國 立 交 通 大 學

網路工程研究所

碩 士 論 文

兩 階 段 式 線 上 記 憶 體 洩 漏 偵 測

Two-Phase Online Memory Leak Detection

研 究 生：梁睿珊

指導教授：林一平　教授

　　　　　蔡孟勳　教授

中 華 民 國 一 〇 一 年 六 月

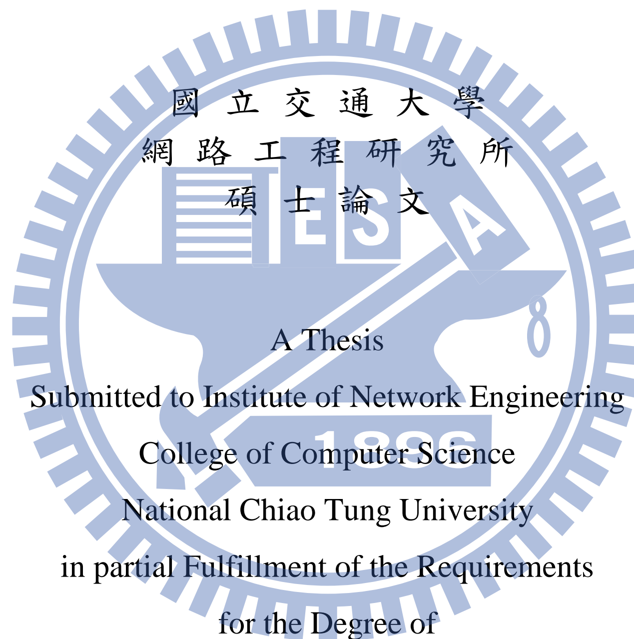
兩階段式線上記憶體洩漏偵測

Two-Phase Online Memory Leak Detection

研 究 生：梁睿珊　　Student：Jui-Shan Liang

指導教授：林一平 博士　　Advisors：Yi-Bing Lin

蔡孟勳 博士　　Meng-Hsun Tsai



國 立 交 通 大 學

網 路 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Network Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2012

Hsinchu, Taiwan, Republic of China

中華民國一〇一年六月

兩階段式線上記憶體洩漏偵測

學生：梁睿珊　　　　　　　　指導教授：林一平教授

蔡孟勳教授

國立交通大學網路工程研究所碩士班

摘　　　要

程式執行過程中若有記憶體洩漏之現象發生，會導致記憶體資源的浪費，降低系統運行的效能，甚至導致整個系統當機。記憶體洩漏的問題往往不易在開發階段立即發現並修復，因此需要線上記憶體洩漏偵測之技術。

本論文提出以兩階段的方式線上偵測 Java 程式之記憶體洩漏。在第一階段（離線分析），使用者執行所要監控之程式一段時間，並使用我們所提出之代理程式分析其記憶體堆疊 (heap)。代理程式接著離線計算記憶體堆疊中各資料結構的總佔用位元數 (total occupied bytes)，以便使用者依據此計算結果選擇可能造成記憶體洩漏之物件類別。在第二階段（線上監看），為了減少程式運行時額外增加的時間與空間，代理程式僅監看使用者所選擇的類別之物件，而非監看記憶體堆疊中的所有物件。本方法會回報被監看物件最後被存取之時間與位置（包含原始碼檔名以及行號），以利使用者參考並修改程式。

Two-Phase Online Memory Leak Detection

Student: Jui-Shan Liang Advisors: Prof. Yi-Bing Lin
Prof. Meng-Hsun Tsai

Institute of Network Engineering
National Chiao Tung University

ABSTRACT

Memory leaks generated by a running program may exhaust memory resources and degrade system performance. In worst case memory leaks eventually crash the whole system. They threaten long-running systems such as customer service systems in telecom operations. It is hard to reproduce these kinds of leaks, let alone to identify and fix them early in the development phase. Thus online memory leak detection is required.

In this paper we propose a two-phase approach to the online memory leak detection problem for Java programs. In phase one (offline analysis), the user executes the investigated program for a while and analyzes its heap with the proposed agent. The agent then summarizes the total occupied bytes of data structures in the heap offline, and the user is then able to select classes whose object instances seem to be potential leaks by examining the total occupied bytes summary. In phase two (online monitoring), to reduce the space and time overhead at runtime, the agent monitors online the objects of selected classes instead of monitoring all objects in the heap. The approach reports the last accessed time and location (including the source file name and the line number) of each leak candidate, with which the user can identify and fix leaks in the program.

Acknowledgements

I would like to express my sincere thanks to my advisors Prof. Yi-Bing Lin and Prof. Meng-Hsun Tsai for their supervision and perspicacious advices. I could not complete this thesis without their guidance. I would also like to thank my committee members, Prof. Ai-Chun Pang and Dr. Ying-Rong Sung, for their comments and encouragements.

Many thanks to my two teammates Hung-Wei Kao and Kuan-Hsien Li in Intelligent Mobile Service Laboratory, National Cheng Kung University. I will always treasure the days that we worked remotely and realized our ideas together.

My colleagues of Laboratory 117 have enriched my graduate student life. Thank them for all the happy memories they have brought me. Thanks also to my love Chih-Ta Lin for his warm support and inspiring discussions.

Last but not least, I would like to express my sincere thanks to my parents and my elder brother. Without their patience and selfless support through all these years, I would not have become who I am today. I feel truly grateful growing up in this family.

Jui-Shan Liang

July 2012

Contents

摘要 I

Abstract II

Acknowledgements III

Contents IV

List of Figures VII

List of Tables VIII

1 Introduction 1

2 Related Work 3

3 Concept of Referencing Among Objects 5

3.1 Total Occupied Bytes 5

3.2 Reference Paths and Reference Rules 8

4 Design and Implementation of Two-Phase Online Memory Leak Detec-

tion **12**

4.1 Overview 12

4.1.1 Structure Diagram 13

4.1.2 Phase One: Offline Analysis 13

4.1.3 Phase Two: Online Monitoring 14

4.2 User Interface Module 15

4.2.1 Phase One Dialog Component 15

4.2.2 Phase Two Dialog Components 18

4.3 Core Module 20

4.3.1 Heap Snapshot Component 20

4.3.2 Dominator Tree Component 21

4.3.3 Object Monitor Component 21

4.3.4 Record Component 22

4.3.5 Reporter Component 23

5 Performance Evaluation **24**

5.1 Locating Memory Leak in Eclipse 3.1.2 24

5.2 Space Overhead Analysis 25

5.2.1 Phase One (Offline Analysis) 25

5.2.2 Phase Two (Online Monitoring) 27

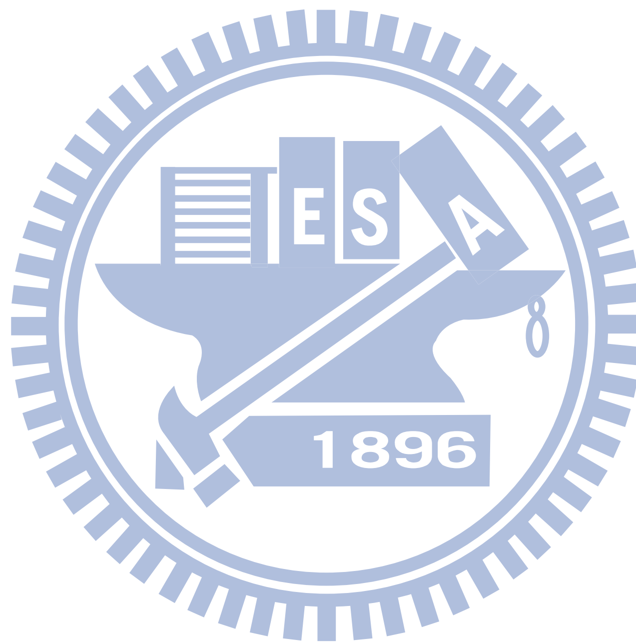
5.3 Time Overhead Analysis 28

6 Conclusions **31**

Bibliography **32**

A The MyProgram Program **34**

B The RuntimeExp Program **44**



List of Figures

3.1 Heap snapshot of our sample program MyProgram 6

3.2 An illustration of total occupied bytes of dominator tree nodes 8

3.3 A sample heap snapshot 9

3.4 Illustrations of reference rules 10

4.1 The proposed agent communicates with a JVM via the JVM Tool Interface 13

4.2 Structure diagram of the proposed two-phase memory leak detection agent 14

4.3 Dialog of phase one (offline analysis) 15

4.4 Heap snapshot of our sample program MyProgram 16

4.5 The dominator tree converted from the heap snapshot shown in Figure 4.4 17

4.6 Home dialog of phase two (online monitoring) 18

4.7 Report dialog of phase two (online monitoring) 19

5.1 Time overhead measurement of RuntimeExp 30

List of Tables

4.1 The record of the setting object of our sample program MyProgram 22

5.1 Online monitoring report generated after comparing 12MB jar files for five times 25

5.2 Space required to store information of each object: $21 + a + 4(b + c + d)$ bytes 26

5.3 Space required by the Record Component to store information of each object: $24 + e$ bytes 27

5.4 Space required by the Reporter Component to store information of each object: $28 + f$ bytes 28

5.5 Result of the time overhead experiment 29

Chapter 1

Introduction

Memory leaks occur in a software program which fails to reclaim memory storages that are no longer in use. Such leaks may gradually exhaust available memory resources, degrade system performance and eventually crash the whole system. These are threats to long-running systems like those providing customer services in telecom operations. System performance degradation caused by memory leaks is typically observed after execution for hours, days or even weeks. These leaks may not be easily reproduced, and are thus difficult to be identified and fixed during development phase. Memory leaks are caused by the following two software defects:

1. *Lost references*: Programs neglect to free allocated memory storages before making them unreachable through any reference.
2. *Useless references*: Programs keep references to memory storages that will never be used again.

Leaks caused by lost references can be automatically detected and fixed by garbage collectors [1]. This paper will focus on the useless reference issue. For the demonstration purpose, we assume that the programs are written in the Java language, where the unit for leak detection is an object created in the programs.

In recent years, memory leaks due to useless references have been intensively discussed in the literature, and existing detection approaches can be categorized into either online or offline. Offline tools detect leaks with heap snapshots by analyzing types, counts and sizes of objects and the references among them [2, 3]. Online detection tools monitor heap states and analyze changes over time to find leak candidates [4, 5, 6].

This paper proposes a two-phase approach to detect memory leaks. During phase one, a heap dump of the investigated program is analyzed offline to decide a list of potential leak candidates. In phase two, the investigated program is executed along with an agent we develop. This agent tracks memory usage status of the leak candidates obtained from phase one at runtime. The time and location where each of these monitored objects is last accessed are reported to the users. This two-phase approach intends to reduce runtime overhead of online analysis and provide a more precise result.

The paper is organized as follows. Chapter 2 describes the related work regarding memory leak detection. Chapter 3 discussed two methods the proposed approach uses to inspect a heap. Chapter 4 proposes the algorithm and implementation of our approach. Chapter 5 evaluates our approach's effectiveness and overheads. The conclusions are given in Chapter 6.

Chapter 2

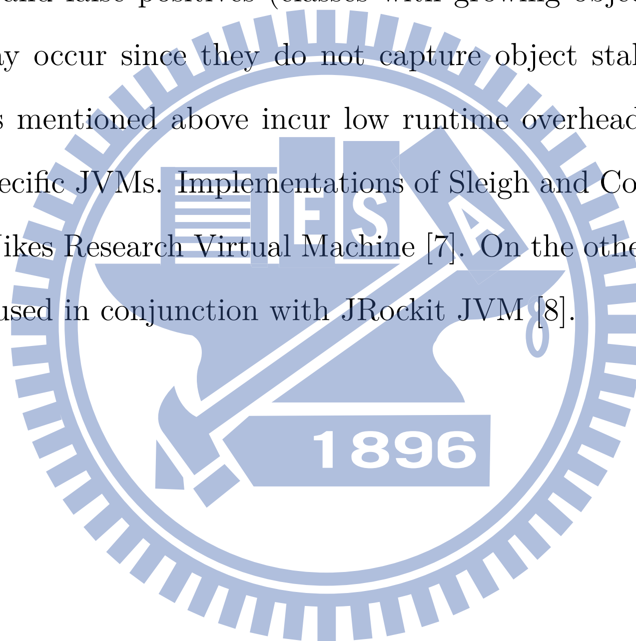
Related Work

This chapter introduces offline and online memory leak detection techniques previously proposed.

Offline memory leak detection approaches find memory leak candidates without additional runtime overhead to the programs being investigated. A graph mining approach was proposed by Maxwell et al [2], where a heap snapshot is first converted to a directed graph by transforming object instances to vertex and references between objects to directed edges. The directed graph is then converted to a dominator tree to reduce edges and optimize graph mining. Leak candidates are determined by mining the frequent subgraphs in the dominator tree. This graph mining approach provides users with insights to potential leaking data structures, and could be helpful when users do not possess *a priori* knowledge about the internal structure of the investigated program. The approach may report *false positives* (frequently occurring object instances that are not leaks) due to the lack of object instances' last accessed time information. *False negatives* (object instances which take up much space but do not occur frequently) may also happen since object instances' occupied space information is not taken into consideration.

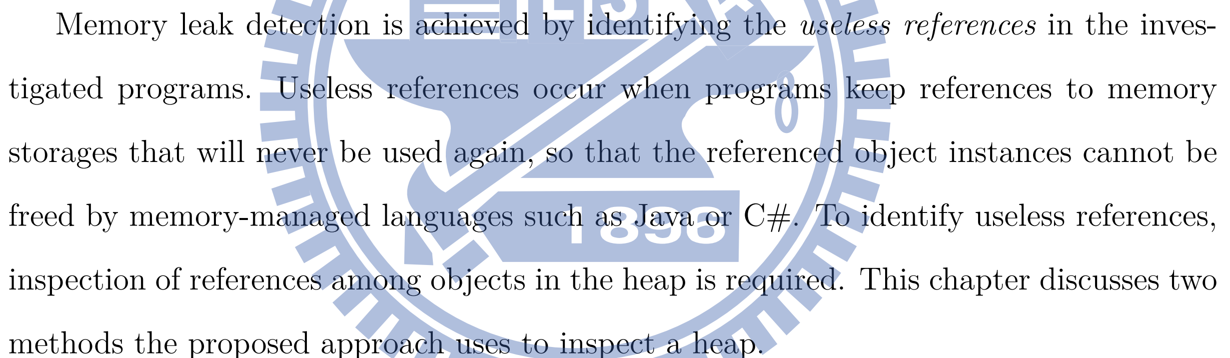
Online memory leak detection approaches find memory leak candidates at runtime with additional time and/or space overheads when executing the investigated programs.

Sleigh [4] maintains per-object staleness information (time since the object is last used) during program execution and used an encoding technique to record locations of code that last accessed the object. Sleigh produces no false positives. False negative may occur since this approach could recover only code locations that generate numerous stale objects. Locations that generate few stale objects are not recoverable due to the lossy nature of its encoding technique. Cork [5] and JRockit Mission Control [6] reduce time complexity of leak detection by modifying Java Virtual Machine (JVM)'s core functions. These tools detect the classes whose object instances grow over time, and mark them as leak candidates. False negatives (classes object instances do not grow but contain stale object instances) and false positives (classes with growing object instances which never become stale) may occur since they do not capture object staleness information. The online approaches mentioned above incur low runtime overhead with the aid of special mechanisms in specific JVMs. Implementations of Sleigh and Cork's approaches required modifications to Jikes Research Virtual Machine [7]. On the other hand, JRockit Mission Control must be used in conjunction with JRockit JVM [8].



Chapter 3

Concept of Referencing Among Objects



Memory leak detection is achieved by identifying the *useless references* in the investigated programs. Useless references occur when programs keep references to memory storages that will never be used again, so that the referenced object instances cannot be freed by memory-managed languages such as Java or C#. To identify useless references, inspection of references among objects in the heap is required. This chapter discusses two methods the proposed approach uses to inspect a heap.

3.1 Total Occupied Bytes

One goal of the proposed approach is to provide users with memory space usage information of the investigated program in the offline phase, so that users can decide which object instances are to be monitored in the online phase.

It is trivial to measure occupied bytes of each object instance. However, people are more interested in the total occupied bytes of a specific set of object instances that are connected by references. For example, when considering the occupied bytes of a list, most

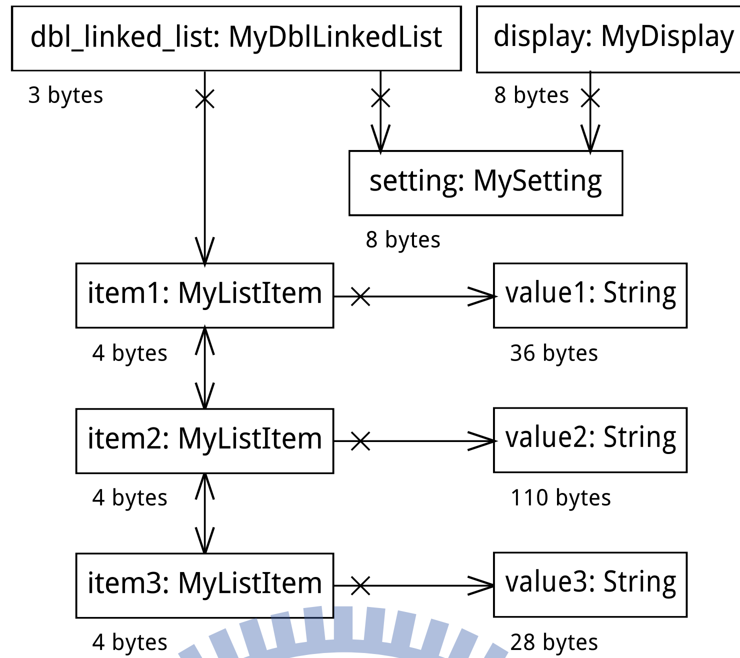


Figure 3.1: Heap snapshot of our sample program MyProgram

people are interested in the total occupied bytes of the object instances that compose of the list (i.e., all the list items) rather than just the occupied bytes of the list head. An intuitive solution to this problem is to sum up the occupied bytes of all reachable object instances from the list head. This intuitive solution may encounter two problems:

1. *Sum to infinity*: if there are circular references among a set of objects instances (e.g., doubly linked list), the sum adds up infinitely.

2. *Duplicate sums*: if multiple object instances reference the same object instance (e.g., an object storing the global setting of a program), the occupied bytes of this object are added to those of each object referencing it.

Figure 3.1 is an object diagram of the generated heap snapshot of our sample program MyProgram (whose source code can be found in Appendix A), described by the Unified Modeling Language (UML). In addition to the ordinary UML object diagram representation, a string is attached to the bottom left of each object instance to indicate the bytes occupied by this instance. An object instance setting (of class MySetting) stores global settings of MyProgram and is referenced by two other object instances display (of class

MyDisplay) and dbl_linked_list (of class MyDblLinkedList). Object display prints messages to STDOUT. dbl_linked_list is a doubly linked list which references its head list item item1. Each list item (i.e., item1, item2 or item3) is an object instance of class MyListItem which references to its predecessor list item, its successor list item, and an object instance (i.e., value1, value2 or value3) that stores its value (of class String).

To use the above mentioned intuitive solution to calculate the total occupied bytes of all object instances composing of dbl_linked_list in Figure 3.1, the sum to infinity problem occurs due to the bi-directional references between neighboring MyListItem object instances. Also the duplicated sums problem occurs because both dbl_linked_list and display maintain references to setting.

To solve these problems, the proposed approach computes the dominator tree from the heap snapshot. The heap snapshot is first converted to a directed graph before the dominator tree computation, where each object instance is converted to a vertex and each reference is converted to a directed edge. In a directed graph, node x dominates node y if all paths from a root to y go through x. This property can be used to represent a relationship between object instances in a heap. The relationship expresses which object instances are kept alive by a specific object instance. In addition, since the heap snapshot is converted to a tree structure, both sum to infinitiy and duplicated sums problems do not occur.

The proposed approach then adopts the aforementioned intuitive solution and use this dominator tree to calculate the *total occupied bytes* of each dominator tree node. The *total occupied bytes* of a dominator tree node is defined as "the bytes occupied by its corresponding object instance" plus "the total bytes occupied by its subtree nodes' corresponding object instances".

Figure 3.2 is the dominator tree computed from Figure 3.1, where each dominator tree node is labelled with its corresponding object instance's occupied bytes and its total occupied bytes. Each rectangle in the graph represents a node in the dominator tree. The first line in the rectangle describes the corresponding object instance's name and

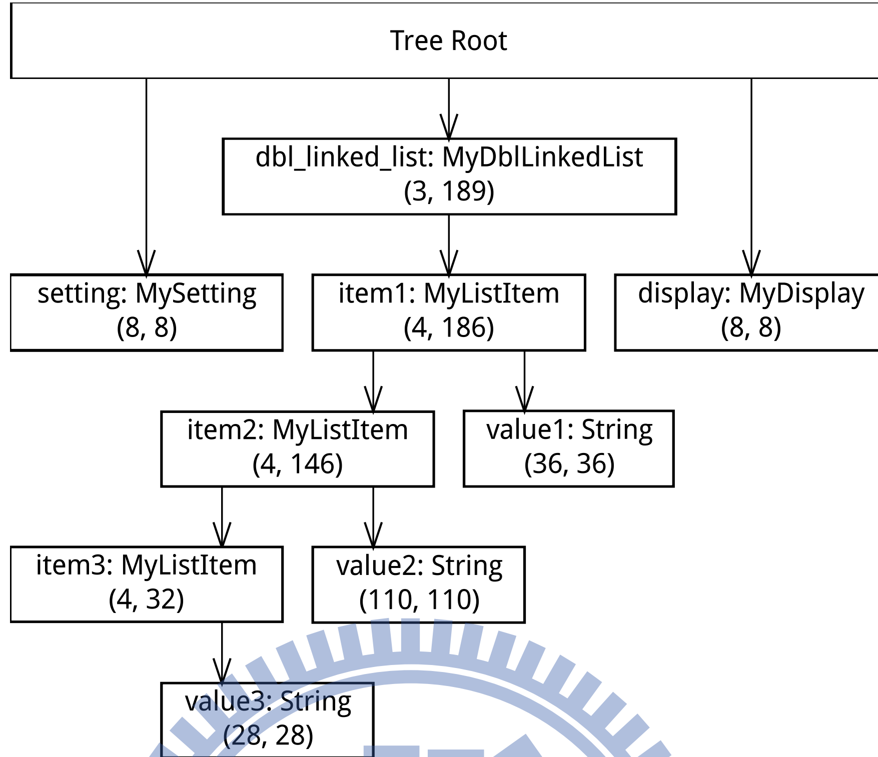


Figure 3.2: An illustration of total occupied bytes of dominator tree nodes

class. The second line in the rectangle is an ordered 2-tuple which lists the bytes occupied by the corresponding object instance and the total occupied bytes of this node. Take the item1:MyListItem node as an example. Its 2-tuple (4, 186) indicates that the bytes occupied by this MyListItem object instance are 4 bytes and the total occupied bytes of this node are 186 bytes. The number of the total occupied bytes, 186, is obtained by summing the storage in bytes occupied by this node's object instance (which are 4 bytes) and the total bytes occupied by the node's subtree nodes' corresponding object instances (listed in breadth-first order): $4 + (4 + 36 + 4 + 110 + 28) = 186$.

3.2 Reference Paths and Reference Rules

One feature of the proposed approach is to specify object instances to be monitored in the online phase. It is trivial to specify all object instances of a specific class. However, in real applications users are usually more concerned about a certain subset of instances of the specific class. Take the usage of the String class's object instances for example. It

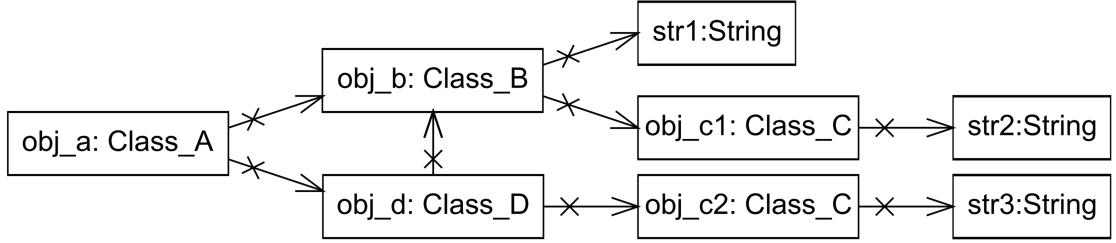
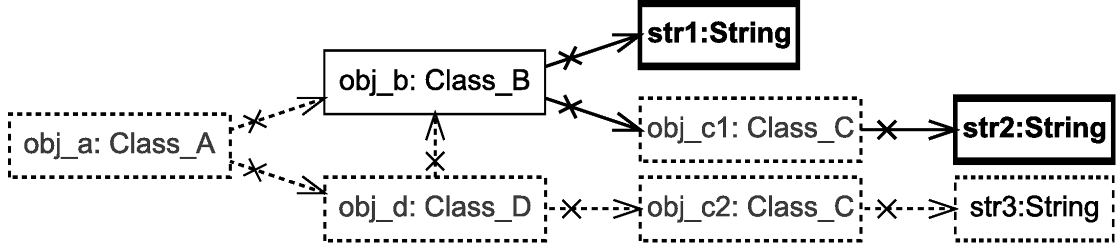


Figure 3.3: A sample heap snapshot

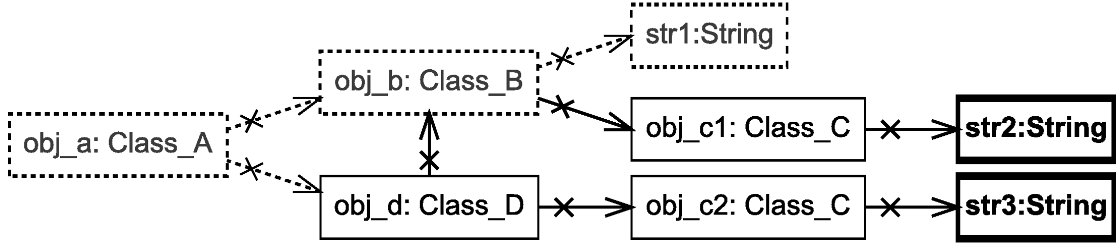
is common to see classes with member fields of type String (e.g., fields containing textual attributes of the object instance). Instead of monitoring all String object instances in the heap, users may prefer to focus on a subset of these instances, such as the String instances referenced by object instances of some other specific class (e.g., a List class).

The proposed approach selects object instances to be monitored online by examining each object instance's *reference path*. A reference path of an object instance obj_n is an ordered sequence of object instances $(obj_1, obj_2, ..., obj_n)$ such that there is a reference from object instance obj_i to object instance obj_{i+1}, where $1 \leq i \leq n-1$. Figure 3.3 is a sample heap snapshot represented as an object diagram in the UML. The set of reference paths of object str2 is {(obj_a, obj_b, obj_c1, str2), (obj_a, obj_d, obj_b, obj_c1, str2), (obj_d, obj_b, obj_c1, str2), (obj_b, obj_c1, str2), (obj_c1, str2), (str2)}. With the reference paths of each object instance calculated, we can specify to monitor those instances with certain reference relationships among other instances, e.g. to monitor only String objects that are reachable from Class_C objects.

A *reference rule* is a rule we use to match the reference paths and in turn to specify the object instances to be monitored in the online phase. A reference rule is represented as an ordered sequence of class names $(cls_1, cls_2, ..., cls_m)$ such that from each of its class name cls_j, there exists at least one path from object instances of class cls_j to object instances of class cls_{j+1}, where $1 \leq j \leq m-1$. Let $getClassName(object)$ be a function that returns the class name of the specified object. A reference path $p = (obj_1, obj_2, ..., obj_n)$ matches a reference rule $r = (cls_1, cls_2, ..., cls_m)$ if r is a subsequence of $p' = (getClassName(obj_1), getClassName(obj_2), ..., getClassName(obj_n))$



(a) Reference rule $r_1 = (\mathsf{Class_B}, \mathsf{String})$



(b) Reference rule $r_2 = (\mathsf{Class_D}, \mathsf{Class_C}, \mathsf{String})$

Figure 3.4: Illustrations of reference rules

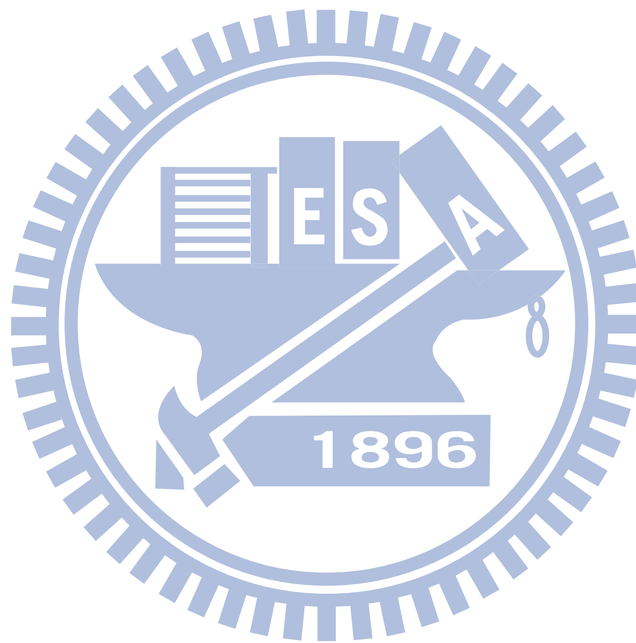
and $getClassName(obj_n) = cls_m$.

Figure 3.4 illustrates to reference rules r_1 and r_2 with the sample heap snapshot in Figure 3.3, where rule $r_1 = (\mathsf{Class_B}, \mathsf{String})$ and rule $r_2 = (\mathsf{Class_D}, \mathsf{Class_C}, \mathsf{String})$. Each solid-lined instances in Figure 3.4 corresponds to one class name in the reference rules. Solid-lined instances with bold text and thick border are object instances to be monitored in the online phase (notice that they correspond to the last class names in the reference rules). The solid-lined arrows show the reachability among object instances of those class names in the rules. Further descriptions of these reference rule illustrations are stated below.

Figure 3.4(a) ($r_1 = (\mathsf{Class_B}, \mathsf{String})$): The object instances specified to be monitored by r_1 are String objects str1 and str2. The reference paths of object str1 that match r_1 are (obj_a, obj_b, str1), (obj_a, obj_d, obj_b, str1) and (obj_d, obj_b, str1). The reference paths of object str2 that match r_1 are (obj_a, obj_b, obj_c1, str2), (obj_a, obj_d, obj_b, obj_c1, str2), (obj_d, obj_b, obj_c1, str2) and (obj_b, obj_c1, str2).

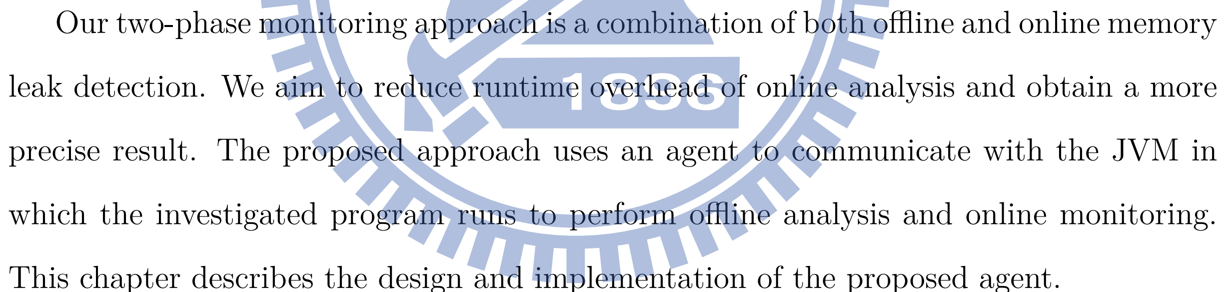
Figure 3.4(b) ($r_2 = (\mathsf{Class_D}, \mathsf{Class_C}, \mathsf{String})$): The object instances specified to be monitored by r_2 are String objects str2 and str3. The reference paths of object str2

that match r_2 are (obj_a, obj_d, obj_b, obj_c1, str2) and (obj_d, obj_b, obj_c1, str2). The reference paths of object str3 that match r_2 are (obj_a, obj_d, obj_c2, str3) and (obj_d, obj_c2, str3).



Chapter 4

Design and Implementation of Two-Phase Online Memory Leak Detection



Our two-phase monitoring approach is a combination of both offline and online memory leak detection. We aim to reduce runtime overhead of online analysis and obtain a more precise result. The proposed approach uses an agent to communicate with the JVM in which the investigated program runs to perform offline analysis and online monitoring. This chapter describes the design and implementation of the proposed agent.

4.1 Overview

The proposed agent investigates a Java program by communicating with the JVM in which the investigated program runs. In our implementation, the agent communicates with a Java SE HotSpot JVM Version 20.0 [9] via the JVM Tool Interface (JVMTI) Version 1.2 [10] provided by the JVM to obtain the investigated program's memory usage information, as shown in Figure 4.1. This section presents a structural overview of the proposed agent.

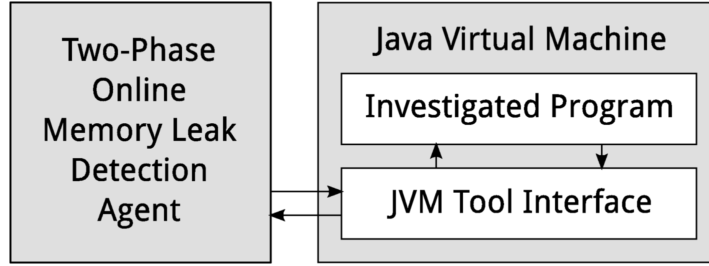


Figure 4.1: The proposed agent communicates with a JVM via the JVM Tool Interface

4.1.1 Structure Diagram

Figure 4.2 is the structure diagram of our agent. The agent consists of two parts: the Phase One Part for offline analysis and the Phase Two Part for online monitoring. Each part consists of two modules: the User Interface Module and the Core Module. Users operate the agent through the User Interface Module, which communicates with the Core Module to perform the two-phase memory leak detection.

Details of the Phase One Part and the Phase Two Part are described in the following two subsections.

4.1.2 Phase One: Offline Analysis

During phase one, the user executes the Phase One Part of the proposed agent, and uses the Phase One Dialog Component to select a program to investigate. The agent then generates a heap snapshot of the investigated program, and analyzes the snapshot offline by converting the heap snapshot to a dominator tree. This dominator tree information is displayed as a tree-style checklist by the Phase One Dialog Component, enabling the user to choose classes of object instances for online monitoring. The selected classes are outputted to a text file which will be imported in phase two for online monitoring.

Heap snapshot generation and analysis may incur runtime overhead. Therefore it is suggested to execute the Phase One Part right before shutting down the investigated program.

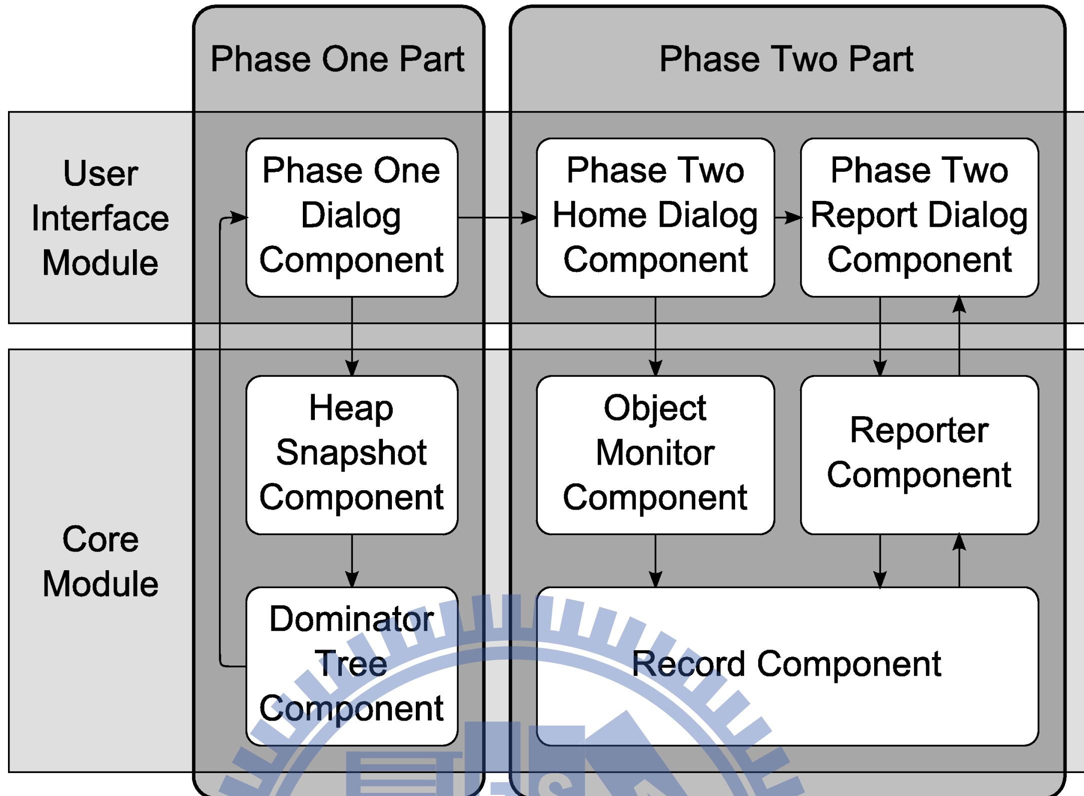


Figure 4.2: Structure diagram of the proposed two-phase memory leak detection agent

4.1.3 Phase Two: Online Monitoring

During phase two, the user executes the Phase Two Part of the proposed agent, and uses the Phase Two Home Dialog Component to import the text file outputted by the Phase One Part. This imported file determines the object instances to be monitored. Then the user selects a program and starts monitoring it. The Object Monitor Component tracks memory usage status of the selected object instances, and records it via the Record Component. The user can choose to generate a report at any time using the Phase Two Report Dialog Component. Reports are provided by the Reporter Component, which communicates with the Record Component to obtain the current memory usage status and generates reports. To help the user to identify the potential leak in the source code, each report shows the object instance count, last accessed time, last accessed location of each selected object instance.

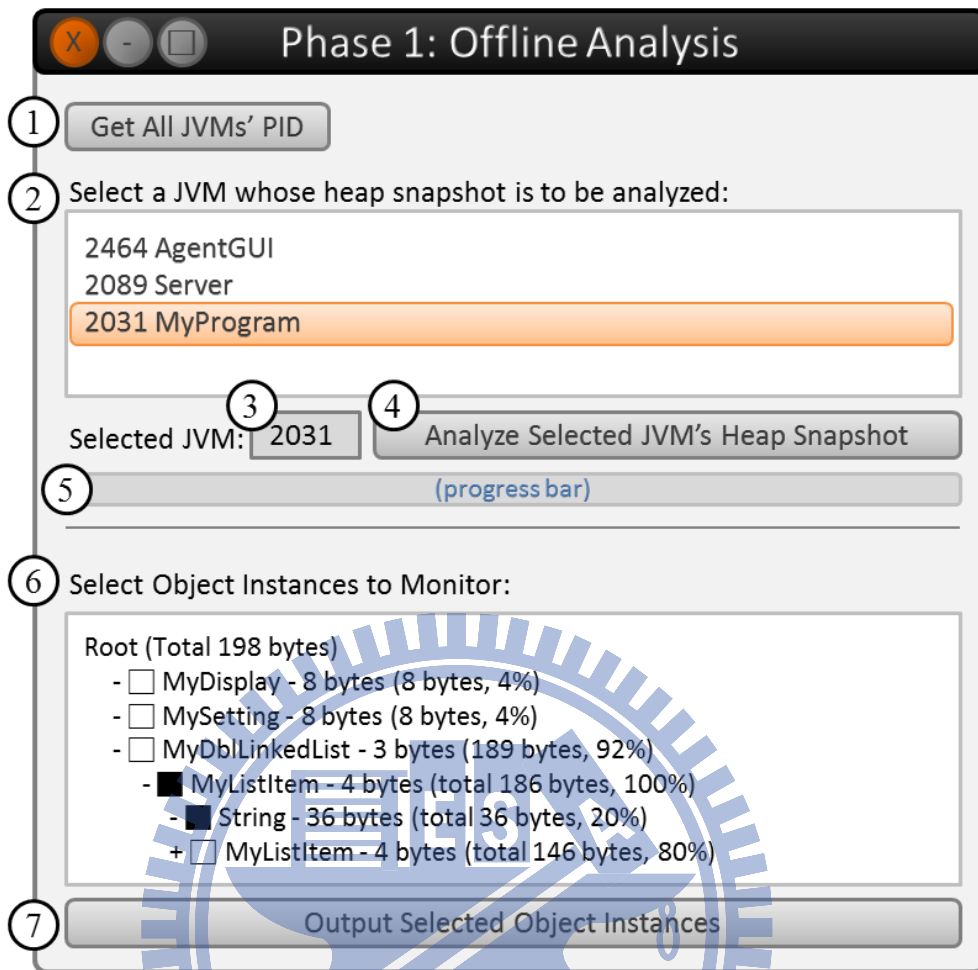


Figure 4.3: Dialog of phase one (offline analysis)

4.2 User Interface Module

The User Interface Module renders a graphical user interface (GUI) that contains one dialog component for phase one (offline analysis) and two dialog components for phase two (online monitoring). The following subsections describe these dialogs and demonstrate how to use the agent to investigate our sample program MyProgram via these dialogs.

4.2.1 Phase One Dialog Component

Figure 4.3 shows the dialog of phase one (offline analysis). During phase one, the agent generates and analyzes the heap snapshot of the investigated program. In our case, a JVM process, in which a Java program runs, is investigated.

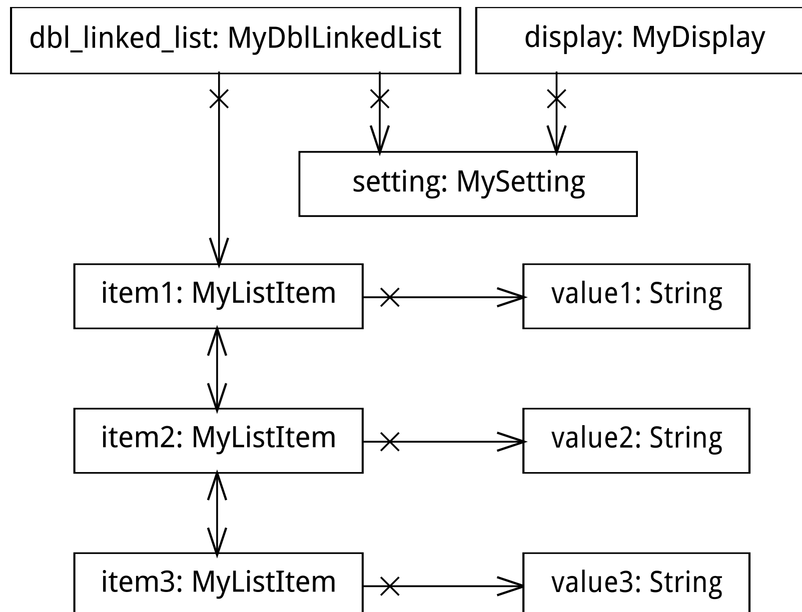


Figure 4.4: Heap snapshot of our sample program MyProgram

Since there may be numerous JVM processes executing concurrently, the user has to select a JVM process to monitor. When the user clicks the "Get All JVMs' PID" button (Figure 4.3 ①), a list of currently running JVMs, including their process IDs (PIDs) and names, is then shown in Figure 4.3 ②. The user may select a specific JVM (e.g., MyProgram) and its PID will be shown in the "Selected JVM" textbox (PID 2031 in Figure 4.3 ③).

When the user clicks the "Analyze Selected JVM's Heap Snapshot" button (Figure 4.3 ④), the agent starts to generate a heap snapshot for the selected program and analyzes it. Heap snapshot generation and analysis can take quite a while, and the overall progress will be reported by the progress bar below (Figure 4.3 ⑤). Details of heap snapshot generation and analysis will be given in Subsection 4.3.1 and Subsection 4.3.2.

Figure 4.4 is an object diagram of the generated heap snapshot of our sample program MyProgram, described by the UML. Its content is the same as Figure 3.1 described in Section 3.1. The agent then analyzes this heap snapshot and converts it to a dominator tree, whose structure is shown in Figure 4.5. This dominator tree information is displayed by the GUI in a tree-style checklist (Figure 4.3 ⑥), from which the user can select the object instances to be monitored in phase two. To help users in choosing object instances

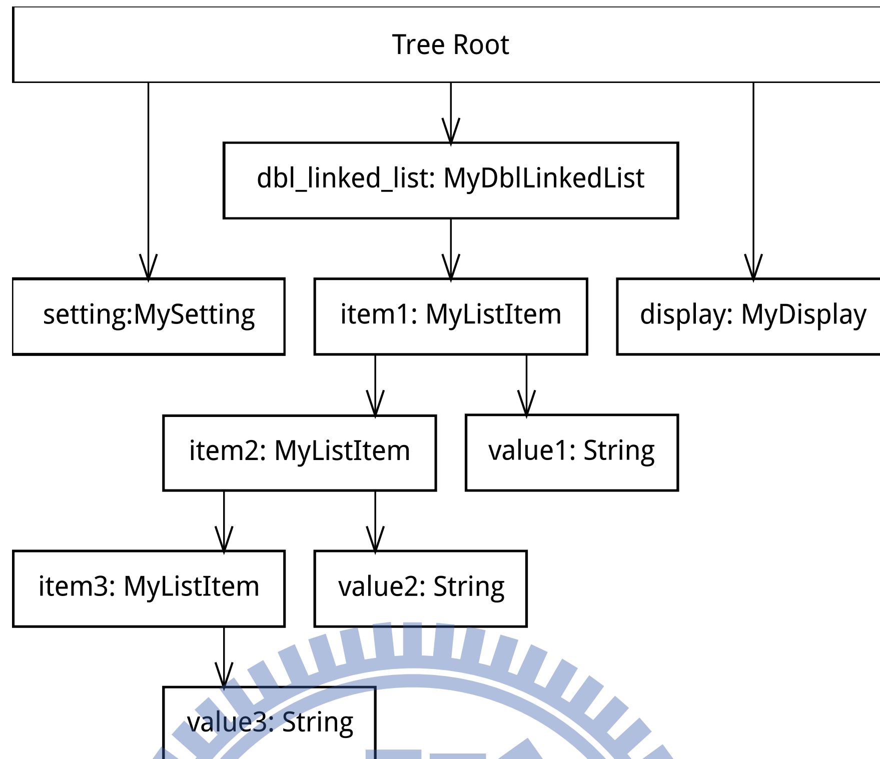


Figure 4.5: The dominator tree converted from the heap snapshot shown in Figure 4.4

to monitor, the agent calculates the total occupied bytes of each dominator tree node, and illustrates it in Figure 4.3 ⑥. In the checklist, each entry represents a dominator tree node, displaying the class name of its object instance, the total occupied bytes, and the percentage of its total occupied bytes versus the sum of all nodes' total occupied bytes on the same level in the same subtree. At first only nodes on the first tree level are shown. The minus symbol (-) in front of a checklist entry indicates that the entry is a leaf node and is not expandable. By clicking the plus symbol (+) in front of a checklist entry, the user can expand and examine the entry's subtree.

With the occupied bytes information, the user can determine object instances that are potential leaks. In our sample, MyDblLinkedList takes up 92% of the heap space, which seems suspicious. Therefore the user selects its child node MyListItem and its grandchild node String, and clicks the "Output Selected Object Instances" button (Figure 4.3 ⑦). The agent then converts the hierarchical information of selected entries to reference rules (definition of reference rule is described in Section 3.2), and outputs them to a text file. This output file will be used in phase two.

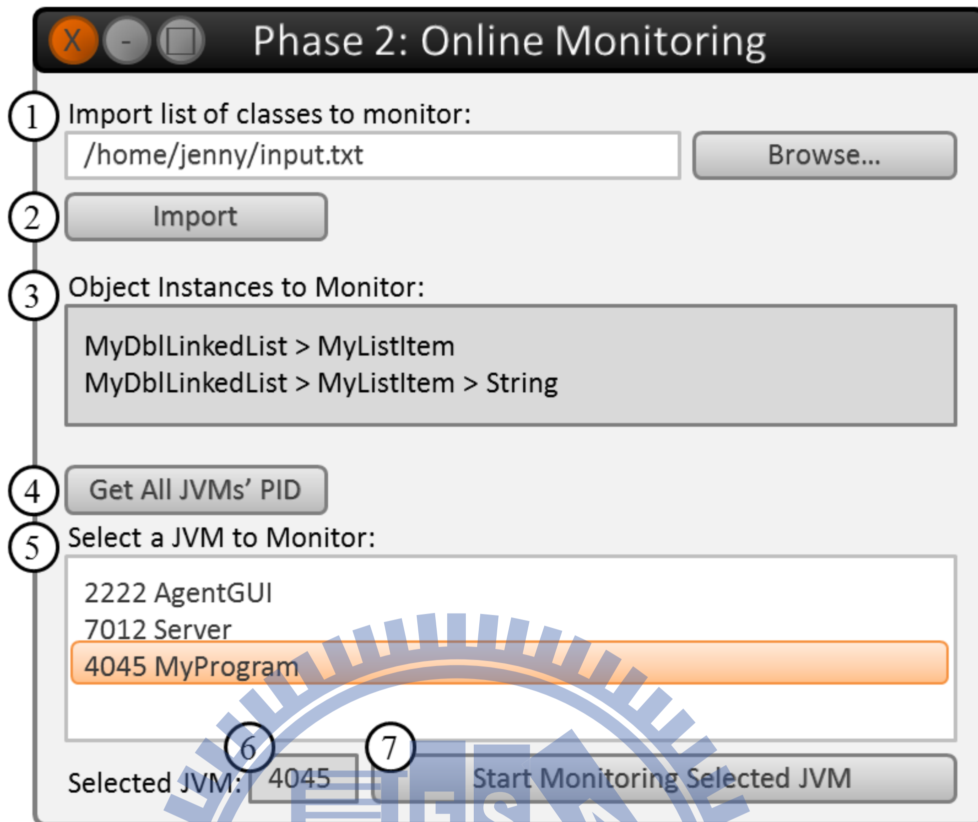


Figure 4.6: Home dialog of phase two (online monitoring)

4.2.2 Phase Two Dialog Components

Figure 4.6 shows the home dialog of phase two (online monitoring). Through this dialog, the user specifies the object instances he or she wants to monitor by importing the output file of Phase One Dialog, and selects a JVM to start online monitoring.

The user first locates the output file of Phase One Dialog (Figure 4.6 ①), and clicks the "Import" button (Figure 4.6 ②). The content of the file is listed in the "Object Instances to Monitor:" textbox below (Figure 4.6 ③).

To select a specific running JVM, the user clicks the "Get All JVMs' PID" button (Figure 4.6 ④). A list of currently running JVMs, including their PIDs and names, is then shown below, which can be selected by the user (Figure 4.6 ⑤). The PID of the selected entry is shown in the "Selected JVM" textbox (Figure 4.6 ⑥). In our example, program MyProgram with PID 4045 is selected.

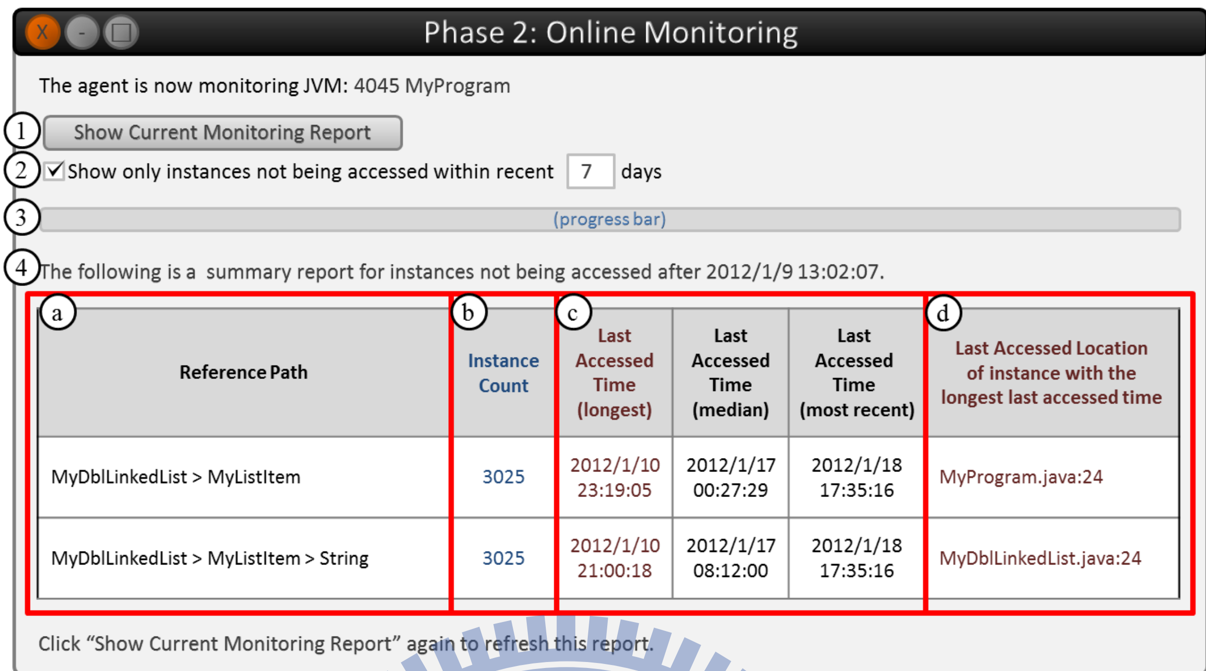


Figure 4.7: Report dialog of phase two (online monitoring)

When the user clicks the “Start Monitoring Selected JVM” button (Figure 4.6 ⑦), the home dialog is closed and the second dialog (Figure 4.7) pops out, indicating that the agent starts online monitoring the chosen JVM process.

The user can click the “Show Current Monitoring Report” button (Figure 4.7 ①) at any time to examine the current monitoring report. At the same time, the user can request the agent to report only those object instances not being accessed within a specific day range (Figure 4.7 ②). After the user clicks the “Show Current Monitoring Report” button, a report is generated by the Reporter Component as shown below in Figure 4.7 ④. This process might take a while and the progress is reported by the progress bar (Figure 4.7 ③).

Figure 4.7 ④ displays a summary table for the status of the reported object instances. The columns of this table are explained below.

Reference Path: the reference path of the monitored object instances. (Figure 4.7 ⓐ)

Instance Count: the number of monitored object instances with the same reference path. (Figure 4.7 ⓑ)

Last Accessed Time: the last accessed times (i.e., longest, median and most recent) of monitored object instances with the same reference path (Figure 4.7 ⓒ). The object instance corresponding with the maximum (longest) time value is intuitively considered the most potential to be a leak. In addition, median and minimum (most recent) time values are also listed for user reference.

Last Accessed Location: the last accessed location (including the file name and the line number) of the object instance with the maximum (longest) last accessed time (Figure 4.7 ⓓ). This column helps the user identify the potential leak in the source code.

The user can click the "Show Current Monitoring Report" button (Figure 4.7 ①) at any time to refresh the report.

4.3 Core Module

The Core Module communicates with the JVM in which the investigated program runs via the JVMTI to perform offline analysis and online monitoring. The following subsections describe components of the Core Module.

4.3.1 Heap Snapshot Component

The Heap Snapshot Component belongs to the Phase One Part of the proposed agent. It creates a heap snapshot of the JVM process selected by the user. The Heap Snapshot Component obtains the JVM's heap information by invoking the *FollowReferences* function provided by the JVMTI.

A heap snapshot generated by the Heap Snapshot Component is essentially a directed graph. Object instances are vertex and references between objects are directed edges in the generated directed graph. This generated directed graph is then passed to the

Dominator Tree Component for further processing, which will be described in the next subsection.

4.3.2 Dominator Tree Component

The Dominator Tree Component belongs to the Phase One Part of the proposed agent. It receives the directed graph generated by the Heap Snapshot Component, and converts it to a dominator tree. The dominance algorithm proposed by Cooper et al. [11] is adopted for conversion from a directed graph to a dominator tree.

The dominator tree structure is used to calculate the *total occupied bytes* of each object. Details of total occupied bytes calculation are described in Section 3.1. This total occupied bytes information is then passed to the Phase One Dialog Component, which helps the user determine object instances that are potential leaks.

4.3.3 Object Monitor Component

The Object Monitor Component belongs to the Phase Two Part of the proposed agent. It monitors the selected object instances at runtime.

The user selects object instances to be monitored via the Phase One Dialog Component (see Subsection 4.2.1), which are then stored as a list of reference rules. These reference rules are imported via the Phase Two Home Dialog Component (see Subsection 4.2.2) and passed to the Object Monitor Component. The component then traverses all object instances in the heap and calculates reference paths of each object instance. For each object instance, if there is at least one reference path matching any of the reference rules specified by the user, then the component monitors the usage status of this object at runtime. Details of reference paths and reference rules can be found in Section 3.2.

The Object Monitor Component monitors the usage status of a specified object instance by invoking the *SetFieldAccessWatch* and *SetFieldModificationWatch* functions

Table 4.1: The record of the setting object of our sample program MyProgram

Object Instance	Reference Path(s)	Last Accessed Time	Last Accessed Location
setting	dbl_linked_list > setting	2012-01-10 21:00:18	MyProgram.java:30
	display > setting		

provided by the JVMTI. JVMTI in turn generates *FieldAccess* events or *FieldModification* events when object instances are used. For each monitored object instance, there is a corresponding record in the Record Component which records the object instance's last accessed time and location. When the monitored object instance is accessed, the Object Monitor Component informs the Record Component to update the object's last accessed time and location. When the monitored object is freed, an *ObjectFree* event is generated and the Object Monitor Component informs the Record Component to remove the freed object's record.

4.3.4 Record Component

The Record Component belongs to the Phase Two Part of the proposed agent. It stores records of monitored object instances at runtime. The records are created, updated and deleted by the Object Monitor Component, and read by the Reporter Component to generate monitoring reports to the user.

Table 4.1 illustrates how the Record Component stores the record of the setting object of our sample program MyProgram (whose heap snapshot is shown in Figure 4.4). For each monitored object instance, the Record Component stores the following information:

Object Instance: the ID of the object instance.

Reference Paths: the reference paths of the monitored object instance. All reference paths matching the specified reference rules are stored.

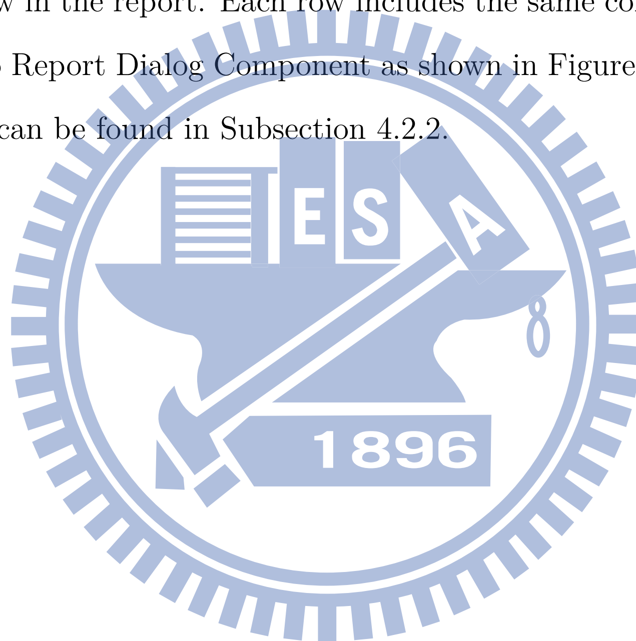
Last Accessed Time: the last accessed time of the monitored object instance.

Last Accessed Location: the last accessed location (including the file name and the line number) of the monitored object instance.

4.3.5 Reporter Component

The Reporter Component belongs to the Phase Two Part of the proposed agent. When the user clicks the "Show Current Monitoring Report" button of the Phase Two Report Dialog Component (see Subsection 4.2.2), the Reporter Component retrieves records from the Record Component and generates a report of current status of the monitored object instances. The Reporter Component then passes the generated report to the Phase Two Report Dialog Component for display.

The Reporter Component aggregates records by reference paths. For each reference path there is a row in the report. Each row includes the same columns as those displayed by the Phase Two Report Dialog Component as shown in Figure 4.7 ④, and descriptions of these columns can be found in Subsection 4.2.2.


ES A 1896

Chapter 5

Performance Evaluation

This chapter evaluates the proposed approach's effectiveness and overheads. To test the feasibility of the proposed agent, we conduct experiments on Eclipse 3.1.2, which is reported to have a memory leak bug. And then we conduct experiments on a tiny program RuntimeExp, which we design for more accurate time overhead evaluation. All our experiments are conducted on a 3.07GHz Intel(R) Core(TM) i7 CPU and 12GB of main memory, running Linux 2.6.32. The investigated programs run in Java SE HotSpot JVMs (Version 20).

5.1 Locating Memory Leak in Eclipse 3.1.2

In this section, we use Eclipse 3.1.2 to test the feasibility of the proposed agent. Eclipse 3.1.2 is reported to have a memory leak bug [12]. Memory leaks are observed when a user repeatedly conducts file comparisons. The leak occurs in Eclipse's NavigationHistory component, which keeps a list of NavigationHistoryEntry objects. Each NavigationHistoryEntry object points to a NavigationHistoryEditorInfo object, which in turn keeps reference to a data structure that holds the results of the comparison. In some cases the NavigationHistory component fails to remove the NavigationHistoryEditorInfo objects after the window is closed, and thus keeps alive all the other connected objects.

Table 5.1: Online monitoring report generated after comparing 12MB jar files for five times

Reference Path	Count	Last Accessed Time			Last Accessed Location
		longest	median	most recent	
ROOT > org.eclipse.ui.internal.NavigationHistoryEntry	5	2012-06-21 14:50:51	2012-06-21 15:52:36	2012-06-21 16:51:13	org.eclipse.ui.internal .NavigationHistoryEntry.java:163
ROOT > org.eclipse.ui.internal.NavigationHistoryEntry > org.eclipse.ui.internal.NavigationHistoryEditorInfo	5	2012-06-21 14:50:51	2012-06-21 15:52:36	2012-06-21 16:51:13	org.eclipse.ui.internal .NavigationHistoryEntry.java:163

We reproduce this leak by consecutively comparing two jar files of size 12MB for five times, and monitor it with the Phase Two Part of our agent. The chosen reference rules are:

1. (org.eclipse.ui.internal.NavigationHistoryEntry)
2. (org.eclipse.ui.internal.NavigationHistoryEntry, org.eclipse.ui.internal.NavigationHistoryEditorInfo)

The generated report is shown in Table 5.1. It can be seen in the report that there are NavigationHistoryEntry and NavigationHistoryEditorInfo objects not being used for two hours. The last accessed locations of these objects are also reported. Thus we believe that the proposed agent is able to provide helpful clues for developers in fixing leaks.

5.2 Space Overhead Analysis

The proposed agent incurs space overhead during both phase one and phase two. Details are described below.

5.2.1 Phase One (Offline Analysis)

Space overhead occurs in phase one due to the Heap Snapshot Component and the Dominator Tree Component.

For each object instance in the heap snapshot, a corresponding graph node is created and appended to an array in the Heap Snapshot Component. Each graph node stores

Table 5.2: Space required to store information of each object: $21 + a + 4(b + c + d)$ bytes

	Type	Bytes
Node ID	String	a
To Nodes	int array	$4b$
From Nodes	int array	$4c$
Discovering Status	char	1
Post-Order	int	4
Reverse Post-Order	int	4
Child Nodes	int array	$4d$
Immediate Dominator	int	4
Object Occupied Bytes	int	4
Total Occupied Bytes	int	4
Total		$21 + a + 4(b + c + d)$

its ID as a string and uses two integer arrays to maintain indices of nodes connected by directed edges to and from the node respectively (corresponding to the references in the heap snapshot).

The directed graph is converted to a dominator tree by the Dominator Tree Component. Each graph node uses 1 byte (char) to store the discovering status (i.e., *not discovered*, *being explored*, or *explored*) for graph traversal, 8 bytes (int) to store its post-order and reverse post-order respectively, and an integer array to maintain indices of its children. After visiting all the graph nodes, each graph node is converted to a tree node. Each tree node uses 4 bytes (int) to store the index of its immediate dominator node. To calculate the total occupied bytes of each tree node as described in Section 3.1, each tree node uses 4 bytes (int) to store the number of occupied bytes of its corresponding object instance and 4 bytes (int) to store its total occupied bytes.

Table 5.2 is a summary of the space required to store the information of each object. To sum up, $21 + a + 4(b + c + d)$ bytes are required for each object, where a is the number of characters of the node's ID, b and c are the node's indegree and outdegree respectively, and d is the number of the node's children in dominator tree.

Table 5.3: Space required by the Record Component to store information of each object: $24 + e$ bytes

	Type	Bytes
Node ID	jlong	8
Last Accessed Time	time_t	4
Last Accessed Location	jmethodID	4
	jlocation	8
Reference Paths	String array	e
Total		$24 + e$

5.2.2 Phase Two (Online Monitoring)

Space overhead occurs in phase two due to the Record Component and the Reporter Component.

For each monitored object instance, a corresponding record is created in the Record Component, as shown in Table 4.1. Each record uses 8 bytes (jlong) to store the ID for the monitored object instance, 4 bytes (time_t) to store the last accessed time and 12 bytes (jmethodID and jlocation) to store the last accessed location of the monitored object instance. In addition to these 24 bytes, each record stores reference paths matching the given reference rules as strings. Records in the Record Component are deallocated only when the corresponding object instances are freed. Table 5.3 is a summary of the space required by the Record Component to store the information of each object. To sum up, $24 + e$ bytes are required for each object, where e is the number of characters required to represent all reference paths of the node.

At runtime, records in Record Component change over time. To avoid corrupting these records, when a monitoring report is requested, the Reporter Component clones the records in the Record Component and thus incurs additional space overhead. The Reporter Component aggregates these cloned records by reference paths. For each reference path, a corresponding row is shown in the report as in Figure 4.7 ④. Each row uses a string to store the object reference path, 4 bytes (int) to store object instance count, 12 bytes to store 3 last accessed times (time_t) and 12 bytes (jmethodID and jlocation) to store the last accessed location. Table 5.4 is a summary of the space required by the

Table 5.4: Space required by the Reporter Component to store information of each object: $28 + f$ bytes

		Type	Bytes
Reference Path		String	f
Object Instance Count		int	4
Last Accessed Time	longest	time_t	4
	median	time_t	4
	most recent	time_t	4
Last Accessed Location		jmethodID	4
		jlocation	8
Total			$28 + f$

Reporter Component to store the information of each object. To sum up, $28+f$ bytes are required for each object, where f is the number of characters of the node's reference path. Memory spaces allocated by the Reporter Component do not last long because they are freed right after outputting the generated report to file.

For real-world applications, a user is supposed to watch a very tiny portion of all the objects during the online monitoring phase. Since the agent only keeps records for monitored objects, the space overhead incurred shall not be expensive in most cases.

5.3 Time Overhead Analysis

The time overhead is of crucial concern to an online monitoring technique because high time overhead may degrade the performance of the investigated program, making the technique an impractical solution for real-world applications.

To accurately evaluate the time overhead of the proposed agent, we develop a simple program RuntimeExp. The source code of RuntimeExp can be found in Appendix B. During each execution of RuntimeExp, the program first allocates a total of 100 objects in heap, part of which are RuntimeExpMonitored objects and the rest are RuntimeExpNotMonitored objects. The number of RuntimeExpMonitored objects are determined according to user input. The program then executes a loop for 1000 iterations. In each iteration, each object is accessed twice (read and written, respectively). We then measure the total execution

Table 5.5: Result of the time overhead experiment

Monitored Object Count	Total Number of Accesses of all Monitored Objects	Execution Time (ms)	Overhead
0	0	71	3%
1	2,000	913	1,219%
2	4,000	945	1,265%
10	20,000	1,024	1,379%
20	40,000	1,126	1,526%
30	60,000	1,254	1,711%
40	80,000	1,368	1,876%
50	100,000	1,462	2,012%
60	120,000	1,601	2,213%
70	140,000	1,706	2,364%
80	160,000	1,792	2,488%
90	180,000	1,912	2,662%
100	200,000	2,098	2,930%

time of these 1000 iterations of the RuntimeExp program with and without the Phase Two Part of our agent attached. When the agent is attached, all the RuntimeExpMonitored objects are monitored.

Table 5.5 shows the result of the time overhead experiment. For the RuntimeExp program executing without our agent attached, the average execution time of all 1000 iterations is 70ms; when the RuntimeExp program is executed with the proposed agent attached but monitors nothing in heap, the average execution time of all 1000 iterations is 71ms, which is very close to that without agent attached. This is because the Object Monitor Component does not have to handle any *FieldAccess* nor *FieldModification* events and thus no additional code is executed and no extra time overhead is incurred.

To better examine the relationship between total number of object accesses and execution time, we plot the data in Table 5.5 as Figure 5.1. It can be observed from the plot that the time overhead is linearly proportional to the number of accesses during program execution. In worst case where all objects are monitored, the total execution time may grow to nearly 30 times, which is not tolerable in real-world applications. However, with the help of the offline analysis phase of the proposed agent, the user is able to effectively select a small portion of the objects. Moreover, the leaked objects basically are objects not frequently used during the long-term execution of programs. Thus online monitoring

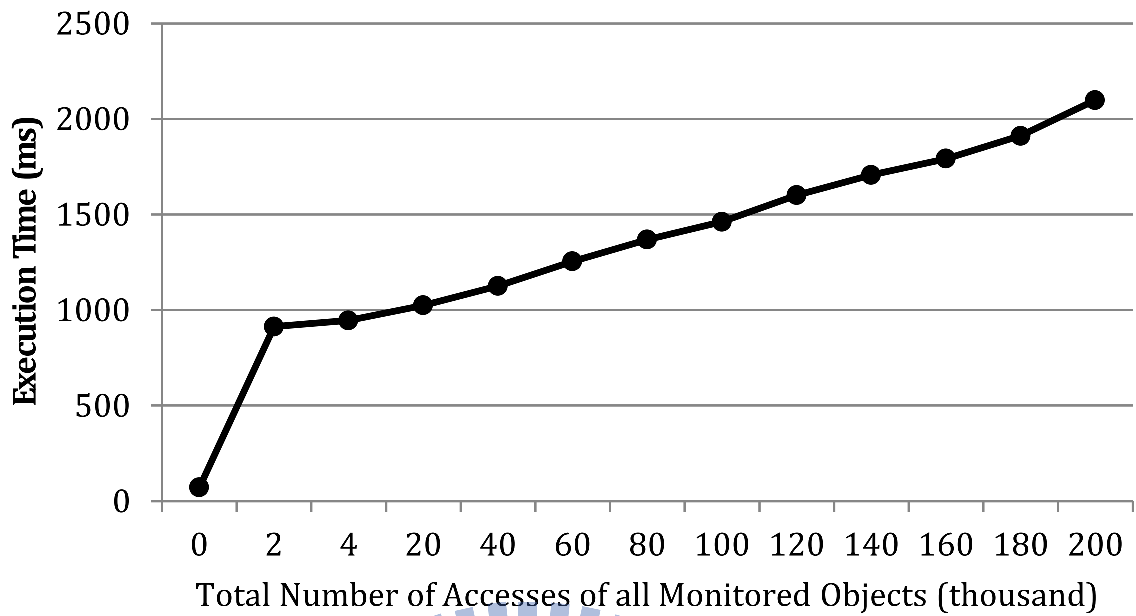
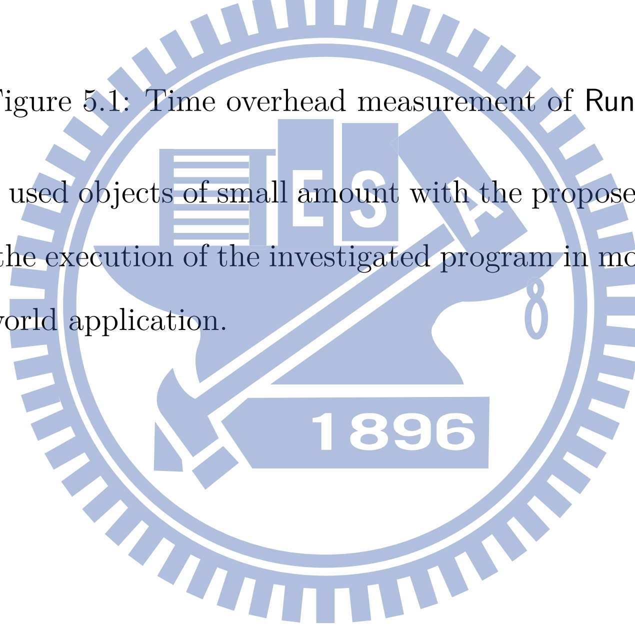


Figure 5.1: Time overhead measurement of RuntimeExp



these infrequently used objects of small amount with the proposed agent incurs very little time overhead to the execution of the investigated program in most cases, and is therefore suitable for real-world application.

Chapter 6

Conclusions

We propose a two-phase approach to detect memory leaks due to useless references, adopting both offline analysis and online monitoring techniques. The Phase One Part (offline analysis) helps the user determine a set of classes whose objects are leak candidates. These leak candidates are then monitored in the Phase Two Part (online monitoring), where the incurred time overhead can be significantly reduced at runtime with the benefit of partial monitoring.

For the demonstration purpose, we assume that the investigated programs are written in the Java language. The proposed approach uses an agent to communicate with the JVM. We test the proposed agent on Eclipse 3.1.2 and show that the report generated by the online monitoring phase of the agent is useful for locating the memory leak. Little per-object space overhead and nearly no time overhead are incurred during the execution of the investigated program due to partial monitoring, making it suitable for real-world applications.

As a final remark, the implementation of the proposed agent does not modify the JVM. Specifically, the proposed agent can be used in most JVMs that support JVMTI.

Bibliography

[1] R. Jones and R. Lins, *Garbage collection: algorithms for automatic dynamic memory management*. Wiley, 1996.

[2] E. K. Maxwell, G. Back, and N. Ramakrishnan, "Diagnosing memory leaks using graph mining on heap dumps," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '10. New York, NY, USA: ACM, 2010, pp. 115–124.

[3] Jprobe. Quest Software. [Online]. Available: http://www.quest.com/jprobe/

[4] M. D. Bond and K. S. McKinley, "Bell: bit-encoding online memory leak detection," *SIGPLAN Not.*, vol. 41, pp. 61–72, October 2006.

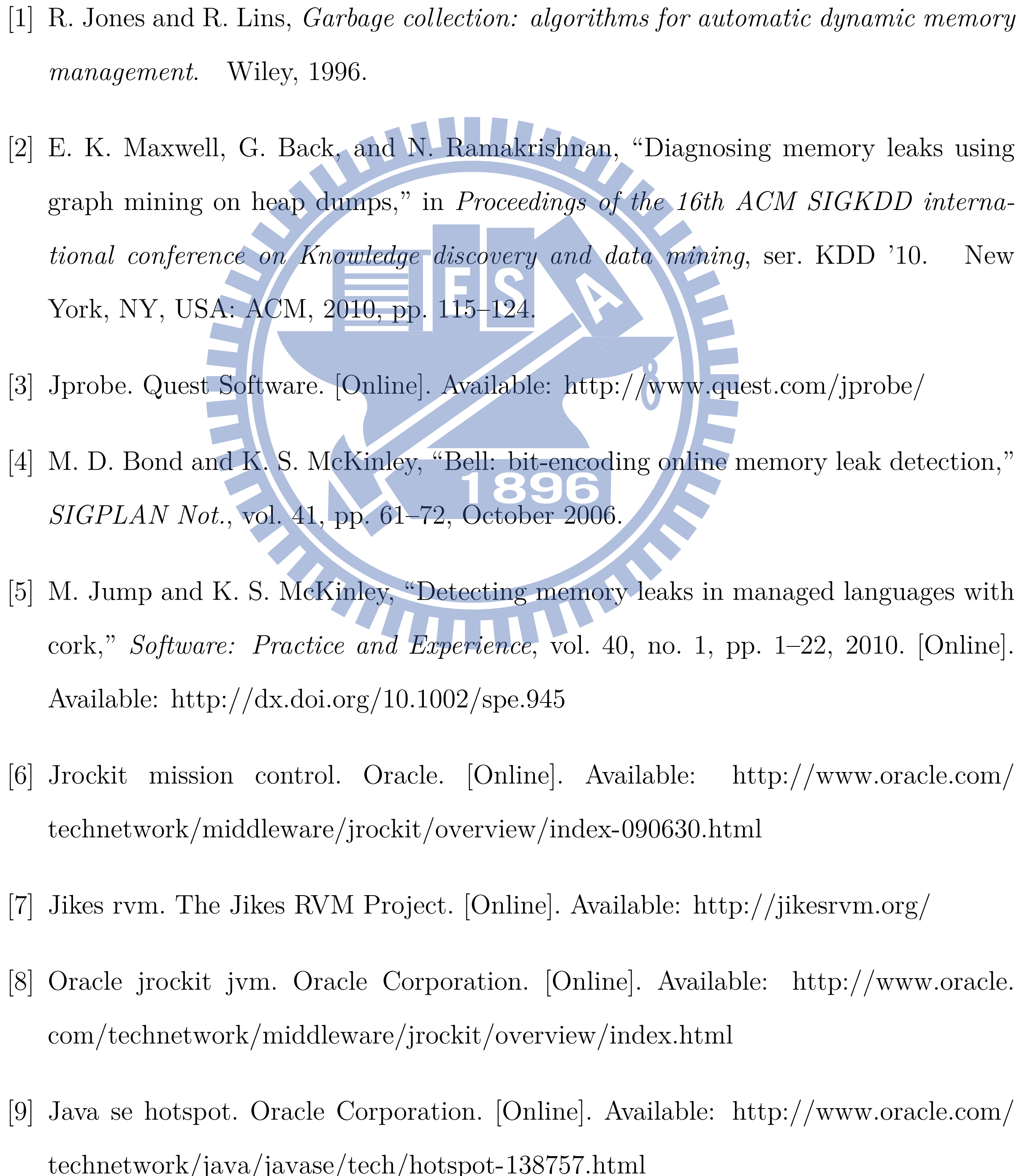
[5] M. Jump and K. S. McKinley, "Detecting memory leaks in managed languages with cork," *Software: Practice and Experience*, vol. 40, no. 1, pp. 1–22, 2010. [Online]. Available: http://dx.doi.org/10.1002/spe.945

[6] Jrockit mission control. Oracle. [Online]. Available: http://www.oracle.com/technetwork/middleware/jrockit/overview/index-090630.html

[7] Jikes rvm. The Jikes RVM Project. [Online]. Available: http://jikesrvm.org/

[8] Oracle jrockit jvm. Oracle Corporation. [Online]. Available: http://www.oracle.com/technetwork/middleware/jrockit/overview/index.html

[9] Java se hotspot. Oracle Corporation. [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/hotspot-138757.html



[10] Jvm tool interface version 1.2. Oracle. [Online]. Available: http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

[11] K. D. Cooper, T. J. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," 2001. [Online]. Available: http://www.hipersoft.rice.edu/grads/publications/dom14.pdf

[12] Eclipse bug 115789 - memory leak. The Eclipse Foundation. [Online]. Available: https://bugs.eclipse.org/bugs/show_bug.cgi?id=115789

Appendix A

The MyProgram Program

We design a sample Java program MyProgram to demonstrate the usage of the proposed agent. MyProgram consists of five files: MyProgram.java, test/MyDblLinkedList.java, test/MyListItem.java, test/MyDisplay.java and test/MySetting.java, whose source codes are listed below.

Source code of MyProgram.java:

```
1  import test.*;
2  import java.io.*;
3  import java.util.Date;
4  import java.util.TimeZone;
5  import java.text.DateFormat;
6  import java.text.SimpleDateFormat;
7
8  public class MyProgram {
9      private static void PrintMyDblLinkedList(MyDblLinkedList
           dbllinkedlist) {
10         System.out.print("MyListItems: (from head to tail)\n");
11         MyListItem current = dbllinkedlist.getHead();
12
13         while (current != null) {
```

```
14             System.out.print("   " + current.getValue() + "\t");
15             MyListItem prev = current.getPrev();
16 
17             if (prev != null) System.out.print("    Prev: " + prev.getValue() + "\t");
18             else System.out.print("    Prev: null\t\t");
19 
20             MyListItem next = current.getNext();
21             if (next != null) System.out.print("    Next: " + next.getValue() + "\n");
22             else System.out.print("    Next: null\n");
23 
24             current = next;
25         }
26         System.out.print("dbllinkedlist contains " + dbllinkedlist.getCount() + " MyListItems now. \n");
27     }
28 
29     private static void DoLoopContent(MySetting setting, MyDblLinkedList dbllinkedlist) {
30         if (dbllinkedlist.getCount() > 0) {
31             dbllinkedlist.removeHead();
32 
33             MyListItem currentHead = dbllinkedlist.getHead();
34             currentHead.setValue(currentHead.getValue() + "MOD");
35         }
36 
37         for (int i = 0; i < 3; i++) {
38             setting.addSerialNo();
39             dbllinkedlist.add("Item " + setting.getSerialNo());
40         }
```

```
41      }
42
43      public static void main(String[] args) {
44          MySetting setting = new MySetting();
45          System.out.print("Generated MySetting object instance:
                setting\n");
46
47          MyDisplay display = new MyDisplay(setting);
48          System.out.print("Generated MyDisplay object instance:
                display\n");
49
50          MyDblLinkedList dbllinkedlist = new MyDblLinkedList(setting);
51          System.out.print("Generated MyDblLinkedList object instance:
                 dbllinkedlist\n");
52
53          BufferedReader in =
54              new BufferedReader(new InputStreamReader(System.in));
55          System.out.print("\n** Press Enter key to continue experiment
                , Ctrl+Z to stop.\n");
56          try { in.readLine(); }
57          catch (Exception e) {
58              System.out.println("Caught an exception!");
59          }
60
61          // for experiment
62          int experiment_type = Integer.parseInt(args[0]);
63
64          if (experiment_type == 1) { // with agent v.s. without agent
65              long elapsed = 0;
66              for (int j = 0; j < args.length - 1; j++) {
67                  // loop count
```

```
68                  int loop_count = Integer.parseInt(args[j + 1]);
69                  if (j > 0)
70                      loop_count -= Integer.parseInt(args[j]);
71
72                  // record start time for experiment
73                  long startTime = System.nanoTime();
74
75                  // call DoLoopContent() for loop_count times
76                  for (int i = 0; i < loop_count; i++) {
77                      DoLoopContent(setting, dbllinkedlist);
78                      //System.out.println(i);
79                  }
80
81                  // calculate current elapsed time
82                  long currentTime = System.nanoTime();
83                  elapsed += currentTime - startTime;
84
85                  // output experiment 1 result to file
86                  try {
87                      if (j > 0)
88                          loop_count += Integer.parseInt(args[j]);
89                      FileWriter fstream = new FileWriter("exp1_data.
                            txt", true); // append
90                      BufferedWriter out = new BufferedWriter(fstream);
91                      out.write(loop_count + "\t" + dbllinkedlist.
                            getCount() + "\t" + elapsed + "\n");
92                      out.close();
93                  } catch (Exception e) {
94                      System.out.println(e);
95                  }
96              }
```

```
97          }
98          else if (experiment_type == 2) { // monitor all v.s. monitor
               partial
99              // experiment duration (note that input is min and is
                   converted to msec here)
100             double experiment_duration_msec = Double.parseDouble(args
                   [1]) * 60 * 1000;
101
102             while(true) {
103                 // record start time for experiment
104                 long startTime = System.currentTimeMillis();
105
106                 while(true) {
107                     DoLoopContent(setting, dbllinkedlist);
108
109                     // calculate current elapsed time
110                     long currentTime = System.currentTimeMillis();
111                     SimpleDateFormat dateFormat =
112                         new SimpleDateFormat("HH:mm:ss");
113
114                     dateFormat.setTimeZone(TimeZone.getTimeZone("GMT"
                           ));
115                     long elapsed = currentTime - startTime;
116
117                     // output message for experiment reference
118                     System.out.println("[ Elapsed Time: " +
                           dateFormat.format(new Date(elapsed)) + " ( " +
                            elapsed + " ms) ]");
119
120                     if (elapsed > experiment_duration_msec) break;
121                     else {
```

```
122                         //thread to sleep for the specified number of
                                milliseconds
123                         try {
124                             Thread.sleep(60000);
125                         } catch (Exception e) {
126                             System.out.println(e);
127                         }
128                     }
129                 }
130 
131                 BufferedReader in2 =
132                     new BufferedReader(new InputStreamReader(System.
                            in));
133                 System.out.print("\n** Press Enter key to continue
                        experiment, Ctrl+Z to stop.\n");
134                 try { in2.readLine(); }
135                 catch(Exception e) {
136                     System.out.println("Caught an exception!");
137                 }
138             }
139 
140         } else { // step by step ver
141             while(true) {
142                 DoLoopContent(setting, dbllinkedlist);
143                 BufferedReader in2 =
144                     new BufferedReader(new InputStreamReader(System.
                            in));
145                 System.out.print("\n** Press Enter key to continue
                        experiment, Ctrl+Z to stop.\n");
146                 try { in2.readLine(); }
147                 catch(Exception e) {
```

```
148                     System.out.println("Caught an exception!");
149                 }
150             }
151         }
152     }
153 }
```

Source code of test/MyDblLinkedList.java:

```
1  package test;
2
3  public class MyDblLinkedList {
4      protected MyListItem head = null;
5      protected int count = 0;
6      private MySetting setting = null;
7
8      public MyDblLinkedList(MySetting set) {
9          // add reference to MySetting
10         setting = set;
11     }
12
13     public void add(String value) {
14         MyListItem new_item = new MyListItem(value);
15         if (head == null) {
16             head = new_item;
17             count = 1;
18         } else {
19             new_item.setNext(head);
20             head.setPrev(new_item);
21             this.setHead(new_item);
22             count += 1;
23         }
```

```
24      }
25
26      public void removeHead() {
27          if (count > 0) {
28              count -= 1;
29              MyListItem new_head = head.getNext();
30              new_head.setPrev(null);
31              this.setHead(new_head);
32          }
33      }
34
35      public void setHead(MyListItem item) {
36          head = item;
37      }
38
39      public MyListItem getHead() {
40          return head;
41      }
42
43      public int getCount() {
44          return count;
45      }
46  }
```

Source code of test/MyListItem.java:

```
1  package test;
2
3  public class MyListItem {
4      protected String value;
5      protected MyListItem next;
6      protected MyListItem prev;
```

```
 7
 8     public MyListItem(String val) {
 9         value = val;
10         next = null;
11         prev = null;
12     }
13
14     public void setNext(MyListItem item) {
15         next = item;
16     }
17
18     public MyListItem getNext() {
19         return next;
20     }
21
22     public void setPrev(MyListItem item) {
23         prev = item;
24     }
25
26     public MyListItem getPrev() {
27         return prev;
28     }
29
30     public void setValue(String val) {
31         value = val;
32     }
33
34     public String getValue() {
35         return value;
36     }
37 }
```

Source code of test/MyDisplay.java:

```
1  package test;
2
3  public class MyDisplay {
4      private MySetting setting = null;
5
6      public MyDisplay(MySetting set) {
7          // add reference to MySetting
8          setting = set;
9      }
10 }
```

Source code of test/MySetting.java:

```
1  package test;
2
3  public class MySetting {
4      private int _serial_no = 0;
5
6      public void addSerialNo() {
7          _serial_no++;
8      }
9
10     public int getSerialNo() {
11         return _serial_no;
12     }
13 }
```

Appendix B

The RuntimeExp Program

We design a tiny Java program RuntimeExp to evaluate the runtime overhead of the proposed agent. RuntimeExp consists of three files: RuntimeExp.java, RuntimeExpMonitored.java and RuntimeExpNotMonitored.java, whose source codes are listed below.

Source code of RuntimeExp.java:

```
1  import java.io.*;
2  import java.util.ArrayList;
3  import java.util.Date;
4  import java.util.TimeZone;
5  import java.text.DateFormat;
6  import java.text.SimpleDateFormat;
7
8  public class RuntimeExp {
9      public static ArrayList<RuntimeExpMonitored> monitored = new
           ArrayList<RuntimeExpMonitored>();
10     public static ArrayList<RuntimeExpNotMonitored> not_monitored =
           new ArrayList<RuntimeExpNotMonitored>();
11     public static void main(String[] args) {
12         try {
13             int percentage = Integer.parseInt(args[0]);
```

```
14          if (percentage < 0 || percentage > 100) {
15              System.out.println("Please pass in an integer between
                    0 and 100.");
16              return;
17          }
18          System.out.println("Percentage of monitored objects: " +
                percentage + "%");
19
20          int remaining = 100-percentage;
21
22          for (int i = 0; i < percentage; i++)
23              monitored.add(new RuntimeExpMonitored("A" + i));
24
25          for (int i = 0; i < remaining; i++)
26              not_monitored.add(new RuntimeExpNotMonitored("B" + i)
                    );
27
28          BufferedReader in =
29              new BufferedReader(new InputStreamReader(System.in));
30          System.out.println("\n** Press Ctrl + \\ first.");
31          System.out.println("** And then press Enter key to
                continue experiment or Ctrl+Z to stop.");
32          try { in.readLine(); }
33          catch(Exception e) {
34              System.out.println("Caught an exception!");
35          }
36
37          System.out.println("\n** This might take a while...");
38
39          // record start time for experiment
40          long startTime = System.currentTimeMillis();
```

```
41
42          // START LOOPING
43          for (int j = 0; j < 1000; j++) {
44              for (int i = 0; i < percentage; i++) {
45                  monitored.get(i).set("A" + i);
46                  monitored.get(i).get();
47              }
48
49              for (int i = 0; i < remaining; i++) {
50                  not_monitored.get(i).set("B" + i);
51                  not_monitored.get(i).get();
52              }
53          }
54          System.out.println("loop end");
55          // calculate current elapsed time
56          long currentTime = System.currentTimeMillis();
57          SimpleDateFormat dateFormat =
58              new SimpleDateFormat("HH:mm:ss");
59
60          dateFormat.setTimeZone(TimeZone.getTimeZone("GMT"));
61          long elapsed = currentTime - startTime;
62
63          // output message for experiment reference
64          System.out.println("\n[ Elapsed Time: " + dateFormat.
              format(new Date(elapsed)) + " ( " + elapsed + " ms) ]"
              );
65
66          System.out.println("\n** Press Ctrl + \\ again.");
67          System.out.println("**Press Enter key to exit.");
68          try { in.readLine(); }
69          catch(Exception e) {
```

```
70                      System.out.println("Caught an exception!");
71                  }
72
73              }
74              catch (NumberFormatException e) {
75                  System.out.println("ERROR: " + e.getMessage());
76                  System.out.println("Please enter an integer!");
77              }
78              catch (Exception e) {
79                  System.out.println(e.getMessage());
80              }
81          }
82  }
```

Source code of RuntimeExpMonitored.java:

```
1   public class RuntimeExpMonitored {
2       private String value;
3
4       public String get() {
5           return this.value;
6       }
7
8       public void set(String str) {
9           this.value = str;
10      }
11
12      RuntimeExpMonitored(String str) {
13          this.set(str);
14      }
15  }
```

Source code of RuntimeExpNotMonitored.java:

```java
public class RuntimeExpNotMonitored {
    private String value;

    public String get() {
        return this.value;
    }

    public void set(String str) {
        this.value = str;
    }

    RuntimeExpNotMonitored(String str) {
        this.set(str);
    }
}
```