPrimitiveTypeArrayRegion Routines

2. Once we get the number of bytes accessed, we obtain the signatures of the array object’s class, the name of the Java method and the Java class in the same way as we did for Get<PrimitiveType>ArrayElements Routines.

3. Since we are tracking each array when it is accessed using JNI’s array access API, we then create a weak global reference for the array object and insert it in the list in the same way as for the Get<PrimitiveType>ArrayElements Routines.

Once we collect the data we need for our analysis, we redirect the JNI function call to the original JNI functions.

4.1.4.3 Set<PrimitiveType>ArrayRegion Routines

Set<PrimitiveType>ArrayRegion Routines are the family of functions that copies back a region of a primitive array from a buffer [5]. Below Table 4.3 lists all the JNI
functions which are part of this family of functions.

<table>
<thead>
<tr>
<th>Set&lt;PrimitiveType&gt;ArrayRegion Routines</th>
<th>Array Type</th>
<th>Native Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetBooleanArrayRegion()</td>
<td>jbooleanArray</td>
<td>jboolean</td>
</tr>
<tr>
<td>SetByteArrayRegion()</td>
<td>jbyteArray</td>
<td>jbyte</td>
</tr>
<tr>
<td>SetCharArrayRegion()</td>
<td>jcharArray</td>
<td>jchar</td>
</tr>
<tr>
<td>SetShortArrayRegion()</td>
<td>jshortArray</td>
<td>jshort</td>
</tr>
<tr>
<td>SetIntArrayRegion()</td>
<td>jintArray</td>
<td>jint</td>
</tr>
<tr>
<td>SetLongArrayRegion()</td>
<td>jlongArray</td>
<td>jlong</td>
</tr>
<tr>
<td>SetFloatArrayRegion()</td>
<td>jfloatArray</td>
<td>jfloat</td>
</tr>
<tr>
<td>SetDoubleArrayRegion()</td>
<td>jdoubleArray</td>
<td>jdouble</td>
</tr>
</tbody>
</table>

Table 4.3: List of Functions for SetPrimitiveTypeArrayRegion Routines

We have directed calls to these functions to our instrumentation functions inside the cbVMStart function (callback for VMStart JVMTI event). Once the call is directed to our instrumentation function, we collect the data from the parameters which are passed to these functions from the native code. For example, consider the instrumentation function for SetIntArrayRegion JNI function i.e. MySetIntArrayRegion. The following steps are performed in order to determine the number of bytes transferred through the call to this JNI function:

1. While using Set<PrimitiveType>ArrayRegion routines, the user has to specify start (the starting index) and len (the number of elements to be copied). So we have used these values to calculate the number of elements of the array that will be accessed as:

\[
data = \text{len} - \text{start}\]
In order to find the number of bytes that were accessed, we multiply \( data \) with the default size of the primitive data types.

2. Once we get the number of bytes accessed, we obtain the signatures of the array object's class, the name of the Java method and the Java class in the same way as we did for the Get<PrimitiveType>ArrayElements Routines.

3. Since we are tracking each array when it is accessed using JNI's array access API, we then create a weak global reference for the array object and insert it in the list in the same way as for the Get<PrimitiveType>ArrayElements Routines.

Once we collect the data we need for our analysis, we redirect the JNI function call to the original JNI functions.

4.1.4.4 GetPrimitiveArrayCritical

This function is similar to the Get<PrimitiveType>ArrayElements functions. But there is a significant restriction on how this function can be used. After calling the GetPrimitiveArrayCritical function, the native code should not run for an extended period of time before it calls the ReleasePrimitiveArrayCritical function. We must treat the code inside this pair of functions as running in a **critical region**. Inside a critical region, native code must not call other JNI functions, or any system call that may cause the current thread to block and wait for another Java thread [5].

We have directed calls to this function to our instrumentation function inside the cbVMStart function (callback for VMStart JVMTI event). Once the call is directed
to our instrumentation function, we collect the data from the parameters which are passed to this function from the native code. The following steps are performed in order to determine the number of bytes transferred through the call to this JNI function:

1. We have used the `GetObjectSize()` JVMTI function for estimating the size of the array accessed using `GetPrimitiveArrayCritical`, because for arrays of primitive data types, `GetObjectSize()` returns the exact size of the array.

2. Once we get the number of bytes accessed, we obtain the signatures of the array object’s class, the name of the Java method and the Java class in the same way as we did for the `Get<PrimitiveType>ArrayElements` Routines.

3. Since we are tracking each array when it is accessed using JNI’s array access API, we then create a weak global reference for the array object and insert it in the list in the same way as for the `Get<PrimitiveType>ArrayElements` Routines.

Once we collect the data we need for our analysis, we redirect the JNI function call to the original JNI function.
Chapter 5

Evaluation

This chapter describes the testing and evaluation methods used to evaluate the implementation of the profiling tool. Section 5.1 explains the approach used for the evaluation of the profiling tool, as not only evaluating the implementation is necessary, but also the correctness and usefulness of the results generated by the profiling tool needs to be verified. Section 5.2 begins with the evaluation of the individual cases, and it includes the test programs used to verify the accuracy and correctness of the results for each individual case. Once the profiling tool has been evaluated thoroughly, it can be benchmarked in order to determine the usefulness of the results it generates. This is done in Section 5.3. Finally, the results of the benchmarks and their significance are discussed in Section 5.4.

5.1 Approach

In order to evaluate this project, the implementation of the profiling tool needs to be verified for correctness first. The first step in this process is to evaluate the implementation of the individual cases. This involves writing Java programs which will be used as test cases, and then comparing the results generated by the profiling tool with manually calculated values. This ensures that the profiling tool produces
correct output for each individual case.

In order to evaluate the accuracy and usefulness of the results produced by the profiling tool, it needs to profile a Java application which will be used as a benchmark. After looking at several available Java benchmarks, we came to the conclusion that they were not suitable candidates for benchmarking the profiling tool. The reason being that the profiling tool looks to exploit certain characteristics in a Java program like,

- Classes with many fields of primitive data types byte, boolean, char and short.
- A Java program that allocates larger arrays or a large number of smaller arrays of non-primitive data type.
- A Java program that extensively uses the JNI APIs to access/modify an array of primitive data types.

So, we decided to write Java applications that can be used as a benchmark for the profiling tool. These Java applications exhibit the characteristics mentioned above. Once, these Java applications were run with the profiling tool and the results produced by the tool were recorded, we measured certain parameters of the application such as execution time and memory usage. The next step in the evaluation process is to rewrite the classes identified by the profiling tool using packed objects instead of standard Java objects and measuring the execution time and memory usage. Due to the properties of packed objects, the memory usage of this modified Java program should be less than when compared to the memory usage of the Java program which uses standard Java objects. This performance gain [9] in terms of reduction in memory consumption can be compared to the memory gain estimated by the profiling tool. This comparison can be useful to evaluate the overall usefulness and correctness of the results produced by the profiling tool.
5.2 Evaluation Method for Individual Cases

The following subsections explain the methodology and the test cases used to verify the correctness and completeness of the results produced by the profiling tool for the individual cases.

5.2.1 Case 1 - Fields of Primitive Data Types

As described in Section 4.2.1, in this case we are listing the fields of type byte, boolean, char and short in a class. In order to verify the correctness of the results produced by the profiling tool, consider the following class and the corresponding output produced by the profiling tool.

```java
class TestCase1{
    private byte byte_field_1;
    private byte byte_field_2;
    private byte byte_field_3;
    private static byte byte_field_4;

    private boolean boolean_field_1;
    private boolean boolean_field_2;
    private boolean boolean_field_3;
    private static boolean boolean_field_4;

    private char char_field_1;
    private char char_field_2;
    private char char_field_3;
    private static char char_field_4;

    private short short_field_1;
    private short short_field_2;
    private short short_field_3;
}```
private static short short_field_4;
}

For the above test class, the detailed output produced by the profiling tool is as shown below:

```java
class signature :LTestCase1;
field count for this class is 16
byte_field_1       B
byte_field_2       B
byte_field_3       B
byte_field_4       B
boolean_field_1    Z
boolean_field_2    Z
boolean_field_3    Z
boolean_field_4    Z
char_field_1       C
char_field_2       C
char_field_3       C
char_field_4       C
short_field_1      S
short_field_2      S
short_field_3      S
short_field_4      S
this class has 3 fields of byte type
this class has 3 fields of boolean type
this class has 3 fields of char type
this class has 3 fields of short type
total count for this class is 12
```

We can observe in the above output that the fields of static type are not included as they are not useful in our heuristic as explained earlier.

Whereas, in the consolidated output, the entry for this class in the list is:
Several test cases were used to test the functionality of the profiling tool for case 1, and the results were found to be correct and complete.

5.2.2 Case 2 - Array Objects of Non-Primitive Data Type

As explained in Section 4.2.2, in this case we are estimating the size of an array object in the generic Java model and in the packed object data model. In order to verify the correctness of the results produced by the profiling tool for this case, let's look at the following test classes and the corresponding output produced by the profiling tool.

```java
class Point {
    public int x;
    public int y;
}
class Line {
    private Point start;
    private Point end;
}
class TestCase2 {
    Line[] l1 = new Line[10];
}
```

Now let us calculate the values \textbf{TotalObjectSize(Line[10])} and \textbf{TotalPackedObjectSize(Line[10])}, so that we can compare these values to the ones generated by the profiling tool for this particular array. The detailed calculations for this array object is included in Appendix A.6.

Non-packed Calculations:
By using the formulas described in Section 4.1.2.1, the following values are calculated.

\[
\begin{align*}
\text{DataSize(Point)} &= 8 \text{ bytes} \\
\text{ObjectSize(Point)} &= 12 \text{ bytes} \\
\text{TotalObjectSize(Point)} &= 16 \text{ bytes} \\
\text{DataSize(Line)} &= 40 \text{ bytes} \\
\text{ObjectSize(Line)} &= 44 \text{ bytes} \\
\text{TotalObjectSize(Line)} &= 48 \text{ bytes} \\
\text{DataSize(Line[10])} &= 40 \text{ bytes} \\
\text{ObjectSize(Line[10])} &= 48 \text{ bytes} \\
\text{TotalObjectSize(Line[10])} &= 528 \text{ bytes}
\end{align*}
\]

**Packed Calculations:**

By using the formulas described in Section 4.1.2.1, the following values are calculated.

\[
\begin{align*}
\text{PackedDataSize(Point)} &= 8 \text{ bytes} \\
\text{PackedDataSize(Line)} &= 16 \text{ bytes} \\
\text{PackedObjectSize(Line)} &= 28 \text{ bytes} \\
\text{TotalPackedObjectSize(Line)} &= 32 \text{ bytes}
\end{align*}
\]

Now we have to calculate \text{TotalPackedObjectSize(Line[10])}. The key difference between \text{TotalObjectSize(Line[10])} and \text{TotalPackedObjectSize(Line[10])} is that in the packed arrays, elements of the arrays are embedded within the array. The fields of the array object elements are placed next to each other with no object
header intervening between them.

PackedDataSize(Line[10]) = 160 bytes
PackedObjectSize(Line[10]) = 176 bytes
TotalPackedObjectSize(Line[10]) = PackedObjectSize(Line[10]) = 176 bytes

Now let us look at the output produced by the profiling tool for this array object.

An array is allocated of class Line
class signature: LLine;
field count of this class is 2
The signature of the 0th field is LPoint;
The signature of the 1th field is LPoint;

DataSize is 40 bytes
ObjectSize is 44 bytes before padding
ObjectSize is 48 bytes after padding
TotalObjectSize is 48 bytes

DataSize(array) is 40 in bytes
ObjectSize(array) is 48 in bytes
TotalObjectSize(array) is 528 in bytes

starting of packed calculations

class signature: LLine;
field count of this class is 2
The signature of the 0th field is LPoint;
The signature of the 1th field is LPoint;

PackedDataSize is 16 bytes
PackedObjectSize is 28 bytes before padding
PackedObjectSize is 32 bytes after padding
TotalObjectSize is 32 bytes

PackedDataSize(array) is 160 in bytes
PackedObjectSize(array) is 176 in bytes
PackedObjectSize(array) is 176 bytes after padding
TotalPackedObjectSize(array) is 176 in bytes

So, we can observe that the values TotalObjectSize(Line[10]) and TotalPackedObjectSize(Line[10]) returned by the profiling tool are identical to the values we calculated manually. Several other test cases were also used to test the functionality of the profiling tool for case 2, and the results were found to be correct and complete.

5.2.3 Case 3 - Objects of Non-Primitive Data Type

As explained in Section 4.2.3, in this case we are estimating the size of an object in the generic Java model and in the packed object data model. In order to verify the correctness of the results produced by the profiling tool for this case, let's look at the following test classes and the corresponding output produced by the profiling tool.

```java
class Point {
    public int x;
    public int y;
}

class Rectangle {
    private Point a;
    private Point b;
    private Point c;
    private Point d;
}

Rectangle r = new Rectangle();
```
Now let us calculate the values $\text{TotalObjectSize}(\text{Rectangle})$ and $\text{TotalPackedObjectSize}(\text{Rectangle})$, so that we can compare these values to the ones returned by the profiling tool. The detailed calculations for this array object is included in Appendix A.7.

**Non-packed Calculations:**

By using the formulas described in Section 4.1.3.1, the following values are calculated.

- $\text{DataSize}(\text{Rectangle}) = 80$ bytes
- $\text{ObjectSize}(\text{Rectangle}) = 84$ bytes
- $\text{TotalObjectSize}(\text{Rectangle}) = 88$ bytes

**Packed Calculations:**

- $\text{PackedDataSize}(\text{Rectangle}) = 32$ bytes
- $\text{PackedObjectSize}(\text{Rectangle}) = 44$ bytes
- $\text{TotalPackedObjectSize}(\text{Rectangle}) = 48$ bytes

Now let’s look at the output produced by the profiling tool for this object.

```plaintext
a new object is allocated
class signature of the new object is: LRectangle;

field count of this class is 4
The signature of the 0th field is LPoint;
The signature of the 1th field is LPoint;
The signature of the 2th field is LPoint;
The signature of the 3th field is LPoint;

DataSize is 80 bytes
ObjectSize is 84 bytes
```
starting of packed calculations

class signature: LRectangle;
field count of this class is 4
The signature of the 0th field is LPoint;
The signature of the 1th field is LPoint;
The signature of the 2th field is LPoint;
The signature of the 3th field is LPoint;

PackedDataSize is 32 bytes
PackedObjectSize is 44 bytes
TotalObjectSize is 48 bytes

So we can observe that the values TotalObjectSize(Rectangle) and TotalPackedObjectSize(Rectangle) returned by the profiling tool are identical to the values we calculated manually. Several other test cases were also used to test the functionality of the profiling tool for case 3, and the results were found to be correct and complete.

5.2.4 Case 4 - Data Exchange Between Java Heap and Native Memory via JNI

As explained in Section 4.2.4, in this case we want to track the data exchange between the Java heap and the native memory when an array of primitive data types is accessed via the JNI APIs. The code example in Appendix A.8 includes the Java code and the native library written in C, which includes the native methods that uses the JNI functions to access/modify arrays of primitive data types.

The Java program in Appendix A.8 calls the native methods defined in the testlib.c
library, to access arrays of primitive data types and perform some basic operations on them. The native methods defined in the testlib.c use the following JNI functions to access arrays:

- returnArrayMethod -> SetIntArrayRegion
- intArrayMethod -> GetIntArrayElements
- criticalIntArrayMethod -> GetPrimitiveArrayCritical
- intArrayRegionMethod -> GetIntArrayRegion

Now, let us calculate the number of bytes of data that would have been exchanged between the Java heap and the native memory when these native methods are called on from the Java code.

1. **returnArrayMethod -> SetIntArrayRegion**
   This native method returns an array of int type. The size of the returned array is determined by the parameter passed on to this method. It uses the **SetIntArrayRegion()** JNI function to set the elements of the array which will be returned from the native code to the Java code. So, if the value of the parameter passed to this method is 10, and if we follow the steps explained in Section 4.1.4.3, the number of bytes transferred can be calculated as **40 bytes** for this particular JNI call.

2. **intArrayMethod -> GetIntArrayElements**
   This native method calculates the sum of all the elements of the array which is passed to it. So, in the example code in Appendix A.8, the **intTest** array is passed to this method. It then uses the JNI function **GetIntArrayElements()** to get a copy of the array elements. So, the number of bytes transferred will be dependent on the length of the array, which in the example code is 10. So, the total number of bytes
transferred in this native method call will be 48 bytes i.e. the size of an int array of length 10.

3. criticalIntArrayMethod -> GetPrimitiveArrayCritical
This native method calculates the sum of all the elements of the array which is passed to it. The difference between this method and the intArrayMethod is the use of GetPrimitiveArrayCritical JNI function to get a copy of the elements of the array which is passed to it. When we use the GetPrimitiveArrayCritical(), the native code should not run for an extended period of time before it calls ReleasePrimitiveArrayCritical. We must treat the code inside this pair of functions as running in a critical region. So, the number of bytes transferred will be dependent on the length of the array, which in the example code is 10. So, the total number of bytes transferred in this native method call will be 48 bytes.

4. intArrayRegionMethod -> GetIntArrayRegion
This native method calculates the sum of the specified number of elements of the array which is passed to it. It uses the GetIntArrayRegion JNI function to get the number of elements starting from the start index to the len index. So the number of bytes transferred will be dependent on the value of these indexes, and these values are 5 and 10 respectively in the example code in Appendix A.8. So, if we follow the steps explained in Section 4.1.4.2, the number of bytes transferred can be calculated as 20 bytes for this particular JNI call.

Since all the above operations are performed on the same intTest array, the total number of data bytes transferred while accessing this particular array will be the addition of data bytes transferred during each individual operation.
Total number of databytes transferred while accessing `intTest` array = 40 + 48 + 48 + 20 = **156 bytes**

Now let us look at the output produced by the profiling tool for the example code of Appendix A.8.

```
[I,156,4
 SetIntArrayRegion,returnArrayMethod, LHelloWorld; ,40,1
 GetIntArrayElements, intArrayMethod, LHelloWorld; ,48,1
 GetPrimitiveArrayCritical, criticalIntArrayMethod, LHelloWorld
 ;,48,1
 GetIntArrayRegion, intArrayRegionMethod, LHelloWorld; ,20,1
```

So, from the above output we can observe that the total number of bytes transferred while accessing the `intTest` array returned by the profiling tool is identical to the value calculated by us manually. Several other test cases were also used to test the functionality of the profiling tool for Case 4, and the results were found to be correct and complete.

### 5.3 Benchmarking

This section describes the Java programs that are used as benchmarks for validating the profiling tool. As explained earlier, we decided not to use the available standard Java benchmarks, since they do not exhibit in abundance the characteristics that we are trying to exploit in Java programs, where using packed objects instead of standard Java objects would be beneficial. So, we have written custom Java programs that exhibit these characteristics i.e. allocating large arrays of non-primitive data types, accessing a large amount of native data using the JNI API, classes with many fields of type byte, boolean, char and short. The parameters that are being evaluated are the memory usage and execution time.
5.3.1 Employee Payroll Management System

This Java program is used to calculate the payroll of all the employees of a company and to manage the employee records. The class diagram of this program is shown in Figure 5.1. The Employee class contains objects of several other classes, which are used to store the records of an employee. The records of all the employees are stored in an array. The fields of the employee object are filled with random values generated with certain logical constraints. Once the fields are populated, the payroll for each employee is calculated and its value is updated in the Employee object. Several search queries are used to search specific fields of the Employee object. For example, find all the male employees working in the company, find all the employees with the blood group A, etc. The search queries return an array of employees which are a match for the field being searched.

After running the Employee Payroll Management program with the profiling tool, the results of the profiling tool are recorded. Based on the results, the next step in the evaluation process is to rewrite the classes identified by the profiling tool with packed objects instead of standard Java objects. The classes identified by the profiling tool are the ones where switching to packed objects can result in significant reduction in the memory consumption of the program. For the Employee Payroll Management program, if we create an array to hold records of 1,000,000 employees, the output of the profiling tool will contain the following entry for this array object:

```
[Employee; , 368000016, 204000032, 1000000, 1
```

In the above entry we can observe that as per the profiling tools estimation, we can reduce the size of this array object from 369 megabytes to 204 megabytes if we switch to packed objects. In order to verify this, we need to rewrite the program using packed objects. Since the Employee class contains objects of other classes, we need to rewrite these classes using packed objects as well. The class diagram
of this modified program is shown in Figure 5.2. This modified program performs the identical operation as the earlier program, the only difference being the use of packed objects.

### 5.3.2 A Java Program to Compress/Decompress a String

In order to validate the results produced by the profiling tool for case 4 i.e. tracking the JNI calls that are used to access/modiﬁe elements of the arrays of primitive data types, we have written a Java program which is used for compressing/decompressing a string using a native library. The native library is called SMAZ [20] and it uses the JNI function `GetCharArrayRegion()` to obtain the elements of the char array
which contains the string to be compressed in the Java code. Once it obtains the elements of the char array, it then proceeds to compress the input string. Since we are tracking calls to the JNI APIs that are used for accessing/modifying elements of the arrays of primitive data types, when the native library uses the GetCharArrayRegion() function to access the elements of the char array that contains the input string, this transaction between the Java heap and the native memory will be recorded by the profiling tool. For example, if the length of the input string is 512,000 characters, then when the native library uses the GetCharArrayRegion() function to access the elements of the input char array, then the profiling tool will have the following entry for this operation in its results:

Figure 5.2: Class diagram of the Employee Payroll Management System using Packed Objects.
In the above entry we can observe that a char array was accessed from the Java class `LSmaz_test`, that contains the calling Java method `smazCompress`, which in turn uses the JNI function `GetCharArrayRegion()` and a total 1,024,000 bytes were accessed.

As explained earlier in Section 2.1.2.4, when we use off-heap packed objects instead of standard Java objects in situations where we use the JNI functions to access/modify elements of the arrays defined in a Java program, we can eliminate the redundant data copying that is required because of the serialization/deserialization of the data between the Java heap and the native memory. In order to prove this claim, we have to rewrite the Java program using off-heap packed objects. The modified program performs the identical functions as the earlier program, except now the char array is defined as an off-heap packed object, so it will be stored in the native memory. Now, when we evaluate the performance of this program, its native memory consumption should be less than the earlier program that used standard Java objects.

### 5.4 Results and Inferences

#### 5.4.1 Comparison of the Size of the Array Object

In order to evaluate the benchmarks described in Section 5.3, we generate a system dump while the program is being executed by the VM. Later, we analyze this dump using the Memory Analyzer tool [21]. This tool presents a snapshot of the heap for the timestamp when the system dump was generated. Table 5.1 lists the heap size of the objects that belong to the Employee Payroll Management program. In order to get more accurate values of the Employee array object i.e. the array that is used
to hold the records of all the employees of a company, while generating the system
dump, we keep the reference to this array object alive and force a garbage collection.
This ensures that the objects which are dead and not relevant to our analysis are
garbage collected excluding the Employee array object.

Table 5.1 lists the shallow heap and the retained heap occupied by the objects on the
heap. Shallow heap refers to the size of an individual object in isolation, whereas the
retained heap includes all the objects that are referenced (and kept alive) by that
object. From Table 5.1, we can observe that there is one array object of Employee
type and its length is 1,000,000. Since this array object contains records of 1,000,000
employees, we can see that there are 1,000,000 employee objects on
the heap. The
employee object in turn contains references to other objects such as MedicalHistory
Information, DeductionInformation, BankingInformation, LeaveInformation, Personal
Information and EmergencyContactInformation. So, the count of objects of
these classes is also 1,000,000. From Table 5.1, we can note that the size of the array
object of Employee type is around 668 megabytes.

With the help of Memory Analyzer, we were able to find out the top consumer object
for the Employee Payroll Management program, and it is the java.lang.Thread
@ 0x680034988 main Thread. To find out the objects present in this thread, we
can look at the dominator tree for this thread in the Memory Analyzer and the
corresponding results are displayed in Table 5.2.

So, from Table 5.2 we can conclude that the top consumer object in this thread is
the array object of Employee type, and it occupies around 668 megabytes on the
heap.
<table>
<thead>
<tr>
<th>Class Loader/ Class</th>
<th>Objects</th>
<th>Shallow Heap (in bytes)</th>
<th>Retained Heap (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>1,000,000</td>
<td>72,000,072</td>
<td>664,001,712</td>
</tr>
<tr>
<td>MedicalHistoryInformation</td>
<td>1,000,000</td>
<td>48,000,048</td>
<td>48,000,669</td>
</tr>
<tr>
<td>DeductionInformation</td>
<td>1,000,000</td>
<td>32,000,032</td>
<td>32,000,463</td>
</tr>
<tr>
<td>BankingInformation</td>
<td>1,000,000</td>
<td>32,000,032</td>
<td>32,000,496</td>
</tr>
<tr>
<td>LeaveInformation</td>
<td>1,000,000</td>
<td>32,000,032</td>
<td>32,000,475</td>
</tr>
<tr>
<td>PersonalInformation</td>
<td>1,000,000</td>
<td>32,000,032</td>
<td>312,000,905</td>
</tr>
<tr>
<td>EmergencyContactInformation</td>
<td>1,000,000</td>
<td>24,000,024</td>
<td>80,000,503</td>
</tr>
<tr>
<td>Employee[]</td>
<td>1</td>
<td>4,000,008</td>
<td>668,000,136</td>
</tr>
<tr>
<td>Test</td>
<td>0</td>
<td>0</td>
<td>650,253</td>
</tr>
</tbody>
</table>

Table 5.1: The Size of Individual Classes and Objects in the Standard Java Model

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Shallow Heap (in bytes)</th>
<th>Retained Heap (in bytes)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Thread @ 0x680034988 main Thread</td>
<td>128</td>
<td>668,001,192</td>
<td>99.07%</td>
</tr>
<tr>
<td>Employee[1000000] @ 0x680540098</td>
<td>4,000,008</td>
<td>668,000,008</td>
<td>99.07%</td>
</tr>
</tbody>
</table>

Table 5.2: Dominator Tree for java.lang.Thread @ 0x680034988 main Thread

In order to compare the size of the array object of Employee type with the corresponding array object of PackedEmployee type in the packed object data model, we ran the modified Java program and generated a system dump and the corresponding results are listed in Table 5.3.

From Table 5.3, we can observe that there is one array object of PackedEmployee type (PackedEmployee$Array) and the length of this array object is also 1,000,000. Since, in the packed object data model the elements of the packed array are em-
Table 5.3: The Size of Individual Classes and Objects in the Packed Object Data Model

<table>
<thead>
<tr>
<th>Class Loader/Class</th>
<th>Objects</th>
<th>Shallow Heap (in bytes)</th>
<th>Retained Heap (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PackedEmployee$Array</td>
<td>1</td>
<td>224,000,032</td>
<td>224,001,244</td>
</tr>
<tr>
<td>PackedEmployee</td>
<td>1</td>
<td>248</td>
<td>4,703</td>
</tr>
<tr>
<td>PackedDeductionInformation</td>
<td>1</td>
<td>72</td>
<td>566</td>
</tr>
<tr>
<td>PackedPersonalInformation</td>
<td>1</td>
<td>64</td>
<td>1,503</td>
</tr>
<tr>
<td>PackedMedicalHistoryInformation</td>
<td>1</td>
<td>64</td>
<td>811</td>
</tr>
<tr>
<td>PackedBankingInformation</td>
<td>1</td>
<td>56</td>
<td>810</td>
</tr>
<tr>
<td>PackedEmergencyContactInformation</td>
<td>1</td>
<td>24</td>
<td>995</td>
</tr>
<tr>
<td>PackedTest</td>
<td>1</td>
<td>16</td>
<td>7,091,012</td>
</tr>
<tr>
<td>PackedLeaveInformation</td>
<td>0</td>
<td>0</td>
<td>553</td>
</tr>
</tbody>
</table>

bedded within the array object, there will be no outgoing references from the array object of PackedEmployee type to the objects of PackedEmployee type, and this is reflected in Table 5.3. From Table 5.3, we can note that the size of the array object of PackedEmployee type is around 224 megabytes.

With the help of Memory Analyzer, we were able to find out the top consumer object for the modified Employee Payroll Management program, and it is the java.lang.Thread 0x7fd859c098 main Thread. To find out the objects present in this thread, we can look at the dominator tree for this thread in the Memory Analyzer and the corresponding results are displayed in Table 5.4.

So, from Table 5.4 we can conclude that the top consumer object in this thread is
Table 5.4: Dominator Tree for java.lang.Thread @ 0x7fd859cdf098 main Thread
the array object of PackedEmployee type, and it occupies around 224 megabytes on
the heap.

So, from the above results we can conclude that, when we switched to packed objects
from standard Java objects to program the Employee Payroll Management program,
the size of the array object used to hold the records of 1,000,000 employees decreased
to 224 megabytes from 668 megabytes, a reduction of 66%.

Table 5.5 shows the size of the array object of Employee type and the array ob-
ject of the PackedEmployee type for different array lengths.

<table>
<thead>
<tr>
<th>Array Length</th>
<th>Size of Employee [ ] (in bytes)</th>
<th>Size of PackedEmployee[ ] (in bytes)</th>
<th>Memory Reduction (in percentage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>668,000,008</td>
<td>224,000,032</td>
<td>66.47</td>
</tr>
<tr>
<td>500,000</td>
<td>334,000,008</td>
<td>112,000,032</td>
<td>66.47</td>
</tr>
<tr>
<td>100,000</td>
<td>66,800,008</td>
<td>22,400,032</td>
<td>66.47</td>
</tr>
<tr>
<td>10,000</td>
<td>6,680,008</td>
<td>2,240,032</td>
<td>66.47</td>
</tr>
</tbody>
</table>

Table 5.5: Comparison of the Sizes of the Employee[ ] Object and
PackedEmployee[ ] Object

So, from Table 5.5, we can note that the size of the array object used to store records
of the employees decreased by 66 % when we switched to packed objects regardless
of the length of the array.

Now, let us look at Table 5.6 that lists the size of the array objects of the Employee type and the PackedEmployee type, estimated by the profiling tool for different array lengths. From Table 5.6, we can note that the values calculated by the profiling tool are lower compared to the actual values. This is expected, as the profiling tool tries to calculate the best possible estimate of the size of the class. The reason for this can be the way in which the objects are laid in the memory by the VM. The amount of padding required between the successive fields of the object or between the successive elements of the array object can have a significant impact on the overall size of that object. While calculating the size of the class, we are considering the impact of padding, but our rule set is limited at the moment. But, we should keep in mind that the possible performance gains in terms of memory usage estimated by the profiling tool is the lower bound, and in reality the gains that can be achieved by switching to packed objects can be much higher.

<table>
<thead>
<tr>
<th>Array Length</th>
<th>Size of Employee [ ] (in bytes)</th>
<th>Size of PackedEmployee[ ] (in bytes)</th>
<th>Memory Reduction (in percentage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>368,000,016</td>
<td>204,000,032</td>
<td>44.57</td>
</tr>
<tr>
<td>500,000</td>
<td>184,000,016</td>
<td>107,000,032</td>
<td>41.85</td>
</tr>
<tr>
<td>100,000</td>
<td>36,800,016</td>
<td>20,400,032</td>
<td>44.57</td>
</tr>
<tr>
<td>10,000</td>
<td>3,680,016</td>
<td>2,040,032</td>
<td>44.57</td>
</tr>
</tbody>
</table>

Table 5.6: Size of the Employee [ ] Object and PackedEmployee[ ] Object Returned by the Profiling Tool

5.4.2 Comparison of the Size of the Classes

Now, let us look at the comparison of the size of the individual classes of the Employee Payroll Management program in the standard Java model and in the packed object
data model. Table 5.7 compares the size of the individual classes obtained from the Memory Analyzer (shallow heap size) and the ones calculated by the profiling tool in the standard Java model. The reason for using the shallow heap size instead of the retained heap size is that the approach used to calculate the values by the profiling tool is similar in how the shallow heap size is calculated. This is particularly true for this Java program, as most of the classes contains fields of string type, and while calculating the size of these classes, the profiling tool will only account for the pointer that points to the string object and not the actual size of that string object.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Memory Analyzer (in bytes)</th>
<th>Profiling Tool (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>72</td>
<td>88</td>
</tr>
<tr>
<td>PersonalInformation</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>EmergencyContactInformation</td>
<td>24</td>
<td>40</td>
</tr>
<tr>
<td>BankingInformation</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td>LeaveInformation</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>MedicalHistoryInformation</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>DeductionInformation</td>
<td>32</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 5.7: Comparison of the Sizes of the Individual Classes in the Standard Java Model Obtained from the Memory Analyzer and the Sizes Estimated by the Profiling Tool

Table 5.8 compares the size of the individual classes obtained from the Memory Analyzer (shallow heap size) and the ones calculated by the profiling tool in the packed object data model. One interesting thing to note is the size of the PackedMedicalHistoryInformation class and the size of the equivalent MedicalHistoryInformation class. This class contains 10 fields of boolean type that are used to store employee’s medical history, and as explained in Section 2.1.2.1 the fields of boolean
type occupies only a single byte in memory in the packed object data model. So, we should expect to see a lower value for the size of this class in the packed object data model, and indeed this is the case, as the size of this class, obtained from the Memory Analyzer is 40 bytes in the packed object data model and 48 bytes in the standard Java model.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Memory Analyzer (in bytes)</th>
<th>Profiling Tool (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PackedEmployee</td>
<td>248</td>
<td>240</td>
</tr>
<tr>
<td>PackedPersonalInformation</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>PackedEmergencyContactInformation</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>PackedBankingInformation</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>PackedLeaveInformation</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>PackedMedicalHistoryInformation</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>PackedDeductionInformation</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 5.8: Comparison of the Sizes of the Individual Classes in the Packed Object Data Model Obtained from the Memory Analyzer and the Sizes Estimated by the Profiling Tool

5.4.3 Comparison of the Execution Time

In this section we compare the two versions of the Employee Payroll Management program with respect to their execution time. The comparison is based on two different timing parameters.

- **Total run time**: Total time required to execute the program.

- **Query time**: Total time required to run a set of queries.
Table 5.9 lists the values of these timing parameters for both versions of the Employee Payroll Management program.

<table>
<thead>
<tr>
<th>Length of the array</th>
<th>Standard Java Model</th>
<th>Packed Object Data Model</th>
<th>Change in Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total run time (in ms)</td>
<td>Query time (in ms)</td>
<td>Total run time (in ms)</td>
</tr>
<tr>
<td>1,000,000</td>
<td>193397</td>
<td>114032</td>
<td>171179</td>
</tr>
<tr>
<td>500,000</td>
<td>87635</td>
<td>49980</td>
<td>87403</td>
</tr>
<tr>
<td>100,000</td>
<td>13087</td>
<td>6603</td>
<td>14875</td>
</tr>
<tr>
<td>10,000</td>
<td>1251</td>
<td>326</td>
<td>1552</td>
</tr>
</tbody>
</table>

Table 5.9: Comparison of the Execution Time in the Standard Java Model and the Packed Object Data Model

From Table 5.9, we can observe that the values of total run time and query time are lower for the program using packed objects for longer arrays, but as the length of the array decreases, the values of total run time and query time are lower for the program using standard Java objects instead. The reason for query time being lower for the program using packed objects for larger arrays can be attributed to the elimination of the cost of pointer chasing. Since, in the packed object data model, the elements of the packed arrays are embedded into the array and they reside next to each other in the memory. So, when the query methods are searching a particular field of the packed array object, it does not result in following an outgoing reference to another object, which is the case in the standard Java model.

Table 5.10 lists the values of timing parameters for both the versions of the Employee Payroll Management program with the JIT compiler turned off.

From Table 5.10, we can note that the values for total run time and query time for the program using packed objects is about four times higher than the corresponding values for the program using standard Java objects. This increase in the values of the
<table>
<thead>
<tr>
<th>Length of the array</th>
<th>Standard Java Model</th>
<th>Packed Object Data Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total run time (in ms)</td>
<td>Query time (in ms)</td>
</tr>
<tr>
<td>1,000,000</td>
<td>157001</td>
<td>764081</td>
</tr>
<tr>
<td>500,000</td>
<td>786921</td>
<td>383061</td>
</tr>
<tr>
<td>100,000</td>
<td>143745</td>
<td>68216</td>
</tr>
<tr>
<td>10,000</td>
<td>13225</td>
<td>6104</td>
</tr>
</tbody>
</table>

Table 5.10: Comparison of the Execution Time in the Standard Java model and the Packed Object Data Model with the JIT Turned Off

Timing parameters can be attributed to the way the bytecode interpreter interprets the program using the packed objects when the JIT is turned off. The following code snippets includes the general outline of the query method used in the programs, and how the bytecode interpreter interprets it with the JIT turned off.

General outline of the search queries used:

```java
for (int i = 0; i < employees.length; i++) {
    Employee employee = employees[i];
    SomeData data = employee.someData;
    if (data.someField > threshold) {
        doSomething();
    }
}
```

But when the search query is written using packed objects, the bytecode interpreter interprets the code in the following way:

```java
for (int i = 0; i < employees.length; i++) {
    new PackedEmployee() target = employees, offset =
    employees.offset + (i * sizeof(PackedEmployee))
    doSomething();
}
```
new SomeData() target = employee.target, offset =
  employee.offset + offsetof(someData);
  // employee object is now garbage

if (data.someField > threshold) {
  // load from data + offsetof(someField)
  // data object is now garbage
  doSomething();
}

So, due to the time required to calculate the offset addresses and the time required to perform additional garbage collections to collect the two allocation that are discarded immediately, the values for the query time and total run time are quite higher for the program that uses packed objects compared to the program that uses standard Java objects, as the bytecode interpreter does not have to calculate the offset addresses for it.

5.4.4 Comparison of Performance of the Standard Java Object with the Off-Heap Packed Object

In order to find the exact performance gain that can be achieved in terms of reduced memory consumption, when we switch to off-heap packed objects instead of standard Java objects in the program that uses a native library to compress/decompress a string, we used IBM’s Health Center tool [22] to monitor the native memory usage of the programs. We monitored the use of Process Physical Memory by the programs, which is the the amount of physical memory (RAM) used by the monitored process. Table 5.11 shows a comparison of the values of process physical memory used by the program using standard Java object and the program using packed objects for strings of different lengths.
<table>
<thead>
<tr>
<th>Length of the string</th>
<th>Process Physical Memory (in MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Standard Java Model</td>
</tr>
<tr>
<td>512,000</td>
<td>89.35</td>
</tr>
<tr>
<td>400,000</td>
<td>83.9</td>
</tr>
<tr>
<td>256,000</td>
<td>75.2</td>
</tr>
<tr>
<td>128,000</td>
<td>68.2</td>
</tr>
<tr>
<td>56,000</td>
<td>62.95</td>
</tr>
</tbody>
</table>

Table 5.11: Comparison of the Process Physical Memory in the Standard Java model and the Packed Object Data Model

From Table 5.11, we can observe that the values of the process physical memory used by the program using packed objects is lower than the program that uses standard Java objects. This can be attributed to the fact that the use of off-heap packed objects causes the data to lie in the native memory, which eliminates the redundant data copying required while using the standard Java objects to send data back and forth between the Java heap and the native memory.

<table>
<thead>
<tr>
<th>Length of the string</th>
<th>Standard Java Model</th>
<th>Packed Object Data Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compression time (in ms)</td>
<td>Decompression time (in ms)</td>
</tr>
<tr>
<td>512,000</td>
<td>110</td>
<td>16</td>
</tr>
<tr>
<td>400,000</td>
<td>86</td>
<td>13</td>
</tr>
<tr>
<td>256,000</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>128,000</td>
<td>28</td>
<td>3</td>
</tr>
<tr>
<td>56,000</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.12: Comparison of the Execution Time in the Standard Java Model and the Packed Object Data Model

Table 5.12 compares the time required to compress and decompress strings of different lengths for both versions of the program. From Table 5.12, we can note that the time required to compress and decompress strings are almost identical in both
versions of the program.
Chapter 6

Conclusion and Future Work

The goal of this thesis was to provide users with a profiling tool that can assist them to determine whether switching to packed objects to program their Java application can result in any performance improvement, as use of packed objects in certain situations can result in reduced memory consumption. The profiling tool collects the data required for its analysis from the program, while it is being executed by the VM. Once the program execution completes, the profiling tool presents a detailed analysis about the user's Java application. The results of the profiling tool are segregated in four cases, and they are:

- Case 1: List of all classes with counts of the number of fields of type byte, boolean, char and short declared in them.

- Case 2: List of all the array objects of non-primitive data type allocated by the program, along with their estimated size in the standard Java model and the corresponding size in the packed object data model.

- Case 3: List of all the objects of non-primitive data type allocated by the program, along with their estimated size in the standard Java model and the corresponding size in the packed object data model.
• Case 4: List of all the JNI calls used by the program to access/modify the elements of the arrays of primitive data types along with an estimation of number of bytes transferred between the Java heap and the native memory for a particular JNI call.

After carefully analyzing the results generated by the profiling tool, the user can determine the performance gain in terms of reduced memory consumption that can be achieved if the classes identified by the profiling tool are rewritten using packed objects instead of standard Java objects.

In order to verify the correctness of the results produced by the profiling tool, its implementation was evaluated for all the four cases using several test cases as explained in Chapter 5. Once the functionality of the profiling tool was verified, in order to validate the usefulness of the results generated by it, Java benchmarks were used. The custom benchmarks described in Chapter 5 exhibit certain characteristics that can be identified by the tool. These characteristics are allocating large arrays or large number of smaller arrays of non-primitive data type, extensive use of the JNI API to access/modify elements of the arrays of primitive data types from the native code, classes with many fields of type byte, boolean, char and short. The benchmarking methodology used by us included the following steps:

• Run the Java benchmark programmed using standard Java objects with the profiling tool and record the results produced by it.

• Evaluate the performance of the benchmark based on the parameters memory usage and execution time.

• Rewrite the classes identified by the profiling tool using packed objects.

• Evaluate the performance of the modified benchmark based on the parameters memory usage and execution time, and compare these results with the ones
obtained earlier to check if any performance improvements are found.

For the Employee Payroll Management program, which is used as a benchmark for validating the results of the profiling tool for case 1, 2 and 3, when the classes identified by the profiling tool were rewritten using packed objects, the size of the array object which is used to hold the records of all the employees of a company decreased by 66%. For the program that is used for compression/decompression of strings, when the char array used to hold the strings are declared as an off-heap packed objects instead of standard Java objects, the native memory usage of the program decreased between 7% to 3.5% without any significant increase in the overall execution time.

The benchmarking results explained in Chapter 5 validate the correctness and usefulness of the results produced by the profiling tool. The profiling tool correctly identifies the classes and the array objects, where switching to packed objects results in reduced memory consumption by the program, as shown in the Chapter 5. But the size of these array objects and classes estimated by the profiling tool is lower than the actual values obtained from the Memory Analyzer, but the profiling tool’s results gives the user an approximate idea about the kind of benefits they can achieve. With help of the results generated by the profiling tool, the user can make an educated decision about whether switching to packed objects in their Java application can lead to any performance improvements.

Further development in this research can be directed towards improving the accuracy of the profiling tool’s estimation of the Java class sizes, which is very critical in estimating the size of the objects as well as array objects of non-primitive data type. This can be achieved by incorporating the following changes:

- Currently, if a class contains an array of either primitive or non-primitive data
type, then we are not adding the size of this array object to the overall size of the class. This is due to the limitation of the library that is used to perform BCI. The current library allows us to obtain the class of the array object, but we cannot find the length of the array. The length of the array is very crucial in determining the size of the array object. But, if a more advanced library is used to perform BCI, then it may be possible to find out the length of the array, which would help to improve the accuracy of the estimation of the class size made by the profiling tool.

- Currently, we are tracking the JNI function calls used to access/modify elements of the arrays of primitive data types. But, this can be extended to track the JNI function calls to the arrays of non-primitive data type as well.

- More complex Java applications can be run with the profiling tool, and after following the benchmarking methodology explained in Chapter 5, if the results of the profiling tool can be validated as correct and useful, then it will improve the credibility of this tool.
Bibliography


[20] "SMAZ Library,"[Online].


A.1 Agent Start-up

This section describes the JVMTI functions required to initialize the JVMTI agent during the VM's start-up phase. It also describes the callback functions used for different events such as VMStart, VMinit, VMDeath and ClassLoad.

Agent Start-Up: The VM starts a JVMTI agent by invoking a start-up function. The JVMTI agent can be started in either the OnLoad phase or in the live phase. Our agent is started in the OnLoad phase.

When an agent is started during the OnLoad phase then the agent library has to export a start-up function with the following prototype:

```
JNIEXPORT jint JNICALL JNICALL
Agent_OnLoad(JavaVM *vm, char *options, void *reserved)
```

The VM will call this function to start the agent early in the VM initialization phase so that:
• System properties can be set before they have been used in the start-up of the VM

• Full set of VM capabilities are still available

• No bytecodes have been executed yet

• No classes have been loaded in the VM yet

• No objects have been created yet

We have to enable several VM capabilities for the JVMTI functions and events that we have used in our code. These capabilities must be added inside the Agent_OnLoad function. The capabilities required for each JVMTI function and event is listed in the JVMTI reference page [13]. For example, to use Class File Load Hook event, the can_generate_all_class_hook_events capability must be true. We set all the capabilities required by our Profiling Tool to true, and then add them to the JVMTI environment using the AddCapabilities JVMTI function. This is described in following code snippet:

```c
jint JNICALL Agent_OnLoad(JavaVM *vm, char *options, void *reserved)
{
    jvmtiCapabilities capabilities;

    /* Get/Add JVMTI capabilities */
    (void)memset(&capabilities, 0, sizeof(capabilities));
    capabilities.can_generate_all_class_hook_events = 1;
    capabilities.can_tag_objects = 1;
    capabilities.can_generate_garbage_collection_events = 1;
    capabilities.can_generate_vm_object_alloc_events = 1;
    err = (*jvmti)->AddCapabilities(jvmti, &capabilities);
}
```
The `Agent_OnLoad` function is also used to enable the events that are going to be used in our code. The events are enabled using JVMTI's `SetEventNotificationMode` function. This is explained in the following code snippet:

```c
err = (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
                                           JVMTI_EVENT_VM_START, NULL);
err = (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
                                           JVMTI_EVENT_VM_INIT, NULL);
err = (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
                                           JVMTI_EVENT_VM_DEATH, NULL);
err = (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
                                           JVMTI_EVENT_CLASS_FILE_LOAD_HOOK, NULL);
err = (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
                                           JVMTI_EVENT_VM_OBJECT_ALLOC, NULL);
```

Note that in above functions, NULL is passed as the third parameter, which enables the event notification globally.

For all the registered events we must have a designated callback function. A callback for the event is set using the `SetEventCallbacks` JVMTI function. For example, if a JVMTI event of type Class File Load Hook occurs, our agent sends it to the callback function `cbClassFileLoadHook()`. This is explained in the following code snippet:

```c
jvmtiEventCallbacks callbacks;

/* Set callbacks and enable event notifications */
memset(&callbacks, 0, sizeof(callbacks));
callbacks.VMStart = &cbVMStart;
```
callbacks.VMInit = &vmInit;
callbacks.VMDeath = &vmDeath;
callbacks.DataDumpRequest = &dataDumpRequest;
callbacks.ClassFileLoadHook = &cbClassFileLoadHook;
callbacks.VMObjectAlloc = &cbVMObjectAlloc;

We need a data structure that will act as a global agent data area and it will be used throughout our code. The data structure used is the one shown below:

```c
/* Global static data */
typedef struct {
    jboolean vmDeathCalled;
    jboolean vmStarted;
    jboolean dumpInProgress;
    jboolean vmInitialized;
    jvmtiEnv *jvmti;
    jrawMonitorID lock;
} GlobalAgentData;
static GlobalAgentData *gdata;
```

In the Agent_OnLoad function we have to setup the initial global agent data area, so that it can be used in the code. This is done in the following way:

```c
static GlobalAgentData data;

(void)memset((void *)&data, 0, sizeof(data));
gdata = &data;
```
The next step is to get the JVMTI environment and assign it to the global agent data structure's jvmti element. This is done in the following way:

```c
jint rc;

/* Get JVMTI environment */
jvmti = NULL;
rc = (**vm)->GetEnv(vm, (void **)&jvmti, JVMTI_VERSION);
if (rc != JNI_OK) {
    fatal_error("ERROR: Can't create jvmtiEnv, error=%d\n",rc);
    return -1;
}
if ( jvmti == NULL ) {
    fatal_error("ERROR: No jvmtiEnv* returned from GetEnv\n");
}
gdata->jvmti jvmti;
```

We need to create a raw monitor using JVMTI's `CreateRawMonitor` function. Once we have created a raw monitor, we can use it along with `RawMonitorEnter` JVMTI function to enter a critical section of the code. While exiting a critical section, we can use the monitor along with `RawMonitorExit` JVMTI function to release the exclusive ownership of the monitor. This is explained in the following code snippet:

```c
/* Create the raw monitor */
err = (*jvmti)->CreateRawMonitor(jvmti, "agent lock",
    &(gdata->lock));

/* Enter agent monitor protected section */
```
static void enterAgentMonitor(jvmtiEnv *jvmti)
{
    jvmtiError err;

    err = (*jvmti)->RawMonitorEnter(jvmti, gdata->lock);
    check_jvmti_error(jvmti, err, "raw monitor enter");
}

/* Exit agent monitor protected section */
static void exitAgentMonitor(jvmtiEnv *jvmti)
{
    jvmtiError err;

    err = (*jvmti)->RawMonitorExit(jvmti, gdata->lock);
    check_jvmti_error(jvmti, err, "raw monitor exit");
}

We have used JVMTI’s AddToBootstrap() function to cause the instrumentation class ProfilingTool, which is inside the profilingTool.jar file to be defined by the bootstrap class loader [13].

jvmtiError
AddToBootstrapClassLoaderSearch(jvmtiEnv* env,
    const char* segment)

After the bootstrap class loader unsuccessfully searches for a class, the specified platform-dependent search path segment will be searched as well.

Callback Function for VM.START Event
The VM Start event signals the start of the VM. Once this event is generated, the
agent can start using any of the JNI functions. In the callback function for the VM Start event, we have to set up the Tracker class (ProfilingTool) which is used for performing bytecode instrumentation. First we get the jclass handle for the Tracker class using JNI's \texttt{FindClass()} function. Then we register the native methods used in the Tracker class using JNI's \texttt{RegisterNatives()} function. Next we set the static int field \textit{engaged} of the Tracker class to 1 using JNI's \texttt{SetStaticIntField()} function. The \textit{engaged} field is set to 0 by default. We can't call JNI methods until VM has started, so once the VMStart event is generated, we now have the ability to call the JNI functions. Now, we set \textit{engaged} field to 1, so the tracker methods inside the Tracker class will be able to call the native methods we have registered earlier.

Next we use JVMTI's \textbf{JNI Function Interception functions} to intercept and resend JNI function calls to our instrumenting functions by manipulating the JNI function table. This enables us to collect the data we require for our analysis of Case 4, which is explained in detail in Section 4.2.4. In the callback function for the VM start event, we modify the entries in the JNI function table, such that they point to our instrumenting function rather than the original JNI function by using JVMTI's \texttt{GetJNIFunctionTable()} and \texttt{SetJNIFunctionTable()} functions. Finally we indicate that the VM Start process has begun by setting the \texttt{vmStarted} element of the global data structure. This is explained in the following code snippet:

```c
/* Java Native Methods for class */
static JNINativeMethod registry[2] = {
{STRING(PROFILING_TOOL_native_newobj),"(Ljava/lang/Object;
Ljava/lang/Object;)V",(void*)&PROFILING_TOOL_native_newobj},
{STRING(PROFILING_TOOL_native_newarr), "(Ljava/lang/Object;
Ljava/lang/Object;Ljava/lang/String;Ljava/lang/Class;)V",
(void*)&PROFILING_TOOL_native_newarr}};
```

105
/* Register Natives for class whose methods we use */
klass = (*env)->FindClass(env, STRING(PROFILING_TOOL_class));
if ( klass == NULL ) {
    fatal_error("ERROR: JNI: Cannot find %s with FindClass\n",
        STRING(PROFILING_TOOL_class));
}
state = (*env)->RegisterNatives(env, klass, registry, 2);
if ( state != 0 ) {
    fatal_error("ERROR: JNI: Cannot register natives for class
        %s\n", STRING(PROFILING_TOOL_class));
}

/* Engage calls. */
field = (*env)->GetStaticFieldID(env, klass,
    STRING(PROFILING_TOOL_engaged), "I");
if ( field == NULL ) {
    fatal_error("ERROR: JNI: Cannot get field from %s\n",
        STRING(PROFILING_TOOL_class));
}
(*env)->SetStaticIntField(env, klass, field, 1);

/* Redirect JNI functions to instrumenting functions */
err = (*jvmti)->GetJNIFunctionTable(jvmti,
    &original_jni_Functions);
err = (*jvmti)->GetJNIFunctionTable(jvmti,
    &redirected_jni_Functions);
redirected_jni_Functions->GetArrayLength = MyGetArrayLength;

err = (*jvmti)->SetJNIFunctionTable(jvmti,
Callback Function for VM_INIT Event

The VM initialization event signals the completion of VM initialization process to
the agent. Once this event is generated, the agent is free to call any JNI or JVMTI
functions. In the callback function for this event, we indicate that VM initialization
process is completed by setting vmInitialized element of the global data structure.

Callback Function for VM.DEATH Event

The VMDeath event notifies the agent that the VM has been terminated. No other
events can occur after the VMDeath event. In the callback function of this event,
we first use JVMTI's `ForceGarbageCollection()` function to force a garbage col­
clection. This ensures that everything has been garbage collected. Next we turn the
Tracker class (ProfilingTool) off by setting the static field `engaged` to 0. We also
have to disable the JVMTI callbacks that were enabled in the Agent_OnLoad func­
tion. The next step is to call dataDumpRequest() function. Since for Case 1 we
are using static analysis, the data collection process takes place inside the dataD­
umpRequest() function. So when we call this function before VM death process is
called, it will send the data for Case 1 to the terminal or the output file. Finally we
have to indicate that the VM death process has begun by setting the vmDeathCalled
element of the global data structure. This process is explained in the following code
snippet:
Disengage calls in HEAP_TRACKER_class.

```c
klass = (*env)->FindClass(env, STRING(PROFILING_TOOL_class));
if (klass == NULL) {
    fatal_error("ERROR: JNI: Cannot find %s with FindClass\n",
                STRING(PROFILING_TOOL_class));
}
field = (*env)->GetStaticFieldID(env, klass,
    STRING(PROFILING_TOOL_engaged), "I");
if (field == NULL) {
    fatal_error("ERROR: JNI: Cannot get field from %s\n",
                STRING(PROFILING_TOOL_class));
}
(*env)->SetStaticIntField(env, klass, field, 0);
dataDumpRequest(jvmti);
gdata->vmDeathCalled = JNI_TRUE;
```

Agent_OnUnload

This function will be called by the VM when it is about to unload the JVMTI agent. Inside this function we have to free all the resources allocated in the Agent_OnLoad function.

```c
/* Agent_OnUnload() is called last */
JNIEXPORT void JNICALL
Agent_OnUnload(JavaVM *vm)
{
    /*all malloc/calloc/strdup space needs to be freed*/
}
```
A.2 java_crw_demo library

The basic BCI that this library does includes:

1. On entry to the java.lang.Object init method (signature "()V"), an invokestatic call to tclass.obj_init_method(object); is inserted.
2. On any newarray type opcode, immediately following it, the array object is duplicated on the stack and an invokestatic call to tclass.newarray_method(object); is inserted.
3. On entry to all methods, a invokestatic call to tclass.call_method(cnum,mnum); is inserted.
4. On return from any method (any return opcode), a invokestatic call to tclass.return_method(cnum,mnum); is inserted.

Basically java_crw_demo library contains two functions:

1. java_crw_demo.classname: It is used to extract out the classname from a class file.

```c
char * java_crw_demo_classname(
    const unsigned char * file_image,
    long file_len,
    FatalErrorHandler fatal_error_handler);
```

Where,

file_image: Pointer to the class file data buffer.

file_len: Length of class file data buffer.

2. java_crw_demo: This function can be called with a class file image and it returns an instrumented class file image. It's arguments are described below:

```c
void java_crw_demo(
```
Where,

**class_number**: A unique identifying number for the class which can be defined by the user in their JVMTI agent.

**name**: The fully qualified class name in the form of "java/lang/Object".

**file_image**: The class file image.

**file_len**: The number of bytes in the class file image.

**system_class**: In order to avoid modifying system classes, for classes loaded earlier in the VM startup this value is set to non-zero.

**tclass_name**: The name of the tracker class that contains static methods which are called as part of the instrumentation code.
tclass_sig : The fully qualified name of the Tracker class.

call_name : The name of the static method in the Tracker class that will be used for method entries or indications of method calls.

call_sig : The method signature for the call_name method.

return_name : The name of the static method in the Tracker class that will be used for method exits or indications of method returns.

return_sig : The method signature for the return_name method.

obj_init_name : The name of the static method in the Tracker class that will be used for object allocations.

obj_init_sig : The method signature for the obj_init_name method.

newarray_name : The name of the static method in the Tracker class that will be used for array allocations.

newarray_sig : The method signature for the newarray_name method.

pnew_file_image : If instrumentation is successful, then this will be a pointer to the instrumented class file image.

pnew_file_len : The length of the new class file image returned in *pnew_file_image.

fatal_error_handler : If non NULL, provides a function to call when fatal errors are encountered while parsing or creating the new class file image.

mnum_callback : If non NULL, provides a callback function to get access to the method names and signatures in the class.

In order to successfully perform bytecode instrumentation using java_crw_demo library’s functions, it is required that the JVMTI agent should request a class load event. We have used JVMTI’s Class File Load Hook event for this purpose.

Class File Load Hook : This event is sent when the VM obtains the class file data, but before it constructs the in-memory representation for that class. The
user’s JVMTI agent can instrument the existing class file data sent by the VM to include profiling/debugging hooks by using this event. The agent must allocate the space for the modified class file data buffer if instrumentation is successful.

```c
void JNICALL ClassFileLoadHook(jvmtiEnv *jvmti_env,
                               (JNIEnv* jni_env,
                                jclass class_being_redefined,
                                jobject loader,
                                const char* name,
                                jobject protection_domain,
                                jint class_data_len,
                                const unsigned char* class_data,
                                jint* new_class_data_len,
                                unsigned char** new_class_data)
```

If the user’s JVMTI agent wishes to modify the class file, it must set `new_class_data` to point to the newly instrumented class file data buffer and set `new_class_data_len` to the length of that buffer before returning from this call [13].

In our Profiling Tool agent as class files are loaded in the VM, a class load event is generated. This event is handled in the callback function `cbClassFileLoadHook`. Inside this function we call `java_crw_demo`’s `java_crw_demo_classname` function to extract the class name of the current class file. Then the `java_crw_demo` function is called. To this function we supply the classname of the current class file, the pointer to the current class file data buffer, the length of the current class file data buffer, the signature of the Tracker class and static methods declared within the Tracker class, the pointer to the instrumented class file data buffer and the pointer to the length of the new class file data buffer. If the instrumentation is successful then the length of the new class file data buffer will be greater than zero. In this case we
need to allocate space for the new class file using JVMTI's allocate function. This process is explained in following code snippet:

```c
static void JNICALL cbClassFileLoadHook(jvmtiEnv *jvmti, JNIEnv* env,
  jclass class_being_redefined, jobject loader,
  const char* name, jobject protection_domain,
  jint class_data_len, const unsigned char* class_data,
  jint* new_class_data_len, unsigned char** new_class_data)
{
  java_crw_demo(cnum,
    classname,
    class_data,
    class_data_len,
    systemClass,
    STRING(PROFILING_TOOL_class),
    "L" STRING(PROFILING_TOOL_class) ";",
    NULL, NULL,
    NULL, NULL,
    STRING(PROFILING_TOOL_newobj),
    "(Ljava/lang/Object;)V",
    STRING(PROFILING_TOOL_newarr),
    "(Ljava/lang/Object;)V",
    &newImage,
    &newLength,
    NULL,
    NULL);

  /* If we got back a new class image, return it back as "the"
      new class image */
  if ( newLength > 0 ) {
    unsigned char *jvmti_space;
```
A.3 JNI Function Interception functions

The following example shows how the JVMTI’s JNI function interception functions intercept the call to the `NewWeakGlobalRef` JNI function and resends it to `MyWeakGlobalRef` function in order to count reference creation.

```c
JNIEnv original_jni_funcTable;
JNIEnv redirected_jni_funcTable;
int my_weak_global_ref_count = 0;

jobject MyNewWeakGlobalRef(JNIEnv *jni_env, jobject lobj) {
  ++my_weak_global_ref_count;
  return original_jni_funcTable->NewWeakGlobalRef(env, lobj);
}

void myInit() {
  jvmtiError err;

  err = (*jvmti_env)->GetJNIFunctionTable(jvmti_env, &original_jni_funcTable);
```
if (err != JVMTI_ERROR_NONE) {
    die();
}
err = (*jvmti_env)->GetJNIFunctionTable(jvmti_env,
    &redirected_jni_funcTable);
if (err != JVMTI_ERROR_NONE) {
    die();
}
redirected_jni_funcTable->NewWeakGlobalRef =
    MyNewWeakGlobalRef;
err = (*jvmti_env)->SetJNIFunctionTable(jvmti_env,
    redirected_jni_funcTable);
if (err != JVMTI_ERROR_NONE) {
    die();
}

After myInit is called the user's JNI code will be executed which makes the call
to create a new weak global reference. Instead the call going to the normal JNI
implementation, now the call goes to MyNewWeakGlobalRef. Once the required
data is collected, the call is directed back to the NewWeakGlobalRef.

static void JNICALL
cbVMStart(jvmtiEnv *jvmti, JNIEnv *env)
{
    ..
    redirected_jni_Functions->GetIntArrayElements =
        MyGetIntArrayElements;
    redirected_jni_Functions->GetDoubleArrayElements =
        MyGetDoubleArrayElements;
    redirected_jni_Functions->GetByteArrayElements =
        MyGetByteArrayElements;
redirected_jni_Functions->GetBooleanArrayElements = MyGetBooleanArrayElements;
redirected_jni_Functions->GetCharArrayElements = MyGetCharArrayElements;
redirected_jni_Functions->GetShortArrayElements = MyShortArrayElements;
redirected_jni_Functions->GetLongArrayElements = MyGetLongArrayElements;
redirected_jni_Functions->GetFloatArrayElements = MyGetFloatArrayElements;
...
}

A.4 Obtaining Stack Trace Information

jvmtiError
GetStackTrace(jvmtiEnv* env,
    jthread thread,
    jint start_depth,
    jint max_frame_count,
    jvmtiFrameInfo* frame_buffer,
    jint* count_ptr)

This function gives the information about the stack of a specified thread (If thread is NULL, the current thread is used) [13]. The following example retrieves top five frames of the current thread and prints the name of the currently executing method and the name of the class that declares this method.

jvmtiFrameInfo frames[5];
jint count;
jvmtiError err;
err = (*jvmti)->GetStackTrace(jvmti, aThread, 0, 5,
                        &frames, &count);
if (err == JVMTI_ERROR_NONE && count >= 1) {
    char *methodName;
    char *declaringClassName;
    jclass declaring_class;

    err = (*jvmti)->GetMethodName(jvmti, frames[0].method,
                        &methodName, NULL);
    err = (*jvmti)->GetMethodDeclaringClass(jvmti,
                        frames[0].method, &declaring_class);
    err = (*jvmti)->GetClassSignature(jvmti, declaring_class,
                        &declaringClassName, NULL);
    if (err == JVMTI_ERROR_NONE) {
        printf("Executing method: %s", methodName);
        printf("Executing class: %s", declaringClassName);
    }
}

We have used JVMTI’s **GetMethodName()** function to obtain method name and method signature.

```
jvmtiError
GetMethodName(jvmtiEnv* env,  
jmethodID method,  
char** name_ptr,  
char** signature_ptr,  
char** generic_ptr)
```

For the method indicated by `method`, GetMethodName() returns the method name via `name_ptr` and method signature via `signature_ptr` [13]. In the above example, we have passed `frames[0].method` as the argument to the GetMethodName(), because
frames[0] would contain the topmost frame i.e. current frame. So GetMethodName() returns the name and signature of the currently executing method.

For obtaining the class in which the currently executing method is declared, we have used JVMTI's GetMethodDeclaringClass() function.

```c
jvmtiError
GetMethodDeclaringClass(jvmtiEnv* env,
                        jmethodID method,
                        jclass* declaring_class_ptr)
```

For the method indicated by method, GetMethodDeclaringClass() returns the class that defined it via declaring_class_ptr [13]. In the above example, we have passed frames[0].method as the argument to the GetMethodDeclaringClass(), because frames[0] would contain the topmost frame i.e. current frame. So GetMethodDeclaringClass() returns the class (jclass type) in which the frames[0].method is declared. After that we have used GetClassSignature() JVMTI function to obtain the signature of the class.

### A.5 Command Line Options

While running the script for the Profiling Tool, the user can specify a command line flag which will determine whether the output file will contain consolidated output or more detailed output. In the consolidated output only the four lists containing output for the four Cases will be printed.

These lists are sorted in following way for each of the four Cases:

- For Case 1, the list contains classes with primitive fields of type byte, boolean, char and short in the descending order i.e. a class with maximum count of
fields of these primitive types will be at the top of the list.

- For Case 2, the list contains arrays of non-primitive data type in the descending order of difference in their size in the generic Java model and the packed object data model.

- For Case 3, the list contains objects of non-primitive data type in the descending order of difference in their size in the generic Java model and the packed object data model.

- For Case 4, the list contains arrays of primitive data types which were accessed using JNI's array access API in the descending order of number of bytes accessed/modified from this array.

A.6 Detailed Calculations for Case - 2 Test Case

This section describes the detailed calculation for the test case used for the evaluation of Case 2 in Section 5.2.2.

By using the formulas described in Section 4.1.2.1, the following values are calculated.

DataSize(Line) = TotalObjectSize(Point) + TotalObjectSize(Point) + Size of two pointers which will point to these two Point objects

So first we need to calculate TotalObjectSize(Point):

DataSize(Point) = size of int field + size of int field
DataSize(Point) = 4 bytes + 4 bytes = 8 bytes

ObjectSize(Point) = DataSize(Point) + ObjectHeaderSize = 8 bytes + 4 bytes = 12 bytes
TotalObjectSize(Point) = ObjectSize(Point) = 12 bytes

As per the padding rules explained in Section 4.2.2, we add 4 bytes of padding, so we have:
TotalObjectSize(Point) = 16 bytes

Now returning to calculations of Line object:

DataSize(Line) = 16 bytes + 16 bytes + 4 bytes + 4 bytes = 40 bytes

ObjectSize(Line) = DataSize(Line) + ObjectHeaderSize = 40 bytes + 4 bytes = 44 bytes

TotalObjectSize(Line) = ObjectSize(Line) = 44 bytes

As per the padding rules explained in Section 4.2.2, we add 4 bytes of padding so that the object size will be aligned to 8, so we have: TotalObjectSize(Line) = 48 bytes

Now let us calculate **TotalObjectSize(Line[10])**:

DataSize(Line[10]) = SizeOfPointer * no. of elements of the array

DataSize(Line[10]) = 4 bytes * 10 = 40 bytes

ObjectSize(Line[10]) = DataSize(Line[10]) + ArrayHeaderSize = 40 bytes + 8 bytes = 48 bytes

TotalObjectSize(Line[10]) = ObjectSize(Line[10]) + (TotalObjectSize(Line) * no. of elements of the array)

TotalObjectSize(Line[10]) = 48 bytes + (48 bytes * 10) = 528 bytes

**Packed Calculations:**

PackedDataSize(Line) = PackedDataSize(Point) + PackedDataSize(Point)
If packed object contains an object of some other class as an instance field, then fields of that object are embedded into the packed object with no object header intervening between them. So, first we have to calculate PackedDataSize(Point):

\[ \text{PackedDataSize(\text{Point})} = \text{size of int field} + \text{size of int field} = 4 \text{ bytes} + 4 \text{ bytes} = 8 \text{ bytes} \]

So, now

\[ \text{PackedDataSize(\text{Line})} = 8 \text{ bytes} + 8 \text{ bytes} = 16 \text{ bytes} \]

\[ \text{PackedObjectSize(\text{Line})} = \text{PackedDataSize(\text{Line})} + \text{PackedObjectHeaderSize} = 16 \text{ bytes} + 12 \text{ bytes} = 28 \text{ bytes} \]

\[ \text{TotalPackedObjectSize(\text{Line})} = \text{PackedObjectSize(\text{Line})} = 28 \text{ bytes} \]

In order to align its size to 8, we have to add 4 bytes of padding. So we will have,

\[ \text{TotalPackedObjectSize(\text{Line})} = 32 \text{ bytes} \]

Now we have to calculate \( \text{TotalPackedObjectSize(\text{Line}[10])} \). The key difference between \( \text{TotalObjectSize(\text{Line}[10])} \) and \( \text{TotalPackedObjectSize(\text{Line}[10])} \) is that in the packed arrays, elements of the arrays are embedded within the array. The fields of the array object elements are placed next to each other with no object header intervening between them.

\[ \text{PackedDataSize(\text{Line}[10])} = \text{PackedDataSize(\text{Line})} \times \text{no. of elements of the array} \]

\[ \text{PackedDataSize(\text{Line}[10])} = 16 \text{ bytes} \times 10 = 160 \text{ bytes} \]

\[ \text{PackedObjectSize(\text{Line}[10])} = \text{PackedDataSize(\text{Line}[10])} + \text{PackedArrayHeaderSize} \]

\[ \text{PackedObjectSize(\text{Line}[10])} = 160 \text{ bytes} + 16 \text{ bytes} = 176 \text{ bytes} \]

\[ \text{TotalPackedObjectSize(\text{Line}[10])} = \text{PackedObjectSize(\text{Line}[10])} = 176 \text{ bytes} \]
A.7 Detailed Calculations for Case - 3 Test Case

Non-packed Calculations:

This section describes the detailed calculation for the test case used for the evaluation of case - 3 in Section 5.2.3.

\[ \text{DataSize(Rectangle)} = 4 \times \text{TotalObjectSize(Point)} + \text{Size of four pointers} \]

By using the value of \( \text{TotalObjectSize(Point)} \) calculated in Section 5.2.2, we get
\[ \text{DataSize(Rectangle)} = 4 \times 16 \text{ bytes} + 4 \times 4 \text{ bytes} = 80 \text{ bytes} \]
\[ \text{ObjectSize(Rectangle)} = \text{DataSize(Rectangle)} + \text{ObjectHeaderSize} = 80 \text{ bytes} + 4 \text{ bytes} = 84 \text{ bytes} \]
\[ \text{TotalObjectSize(Rectangle)} = \text{ObjectSize(Rectangle)} = 84 \text{ bytes} \]

As per the padding rules explained in Section 4.2.2, we add 4 bytes of padding so that the object size will be aligned to 8, so we have:
\[ \text{TotalObjectSize(Rectangle)} = 88 \text{ bytes} \]

Packed Calculations:

\[ \text{PackedDataSize(Rectangle)} = \text{PackedDataSize(Point)} + \text{PackedDataSize(Point)} + \text{PackedDataSize(Point)} + \text{PackedDataSize(Point)} \]

If packed object contains object of some other class as an instance field, then fields of that object are embedded into the packed object with no object header intervening between them. So if we use the value \( \text{PackedDataSize(Point)} \) calculated in Section 5.2.2, we get:
\[ \text{PackedDataSize(Rectangle)} = 8 \text{ bytes} + 8 \text{ bytes} + 8 \text{ bytes} + 8 \text{ bytes} = 32 \text{ bytes} \]
\[ \text{PackedObjectSize(Rectangle)} = \text{PackedDataSize(Rectangle)} + \text{PackedObjectHeaderSize} = 32 \text{ bytes} + 12 \text{ bytes} = 44 \text{ bytes} \]
\[ \text{TotalPackedObjectSize(Rectangle)} = \text{PackedObjectSize(Rectangle)} = 44 \text{ bytes} \]
As per the padding rules described in Section 4.2.2, in order to align the size this object to 8 we have to add 4 bytes of padding.

TotalPackedObjectSize(TableName) = 48 bytes

A.8 Test Case for the Case - 4

This section includes the code for the test case used for evaluating the functionality of the Case 4.

Test case used for the Case 4:

Java Program of the Test Case

```java
public class HelloWorld {

    public native int[] returnArrayMethod (int size);
    public native void intArrayMethod(int[] intArray);
    public native void criticalIntArrayMethod(int[] intArray);
    public native void intArrayRegionMethod (int[] intArray, int start, int len);

    static {
        System.loadLibrary("testlib");
    }

    static public void main(String argv[]) {

        int[] intTest = new int[10];
        intTest = helloWorld.returnArrayMethod(10);
        helloWorld.intArrayMethod(intTest);
        helloWorld.criticalIntArrayMethod(intTest);
        helloWorld.intArrayRegionMethod(intTest, 0, 5);
    }
}
```
C program of the Test Case

/\Inside testlib.c implementation of native methods*/

JNIEXPORT jniArrary JNICALL Java_HelloWorld_returnArrayMethod
  (JNIEnv *env, jobject obj, jni int size)
{

  jniArrary result;
  result = (*env)->NewintArray(env, size);
  if (result == NULL) {
    return NULL; /* out of memory error thrown */
  }

  int i;
  // fill a temp structure to use to populate the java int
  // arra
  jint fill[256];
  for (i = 0; i < size; i++) {
    fill[i] = 0; // put whatever logic you want to populate
    the values here.
  }

  // move from the temp structure to the java structure
  (*env)->SetintArrayRegion(env, result, 0, size, fill);
  return result;
}

JNIEXPORT void JNICALL Java_HelloWorld_intArrayMethod
  (JNIEnv *env, jobject obj, jniArrary array) {
  int i, sum = 0;
  int count = 5;
```c
jint arr1[count];
int sum1 = 0;

jsize len = (*env)->GetArrayLength(env, array);
jint *body = (*env)->GetIntArrayElements(env, array, 0);
for (i=0; i<len; i++){
    sum += body[i];
}
(*env)->ReleaseIntArrayElements(env, array, body, 0);
printf("sum is = %d\n ", sum);
}

JNIEXPORT void JNICALL Java_HelloWorld_criticalIntArrayMethod
    (JNIEnv *env, jobject obj, jintArray array){

    int i, sum = 0;
    int count = 5;
jint arr1[count];
int sum1 = 0;

    jsize len = (*env)->GetArrayLength(env, array);
jint *body = (*env)->GetPrimitiveArrayCritical(env, array, 0);
    for (i=0; i<len; i++){
        sum += body[i];
    }
    (*env)->ReleasePrimitiveArrayCritical(env, array, body, 0);
    printf("sum is = %d\n ", sum);
}

JNIEXPORT void JNICALL Java_HelloWorld_intArrayRegionMethod
    (JNIEnv *env, jobject obj, jintArray array, jint start, jint index){
int len, i = 0;
```
len = index - start;

jint arr[len];
int sum = 0;

(*env)->GetIntArrayRegion(env, array, start, index, arr);
for (i = 0; i<len; i++){
    sum += arr[i];
}

printf("sum is = %d\n ", sum);
}
Vita

Candidate's full name: Umang Umesh Pandya

University attended:
Master of Computer Science, 2014
University of New Brunswick,
Fredericton, Canada.

Bachelor of Engineering, Electronics and Communication, 2007,
K.J.Somaiya College of Engineering,
University of Mumbai,
Mumbai, India.