Cold Objects in the Java Virtual Machine

by

Baoguo Zhou

Bachelor of Computer Science, Hubei University of Technology, 1993

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

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s): Gerhard W. Dueck, PhD, Computer Science
Examiuing Board: Kenneth B. Kent, PhD, Computer Science
David Bremner, PhD, Computer Science
Richard J. Tervo, PhD, Electrical and Computer Engineering

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

December, 2015

©Baoguo Zhou, 2016
Abstract

Objects in the heap are alive when they are reachable from the root set, otherwise they are dead. Live objects that are not accessed for a specified time are called cold objects. If cold objects are moved to cold regions, then these regions need not to be included in garbage collection. Consequently, the pause time of garbage collection could be reduced and Java application throughput could be increased.

In this thesis, the proportion of cold objects in real Java applications has been investigated. The results show that some Java applications have as much as 22% cold objects. If these cold objects need not to be marked and swept during garbage collection, significant pause time could be saved. It has been suggested that all active objects can be identified by periodically walking the stack. This method is called the Stack-based solution. The correctness and efficiency of the Stack-based solution has been evaluated and confirmed with an Access Barrier methodology. The experimental results show that the Stack-based solution is acceptable to identify cold objects. Furthermore, some improved approaches to minimize the overhead have been implemented.
for the Stack-based solution. The improved approaches have shown good results. Experimental results show that the Java application throughput has been increased and the pause time of the garbage collection has been reduced under the improved approaches.
Dedication

To my parents, my wife Susan Zhang, my son Jim Zhou and my daughter Joy Zhou.
Acknowledgements

It is a pleasure to thank those who made this thesis possible. I would like to thank my supervisor Prof. Gerhard W. Dueck from the University of New Brunswick. My thanks also go out to Kim Briggs from IBM Canada for his technical guidance and help. Thanks to all my colleagues at IBM CAS-Atlantic who have been helpful to my thesis. I would also like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program.

Barry Zhou
Table of Contents

Abstract ii

Dedication iv

Acknowledgments v

Table of Contents xii

List of Tables xiv

List of Figures xvi

Abbreviations xvii

1 Introduction 1

2 Background 4

2.1 Java ........................................ 4

2.2 Java Virtual Machine ..................... 4

2.3 Memory Layout of the JVM ............... 6

2.3.1 Heap .................................. 7
## 3 Cold Object Preliminaries

3.1 Balanced GC ............................................. 14
   3.1.1 The Goal of Balanced GC .......................... 14
   3.1.2 Heap Organization in the Balanced GC .......... 15
   3.1.3 Garbage Collection in Balanced GC ............... 15

3.2 Objects .................................................. 16

3.3 Regions .................................................. 17
   3.3.1 Pinned Regions .................................... 18
   3.3.2 Cold Regions ..................................... 18

3.4 Threads ................................................ 18

3.5 Approaches to Collect Cold Objects ................. 20
   3.5.1 Measurement of “Coldness” ........................ 20
      3.5.1.1 Barrier-based Approach ...................... 20
      3.5.1.2 Stack-based Approach ....................... 20
   3.5.2 Collection of Cold Objects ........................ 21
3.6 Why Chose the Balanced GC Policy? ............. 21

4 Stack-based Solution .................... 23
  4.1 Stack-based Marking .................. 23
  4.2 Sampling Stack Frames ............... 24
  4.3 Pinning the Regions .................. 25
  4.4 Active-bit-map in Stack Frames ....... 26
  4.5 Region States ....................... 27
  4.6 Threads ................................ 28
    4.6.1 Mutator Threads ................. 28
    4.6.2 Daemon Thread .................... 29
    4.6.3 GC Threads ....................... 30
    4.6.4 Thread Schedule .................. 31
  4.7 Algorithms .......................... 32
    4.7.1 Selection Algorithm ............. 32
    4.7.2 Deselection Algorithm .......... 35
  4.8 Measurement in the Algorithm ........... 37
    4.8.1 Density ......................... 37
    4.8.2 Total Accessed Reference Count for Region .. 37
    4.8.3 New Active Reference Count for Pinned Region .. 38

5 Barrier-based Solution .................. 39
  5.1 Introduction of the Access Barrier .......... 39
    5.1.1 Objects and Fields in the Heap .......... 39
7 Evaluation of the Stack-based Approach

7.1 Methodologies ........................................ 62
7.2 Extension of the Running Duration Under JIT-off Mode .... 64
7.3 Matching of the Sampling Interval Under JIT-off Mode .... 65
7.4 Terms .................................................. 66
7.5 Experimental Environment ............................ 69
7.6 Experiments .......................................... 69
   7.6.1 Determination of Running Duration and Sampling In-
       terval in Derby ........................................ 69
   7.6.2 Determination of Running Duration and Sampling In-
       terval in Compiler.compiler .......................... 71
   7.6.3 Number of Cold Objects .......................... 72
   7.6.4 Reliability Analysis of Stack-based Solution .......... 74
   7.6.5 Convergence Time Analysis ........................ 75
7.7 Summary ............................................. 77

8 Performance Optimization — Activity Bit Embedding 80

8.1 Original Approach to Active-bit-map ........................ 80
8.2 An Approach to Embedded Activity Bit ....................... 82
8.3 Experiment ............................................. 84
List of Tables

5.1 Field Types in Java ........................................... 40
5.2 Read Barrier Execution Loops ........................... 45
5.3 Write Barrier Execution Loops ........................... 46
5.4 Access Barrier Execution Times in Derby ................ 47

6.1 Density of Cold Objects ................................. 55
6.2 Cold Regions ............................................. 56
6.3 Reactivation of Cold Objects .......................... 59
6.4 Garbage Rate of Cold Objects ....................... 60

7.1 Performance Under JIT-on and JIT-off Mode in Derby ... 70
7.2 Performance Under JIT-off Mode in Compiler.compiler ... 71
7.3 Number of Cold Objects .............................. 73
7.4 FalseInactivity .......................................... 74
7.5 FalseActivity ............................................ 74
7.6 HitRate ................................................ 74
7.7 Additional 4 Representative Benchmarks in SPECjvm2008 Suite 78
8.1 Comparison Between Active-bit-map Approach and Embedded-
activity-bit Approach .................................. 85
8.2 Time Cost on Accessing Each Reference .................. 87

9.1 Comparison Between SequentialRA and ConcurrentRA . . . 94
9.2 Increasing the Workload in the GC Main Thread .......... 95
9.3 Differences with 30x and 60x Workload in the GC Thread ... 96
9.4 Increasing the Workload in the Daemon Thread ........... 97
9.5 Differences with 30x and 60x Workload in the Daemon Thread 98

List of Figures

2.1 Runtime Memory Layout ..................................... 6
2.2 Java Stacks ..................................................... 8
2.3 Stack Frame ..................................................... 9

3.2 Region in Balanced GC [5] .................................... 16
3.3 Region Age [12] ................................................ 18
3.4 Region Classification: Red represents a pinned region, Light
Gray represents the tenured region, Dark Gray represents the
cold region. ...................................................... 19
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP</td>
<td>Base Pointer</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collection</td>
</tr>
<tr>
<td>GCMV</td>
<td>Garbage Collection and Memory Visualizer</td>
</tr>
<tr>
<td>GenCon</td>
<td>Generational Concurrent</td>
</tr>
<tr>
<td>GGC</td>
<td>Global Garbage Collection</td>
</tr>
<tr>
<td>GMP</td>
<td>Global Marking Phase</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>JIT</td>
<td>Just In Time</td>
</tr>
<tr>
<td>JRE</td>
<td>Java Runtime Environment</td>
</tr>
<tr>
<td>LIFO</td>
<td>Last In First Out</td>
</tr>
<tr>
<td>Optthrput</td>
<td>Optimal throughput</td>
</tr>
<tr>
<td>Optavgpause</td>
<td>Optimal average pause</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PGC</td>
<td>Partial Garbage Collection</td>
</tr>
<tr>
<td>SP</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>SPEC</td>
<td>Standard Performance Evaluation Corportion</td>
</tr>
<tr>
<td>SSD</td>
<td>Solid State Drive</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Java [24] is a platform independent programming language, and it is widely used in the information technology industry. Some Java applications tend to store a large number of objects on the heap for caching in order to obtain better performance. For example, Java applications with a database in memory or with frequently accessed files. Those objects are persistent in the heap, but they are seldom or never accessed. Objects that have not been accessed for a long time are considered to be cold.

The presence of cold objects in the heap is problematic because cold objects may be co-located in virtual memory with more active objects. Any page of virtual memory that contains cold objects may also contain active objects, and application references to the active objects will prevent the page from being swapped out of virtual memory.

A second fact is that the garbage collector [27] has to walk all objects regard-
less if they are cold or active. An idea is that if cold objects are collected and are moved to cold regions that are excluded by garbage collection, then garbage collection will be more efficient and pause times caused by garbage collections can be shortened.

Once cold objects are moved to cold regions, cold objects in the cold regions will be excluded from the garbage collection. Therefore, significant pause times of the garbage collection can be reduced. Furthermore, if cold regions are physically allocated in an external memory device, for example, a Solid State Drive (SSD) [13], internal physical memory can be saved.

Before cold objects are moved to cold regions, cold objects tend to be co-located with active objects. When active objects are accessed, both active objects and cold objects might be loaded into the cache at the same time. Therefore, cold objects will degrade the cache-hit [2] performance. If cold objects are collected and moved to cold regions, the cache-hit performance of Java applications could increase.

This thesis is concerned with how to identify and mark cold objects. A previous approach—the Stack-based solution—is described, and a new Barrier-based solution is proposed and validated. Furthermore, the efficiency of the Stack-based solution is evaluated. This thesis is structured as follows.

Chapter 2 and Chapter 3 introduce the reader to the necessary background and cold object preliminaries, which include the Java Virtual Machine [27], the Just-in-Time compiler [27], the Java stack [27], garbage collection, IBM J9 garbage collection policies [7], and a definition of cold objects.
In Chapter 4, the Stack-based solution to identify cold objects is reviewed, which explains how the Stack-based solution works. A Barrier-based solution to identify cold objects is presented in Chapter 5. Chapter 5 describes the Access Barrier and validates the Barrier-based solution. In Chapter 6, an investigation of the proportion of cold objects in real life applications is shown.

The Stack-based approach walks stack frames periodically. Since the sampling is not continuous, this mechanism will likely miss some information. One question that arises is how good the Stack-based approach is. In Chapter 7, the Stack-based approach is evaluated with the Access Barrier, which is an exact method.

In Chapter 8 and Chapter 9, two attempts to optimize the performance for the Stack-based approach are presented. The goal is to minimize the pause time of garbage collection and maximize the throughput of Java applications. Chapter 10 briefly discusses the collection of cold objects. Chapter 11 provides a short summary of the complete thesis as well as directions for future work.
Chapter 2

Background

This chapter will introduce the fundamental concepts of Java [24], the Java Virtual Machine [27], garbage collection (GC) [27, 25], and benchmarks [14, 15].

2.1 Java

Java is a high level computer programming language. Both object-oriented and platform independence are significant properties of Java. Java is intended to let developers “write once, run anywhere” [17].

2.2 Java Virtual Machine

The Java Virtual Machine [27] (called JVM) is a virtual computing machine that will run pre-compiled Java code. The Java Virtual Machine specifica-
tion [27] defines the behaviors of the JVM. The JVM acts as an interpreter of bytecode and handles interactions with the operating system. At run time, the JVM loads the class files and executes pre-compiled Java bytecode. The class loader, the garbage collection, the bytecode interpreter, and the Just-in-Time compiler are important components in the JVM. J9 [10] is a Java Virtual Machine developed by IBM. In this thesis, all experiments were run in IBM’s J9 Virtual Machine.

In Java, Java source files (.java files) are compiled into bytecode [27, 28] files (.class files). Bytecode is based on the instruction set of a virtual machine — the JVM. When bytecode is executed, the bytecode interpreter translates bytecodes into native executable machine instructions.

There are two ways to execute Java bytecode. One way is to use a bytecode interpreter, and another way is to use the Just-In-Time compiler (JIT) [27, 18]. The JVM maintains a call counter for each method. Whenever a method is invoked, the call counter will be incremented. When the call counter of a method reaches pre-defined threshold, the JIT compiles bytecode into native code and the native code will be executed. JIT is an important technology to improve the performance of Java applications. When running a JVM, the user may enable or disable the JIT. By default, the JVM enables the JIT. When needed, the mode of the bytecode interpreter can be chosen with the command line option -Xint, which turns off the JIT.
2.3 Memory Layout of the JVM

In the JVM, there are two categories of runtime data. One is JVM-level data, they are created on JVM start-up and are destroyed when the JVM exits. It includes the class/method storage area and the heap. Another is thread-level runtime data, they are created when a thread is created, and they are destroyed when the thread exits [27]. The runtime memory layout consists of components [27] in Figure 2.1:

![Runtime memory layout](image)

Figure 2.1: Runtime Memory Layout

1. The class/method storage area, which is used to store class methods, constants, and class attributes.

2. The heap, which is used to store objects that are created at runtime.

3. Java stacks, which are used to store local variables, passing arguments by caller, etc.

4. PC register, which is used to keep track of the currently executed instruction.

5. Native method stacks, which are used to support the native method.
The class/method area [27] is a shared resource among all threads in the JVM. The class/method area stores constants, class structures, method codes, and runtime method fields. In the multi-threaded environment, threads are run in parallel or concurrently. Each Java thread in the JVM has a Program Counter (PC) [27] register to record its own current execution instruction address.

### 2.3.1 Heap

The heap [27] is a memory space to store objects in the Java application, and it is a shared resource for all threads in the JVM. When an object is created, the required memory is allocated on the heap and the object is stored there. When an object is dead, it is automatically reclaimed by the GC. The heap is created on JVM start-up with a fixed-size or variable-size memory. The heap size can be configured with the following VM options in the IBM J9:

- `-Xmx <size>` - to set the maximum Java heap size
- `-Xms <size>` - to set the initial Java heap size

### 2.3.2 Java Stack

In Java, each thread has its own stack [27]. A Java stack is created whenever a new thread is created. A stack is divided into multiple stack frames. A new stack frame is generated on the stack whenever a method is called, and
it is pushed to the top of the stack. When a method returns, its stack frame will be removed.

Figure 2.2: Java Stacks

2.3.3 Stack Frames

Stack frames [27] are allocated on the JVM stack of the thread. Each frame usually contains:

1. Local variables and parameters,

2. Return value,

3. Operand stack, and

4. Reference to the runtime constant pool for the class of the current method.
Each frame has a base pointer (called \( BP \)) [27] and a stack pointer (called \( SP \)) [27], and the length of frame depends on the number of variables and parameters of the current method. The length of frame is variable, and SP is dynamic.

![Figure 2.3: Stack Frame](image)

The top of the stack is the stack frame for the currently active method. When a method calls another method, the invoked method will become the current method, its stack frame will become the current frame.

### 2.4 Garbage Collection

Java uses an automatic garbage collector [27] to manage memory in the object lifecycle. As such, the programmer does not look after objects’ deallocation.
When objects are dead, the GC will reclaim dead objects automatically. In commercial JVM products, GC algorithms are diverse. The IBM J9 virtual machine edition 7.0 provides 4 GC algorithms [8]. The -Xgcpolicy options control the behaviors of the GC. Different GC policies offer trade-offs between the throughput of the application and the pause times that are caused by the GC.

1. Optthruput [8]

The Optthruput policy aims to maximize Java application throughput. The Optthruput policy uses a non-segmented heap. When there is not enough free memory to allocate a new object, the Optthruput GC is triggered. Therefore, this policy can lead to high application throughput, but it also leads to long pause times during global garbage collection for a large-heap Java application.

2. Optavgpause [8]

The Optavgpause policy aims to minimize pause times. The Optavgpause policy sacrifices some throughput to shorten individual pause times. The Optavgpause policy introduces concurrent mark-sweep and the GC can perform marking tasks concurrently with Java applications. That is, when an application is executing, the GC can be executed in parallel. Therefore, pause time in the Optavgpause policy is shorter than in the Optthruput, but application throughput is reduced because some GC tasks are executed in parallel with Java applications.
3. Gencon [8]

The Gencon policy aims to balance between Java application throughput and pause time. The Gencon policy applies a traditional generational garbage collection algorithm [22] that uses a segmented heap, and introduces a concurrent mark algorithm. The heap is divided into a nursery area and a tenured area [6]. A copying algorithm [23] is used to move objects from a nursery area to a tenured area. The Gencon policy is particularly suitable for applications that have many short-lived objects.

4. Balanced [8]

The Balanced policy aims to avoid spikes in pause times, and it contains mark, sweep, compact and copying operations. The Balanced policy uses a region-based heap, which tries to avoid global collections on a whole heap to reduce long pause times.

2.5 Java Benchmarks

A Java benchmark is a program that is used to measure computing performance. The benchmark application generates test loads and calculates performance metrics. This thesis will leverage the existing Java benchmarks, such as SPECjvm2008 [15] and SPECjbb2005 [14], to test the JVM performance. Both SPECjvm2008 and SPECjbb2005 are commercial Java benchmarks. They reflect real life Java applications and are widely applied in the
Java world.

2.5.1 SPECjvm2008

“SPECjvm2008 (Java Virtual Machine Benchmark) is a benchmark suite for measuring the performance of a Java Runtime Environment (JRE)” [15]. SPECjvm2008 contains sufficient and real life Java applications, that can be used to test the JVM performance. SPECjvm2008 suite consists of 10 categories of packages, and they are divided into 38 sub-benchmarks, with each sub-benchmark representing a kind of real life Java application [29].

1. Compiler: Compiling Java source files with the OpenJDK compiler

2. Compress: Compressing data using Compression algorithm

3. Crypto: Encrypting and decrypting

4. Derby: Applying database technology

5. MPEGaudio: Decoding audio

6. Scimark: Making numerical computations

7. Serial: Benchmarking serialization and de-serialization of objects

8. Startup: Starting for each benchmark in SPECjvm and executing a single iteration of the benchmark

9. Sunflow: Processing an image in a multi-threaded environment
2.5.2 SPECjbb2005

“SPECjbb2005 (Java Server Benchmark) is SPEC’s benchmark for evaluating the performance of server-side Java” [14]. The benchmark can be used to validate and test performance of various aspects of the JVM, JIT compiler, garbage collection, and threads.

2.5.3 Considerations in Benchmarks

In this thesis, benchmarks are applied in a wide variety of experiments. Benchmarks are important to measure how many cold objects exist and how much GC time can be saved by dealing with them separately. The following characteristics of benchmarks are exploited:

- Long running time: Compared with the DaCapo [21] benchmark, both SPECjvm2008 and SPECjbb2005 are macro benchmarks. They can execute not only for a long time, but also can execute for many iterations. They satisfy the long running time requirements.

- Performance output: By running the benchmarks on the original JVM, the performance can be measured. The JVM will be modified to deal with cold objects. The performance of the modified JVM will be compared to the original. It is expected that not all benchmarks will see the same improvement.
Chapter 3

Cold Object Preliminaries

This chapter introduces the basic concepts related to cold objects. It details Balanced garbage collection [1] (Balanced GC), cold objects, and cold regions. Furthermore, this chapter discusses why the topic of cold objects will be studied and why the Balanced GC is chosen as the GC policy for this study.

3.1 Balanced GC

3.1.1 The Goal of Balanced GC

In Java applications with a large heap, the Gencon GC will often result in long pauses. This is due to the fact that the collection of tenured objects takes significant time. With increasing heap sizes, the pause time will be increased under the Gencon policy. On average, the pause time is acceptable
for typical applications, but occasionally the pause time is significant. The Balanced GC aims to avoid long pause times. The aim is that even in a large heap, the pause time will be within a band [8], as shown as in Figure 3.1.

![Pause Time Goal of Balanced GC](image)

**Figure 3.1: Pause Time Goal of Balanced GC [5]**

### 3.1.2 Heap Organization in the Balanced GC

Balanced GC has a segmented heap. The heap is divided into multiple equal-sized regions. The size of the regions is decided at the JVM start. The layout is shown in Figure 3.2. Initially, a region is free. Once a free region is used to store objects, it becomes an Eden region. New objects will be allocated in an Eden region. The Balanced collector chooses Eden regions and some other regions with good prospects of containing dead objects for its collection.

### 3.1.3 Garbage Collection in Balanced GC

There are three different types of sub-operations in Balanced GC [1].

1. Partial garbage collection (PGC), collects garbage objects only in a subset of regions, not in all regions. Eden regions are always included
in the collection set. Since a low proportion of regions are collected, pause time is not too long.

2. *Global mark phase* (GMP), traces all live objects in the heap, and marks them as active when they are reachable from the root set. GMP is an incremental operation executed during both the GC phase and the mutator thread execution phase.

3. *Global garbage collection* (GGC), marks and sweeps the whole heap. Under normal conditions the GGC should not be preformed. It is used as a last resort, when memory becomes very tight. It will produce a long pause, and thus should be avoided.

In practice, the parameters should be set in such a way that the PGC will do the required work.

### 3.2 Objects

An *object* is a class instance or an array [24]. Objects are created with a `new()` operator by a programmer, and objects are allocated in the heap.
The heap memory is occupied by live and dead objects. *Live objects* are those that are reachable from the root set [26, 20], and will not be collected by the GC. *Dead objects* are those that are not reachable from the root set [20], but have not yet been collected by the GC. Dead objects occupy heap memory space until they are eventually collected by the GC. *Cold objects* are those that are alive, but are seldom accessed, and the elapsed time since the most recent access exceeds a given threshold.

### 3.3 Regions

In the Balanced GC, the whole heap is divided into equal-sized *regions*. The GC is usually not performed on the whole heap except GGC, and several regions that have to reclaim objects form a set, the PGC will be performed on this set [1].

- **Region Age**

  Balanced GC [1] calculates the age of objects for each region with 25 possible cycles. An age 0 region—Eden space—contains the objects that have been most recently allocated, which is shown in Figure 3.3.

- **Young Regions and Tenured Regions**

  In Balanced GC [1], *tenured regions* [31] are those whose age is 24. In contrast, if a regions’ age is less than 24, they are called *young regions*. 

17
3.3.1 Pinned Regions

*Pinned regions* [3] are those that are tenured regions and excluded from PGC. The purpose of pinned region is to pin objects in the current region and avoid objects moving to other regions via copy-forward [30] and compaction [30] during the GC phase. Since pinned regions are excluded from PGC, there are no copy-forward and compaction GCs on the pinned regions.

3.3.2 Cold Regions

*Cold regions* are those that are reserved to contain only cold objects. Cold regions are a subset of regions in the heap. They are used to store cold objects. A customized GC policy can be performed on the cold regions.

3.4 Threads

Both the JVM and Java applications usually run under the operating system as one process with multiple threads. In this process both JVM threads and application threads run in parallel on multiple CPUs or concurrently on the
single CPU. For convenience sake, threads in the JVM are categorized as mutator threads and system threads.

1. **Mutator threads**: They are Java programs’ threads, and execute the user specific tasks. In a concurrent Java application, usually there are multiple mutator threads and they are executed concurrently or in parallel. For each mutator thread, the JVM will create a thread context.

2. **System threads**: They are the JVM’s threads and serve as the JVM. This thesis is concerned with garbage collection threads. From the view of implementation, they usually contain multiple threads.
3.5 Approaches to Collect Cold Objects

In order to collect cold objects, there are two significant steps—identifying cold objects and harvesting cold objects.

3.5.1 Measurement of “Coldness”

The way of identifying cold objects is based on the activity of the object. If the object is inactive for a specified period, it is a cold object, otherwise not. The problem is how to capture the activity of objects. Two approaches to object activity tracking are described in Chapter 4 and Chapter 5.

3.5.1.1 Barrier-based Approach

An Access Barrier is a mechanism provided by the JVM. Whenever access is given to an object, for example, reading a field in an object and/or writing a field in an object, a read/write event will be triggered. An Access Barrier is a good way to capture and record the activity of objects. A subsequent chapter will introduce this approach in detail.

3.5.1.2 Stack-based Approach

Another potential way to identify cold objects is to leverage stack frames. Each Java thread has a private stack. A Java stack stores frames, contains local object references, passing arguments and partial results. When a method is called, the JVM will generate a new frame and push it on the stack. When
the method returns, the stack frame will be popped.

The approach is to sample the stack frame to identify active objects. When an object reference is observed in the stack frame, it shows this object is active at that time [19]. Reference information can be captured in the stack frames. The object activity can be recorded when the references are walked in the stack frame. A subsequent chapter will introduce this approach in detail as well.

### 3.5.2 Collection of Cold Objects

Since cold objects in cold regions are alive, but inactive, cold regions can be excluded from PGC collection sets. Some objects that satisfy the criteria of cold objects within collectible pinned regions are moved to cold regions. In some scenarios, when cold objects co-locate with active objects in collectible pinned regions, and if there is a low proportion of active objects, it is more efficient to move the active objects instead the cold objects.

### 3.6 Why Chose the Balanced GC Policy?

In the Balanced GC [1] it is easy to separate some regions to be used as cold regions. Regions can be marked as cold regions when required. In contrast, both the OptThroughput and the OptAveragePauseTime have a non-segmented heap, so it is not easy to dynamically split specific area as cold regions. Although Gencon GC is a segmented heap, it is divided into
two zones as two generations.

Furthermore, when cold objects become active, putting reactivated cold-objects back to non-cold storage space is a new issue. Both the OptThroughput and the OptAveragePauseTime have no mechanism to copy-forward objects. There is a copy-forward mechanism in the Gencon policy and it can put re-activated cold objects back to the tenured generation, but the operation is costly.

However, the Balanced GC has a natural hierarchy for regions. Cold regions can be created by marking a region property as “Cold”. Secondly, if some cold objects are reactivated, it can be achieved by marking region’s properties as “Managed”. Consequently, the region will be put back to region pool. Therefore, the Balance GC can manage cold regions efficiently.
Chapter 4

Stack-based Solution

This chapter will describe a previously introduced approach [3] (called Stack-based solution) to walk stack frames and identify cold objects. The framework of this approach has been implemented and it is a good entry point and foundation to research cold objects. In the subsequent chapters, the Stack-based solution will be evaluated and analyzed. Furthermore, the subsequent chapters will present ways to improve the Stack-based solution.

4.1 Stack-based Marking

Cold objects are the objects that remain inactive for a specific duration. Since the objects are inactive, there have been no read/write accesses to the objects. It is difficult to determine the inactive objects because there are neither hints nor operations on cold objects. However, if we change our view, it
is possible to spot the active objects, because there are read/write accesses to them.

In the specific memory space where objects are stored, if those active objects are picked out, the remaining must be inactive objects. “An analogous example will be that there are both black and bright balls in a dark room. There is no way to pick out the black balls in the dark, but the bright balls can be picked out even without light, so those that remain must be black” [3].

The Stack-based approach identifies active objects through walking stacks. As mentioned earlier, each thread has its own stack. Whenever a method is invoked, a new stack frame will be generated. Local variables and passed arguments will be stored in the stack frames. Assuming there are no unused local variables and unused passed arguments, objects corresponding to local variables and passed arguments in the current stack frame are considered to be active. Once these active objects are marked as active, objects that remain unmarked for a specified amount of time are cold.

4.2 Sampling Stack Frames

The stack frames of application programs are located in the mutator threads. There is no interface to collect references in the stack frames that are defined in the mutator threads from the perspective of the Java virtual machine specification. A new mechanism has been designed and implemented in order to collect the reference information from the stack frames.
Whenever a mutator thread is created, the JVM creates a context for this mutator thread. The mutator thread has a hook function for the JVM to execute a specific job. The JVM can periodically notify the mutator threads to walk the stack frames. This procedure is called *sampling.*

### 4.3 Pinning the Regions

The objects in the evacuated regions are always moved to the survivor regions by the copy-forward algorithm in the Balanced GC. It is a crucial fact that the objects in the heap are always movable in the Balanced GC. Furthermore, whenever an object is moved, its reference, which is presented to Java programmers, is changed as well. But this process is transparent for the mutator thread and Java programmers.

In the Balanced GC, algorithms of both copy-forward and compacting move the objects. In order to enable detection of inactive objects, some regions are selected for pinning. *Pinned regions* are excluded from PGC collection sets, so the location of objects contained within pinned regions remain fixed. Not all regions need to be pinned. Otherwise, no GC will take pace. Only those regions that satisfy the pinned region selection criteria are pinned. The Selection algorithm will be described in the subsequent sections.
4.4 Active-bit-map in Stack Frames

When an object is identified as active, a flag will be set. In the Stack-based solution, an active-bit-map approach is used to record the flag.

The JVM uses a mark-bit-map to track the memory footprint for the Java heap. Every bit in the mark-bit-map represents an 8-byte-size memory space of the heap. For example, if an object has the size of 64 bytes, after the object is instantiated successfully, it occupies 64 bytes in the heap. Then, the 8-bits corresponding mark-bit-map will be set.

The active-bit-map has a similar structure to the mark-bit-map in Figure 4.1. It keeps track of the activity of the references in the pinned regions. When an object is identified as active, the corresponding bit in the active-bit-map is marked as “1”.

![Mark-bit-map and Active-bit-map](image)

Whenever a region is pinned, an active-bit-map is assigned to this pinned region, and initialized to “0”. Once a bit is set to “1”, it remains “1” until the region is unpinned or becomes a collectible pinned region. Whenever a region is unpinned, the active-bit-map is released.
4.5 Region States

Figure 4.2 shows region state transition.

Figure 4.2: Region State Transitions During Marking Cold Objects

1. Initially, a region’s state is Free.

2. When new objects are allocated in Free regions, then these regions become Managed. When a Managed region is evacuated by a copy-forward GC, it is returned to the Free state.

3. When a Managed region satisfies pinned region selection criteria, it will be pinned and become a pinned region. When a pinned region is deselected, then it will return to the Managed state.

4. When a Pinned region has collectible cold objects, it will transit to the Cold region.
5. When a *Cold* region is collected, active objects are moved to *Managed* regions, the text `cold` region will transit to the *Free* region.

Collectible cold objects are always sourced from pinned regions, and are harvested and copied to cold regions.

### 4.6 Threads

From the view of the JVM, there are two kinds of threads—system threads and mutator threads. The system threads are those that implement the JVM functionality. The mutator threads execute the user-specific functionality.

The JVM contains various system threads. This thesis is only concerned with the GC threads and the reference sampling daemon thread. The GC threads are responsible for the functionality of the GC. A number of GC threads can be deployed when the JVM starts up, and multiple GC threads can work in parallel. The daemon thread is designed to mark the active-bit-map and periodically notify the mutator thread to walk the stack. All involved threads in this thesis are shown in Figure 4.3.

#### 4.6.1 Mutator Threads

During runtime, many mutator threads are concurrently executing in the system. Mutator threads possess their own stacks and stack frames. The job of walking stack frames will be undertaken by the mutator threads. The daemon thread periodically sends the sampling signal to the mutator
threads. Once the mutator threads receive the sampling signal, the mutator threads will execute the walking operations at their next safe point. The mutator thread walks its own stack frames, picking out active references that belong to pinned regions, and storing active references in thread-local reference buffers. Each stack walk proceeds from the topmost frame down to the most recent frame that was present in the previous stack walk.

4.6.2 Daemon Thread

The daemon thread runs concurrently with the mutator threads and is suspended during GC cycles. The daemon accesses the reference buffers from mutator threads only after they have completed a stack walk, and these mutator threads will not modify their reference buffers until the daemon signals them to start a new stack walk. Therefore there is no contention for access to reference buffers between daemon and mutator threads. Only the daemon thread accesses pinned region activity maps between GC
cycles, and only the master GC thread accesses them during GC cycles. Therefore, there is no need to explicitly mediate access to pinned region activity maps at any time.

### 4.6.3 GC Threads

In the Balanced GC, the GC threads periodically perform global marking. When there is no memory to allocate for new objects, a GC—PGC or GGC—might be triggered to reclaim dead objects.

A management module, which is called the *pinned region manager*, processes selection and deselection of pinned regions. The pinned region manager is added to the GC thread. Therefore, the pinned manager will be executed once every GC cycle. The management module functionality of pinned regions includes:

1. selection of pinned regions,
2. deselection of pinned regions,
3. walking the pinned regions, and
4. starting, suspending, or resuming the daemon thread.

During every GC cycle, the pinned manager task will be executed once. Once entering the pinned manager task, it will disable the daemon sample thread. The sampling daemon thread is re-enabled at the end of every GC cycle.
4.6.4 Thread Schedule

The GC threads, the daemon thread and mutator threads will co-operate to accomplish the work of marking cold objects.

Figure 4.4: Thread Schedule At Runtime

In Figure 4.4, the GC threads (see the blue rectangle) are executed periodically. When the GC threads work, they stop the world. The daemon thread and all mutator threads are suspended. Between GC cycles the daemon thread and all mutator threads are concurrently executed. Each mutator thread will complete at most one stack walk event between successive daemon thread harvest cycles. That is, each mutator has a chance to walk its own stack to sample active references at every daemon cycle.

The run of the daemon thread is controlled by the pinned region manager described in Subsection 4.6.3. When the GC is triggered, the pinned region manager suspends the daemon thread, and the JVM suspends all mutator threads. When the GC is done, the JVM will resume all mutator threads.
and the pinned region manager will resume the daemon thread.

4.7 Algorithms

In Stack-based marking, there are two important algorithms - the Selection algorithm and the Deselection algorithm.

4.7.1 Selection Algorithm

Regions are pinned in the Selection procedure. If the individual object activity is expected to be captured in a specific region, this region should be pinned. Once a region is pinned, because the PGC does not take place at this region, objects in the pinned region will not be moved to other regions. There are three factors to be considered in the selection of pinned regions.

- Region age,

- Region density, and

- Region activity.

Region age is a logical age that increases with successive GC cycles, up to a maximum age of 24. During the object allocation, objects are allocated in young regions, such as Eden regions. The young regions have an unstable state; many objects are allocated in these regions, and at the same time many objects die soon [30] and are collected by the GC. It is not a good
idea to consider young regions for the collection of cold objects. In contrast, the tenured regions have a stable state. They experience many cycles of the PGC, and eventually there are not many short-lived objects. Therefore, only tenured regions are considered to be pinned.

Region density is the proportion of the region size occupied by objects to the whole region size. Since the goal of this thesis is to collect cold objects, it is natural that a higher density region has higher probability of collecting many cold objects. A tenured region that has only a few objects has a low density, it has a low probability of collecting many cold objects.

Region activity reflects how active the objects in the region are. Region activity increases by one with each access to an object. An assumption is that when all active objects are marked and there is no new active objects for a specified duration, the remaining objects are cold. The higher the activity is, the shorter the time of determining cold objects.

The pre-requisite of selection is

- the region is tenured, i.e., whose age is 24 (24 is the maximal region age), and
- the region density is greater than a preset threshold, the default is 60%.

When a region satisfies the above criteria, it will be selected as a candidate for pinned regions. All pinned region candidates are ranked according to the following formula:
The larger the DA is, the better the candidate’s quality. The Selection algorithm is given below.

```
DA = log(region density * region activity)

The Selection algorithm
{
    for each region in the heap
        if tenured region and density > pinned_density_threshold
            calculate DA
            push the region into a set of candidates
        end if
    end for
    Rank candidates’ DA in descending order
    Select candidates whose DA > DA_threshold
        Set "Pinned" flag in this candidate region
        Add this candidate to the set of pinned regions
}
```
4.7.2 Deselection Algorithm

When a region is pinned, every active object will be marked individually in the pinned region. For a pinned region, there are two choices over time.

1. Becoming a collectable cold object region: whenever there are no new active objects for a specified time, this region is considered to have collectable cold objects. Afterwards, collectable cold objects are collected from this region to a cold region. This region will be unpinned after cold objects are collected, and this region will return to its initial state—the Managed state.

2. Forced to terminate the Pinned state: if there are no collectable cold objects in a specific pinned region, it defeats the purpose of collecting cold objects. That is, it is not worth continuing to process cold objects in this pinned region. This pinned region will be removed from the set of pinned regions. It will return to its Managed state as well.

Consider the following scenario. Suppose each object in the pinned regions is high activity, with all objects in the pinned regions being set eventually as active over time. Since the operation of marking the active objects will persist, there will be no cold objects. That is, it is demanding work with diminishing return. So when the proportion of cold objects in a specific pinned region is lower than a preset threshold, it is not worth continuing to process cold objects in this pinned region. Another scenario is when density of a specific pinned region drops below
a threshold. For example, initially this pinned region contains many objects, most objects die over time, and few objects remain in the region. It is natural to remove this region from the set of pinned regions.

The Deselection algorithm
{
    for each pinned region
        if this pinned region has collectable cold objects
            Collect cold objects in this pinned region
            Clear "Pinned" flag for this region
            Set "Managed" flag for this region
        endif
        if region density < unpinned_density_threshold
            Clear "Pinned" flag for this region
            Set "Managed" flag for this region
        end if
        if proportion of collectable cold objects < threshold
            Clear "Pinned" flag for this region
            Set "Managed" flag for this region
        end if
    end for
}
4.8 Measurement in the Algorithm

In order to implement the Selection and Deselection algorithms, the following metrics are used:

1. Region density for every region,
2. Total accessed reference count for every region, and
3. New active reference count for each pinned region.

4.8.1 Density

The density is one of the selection factors for Managed regions. As long as the density of a region is greater than a density threshold, this region is eligible to be selected as a pinned region. Furthermore, the density is also one of the deselection factors. When the density of a pinned region is lower than a threshold for a pinned region, this region will be unpinned.

4.8.2 Total Accessed Reference Count for Region

The total accessed reference count reflects region activity. A moving average for the region activity count is maintained for every region. This is one of the Selection algorithm factors. Regions with high moving averages are assumed to be more active, and active-bit marking will converge more rapidly.
4.8.3 New Active Reference Count for Pinned Region

The new active reference count reflects the incremental activity. At the end of each GC cycle, it resets to zero. While walking stack frames, if a new active object is found, the new active reference count will increase by one.

The new active reference count is used to evaluate whether a pinned region has collectible cold objects. If the new active reference of a pinned region remains zero for a cold threshold, this pinned region can harvest cold objects.

Figure 4.5: New Active Reference - red points are marked active references, black points are un-accessed references
Chapter 5

Barrier-based Solution

When a read or write access to an object occurs, the JVM allows the Access Barrier to capture the access operation on the object. A hook can be attached to the Access Barrier and access operations can be recorded in the hook.

5.1 Introduction of the Access Barrier

5.1.1 Objects and Fields in the Heap

An object is a class instance or an array[27]. In J9, objects on the heap are categorized as *mixed objects* or *indexable objects*. A mixed object is a single object, and an indexable object is an array object. Each object on the heap consists of fields. The fields might be primitive types or reference types. The fields’ type are categorized in Table 5.1 [24].

When an object’s field is accessed, actually the object is accessed. At this
Table 5.1: Field Types in Java

<table>
<thead>
<tr>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. boolean</td>
</tr>
<tr>
<td>2. byte</td>
</tr>
<tr>
<td>3. char</td>
</tr>
<tr>
<td>4. long</td>
</tr>
<tr>
<td>5. short</td>
</tr>
<tr>
<td>6. int</td>
</tr>
<tr>
<td>7. float</td>
</tr>
<tr>
<td>8. double</td>
</tr>
<tr>
<td>9. reference</td>
</tr>
<tr>
<td>10. returnAddress</td>
</tr>
</tbody>
</table>

point, this object will be marked as active.

5.1.2 Mixed Objects Access

Mixed objects are not indexable on the heap. A mixed object is composed of a single or multiple fields. When a field in a mixed object is accessed by a read/write, an access point can be triggered and captured. So each field type corresponds a hook in the JVM, such as an U32 hook corresponds an entry point for a 32-bit unsigned field, an I32 corresponds an entry point for a 32-bit signed field, etc.

Figure 5.1: Mixed Objects Access
For example, when a 32-bit unsigned field is written, a writeI32() method of mixed objects is invoked. This method can be caught by the barrier. Similarly, a read operation can be captured by appropriate method.

### 5.2 Indexable Objects Access

The implementation of indexable objects’ Access Barrier has a similar implementation to mixed objects. But indexable objects have also some additional access methods that are not included in the mixed objects, such as 8-bit and 16-bit read or write, which are shown inside the red rectangle in Figure 5.2.

![Figure 5.2: Indexable Objects Access](image)
5.2.1 Object Allocation

Besides reading or writing fields within objects, object allocation, object clone, and JNI global object allocation are included in the Access Barrier. That is, when an object is allocated, the allocation operation can be captured as well.

![Object Allocation Diagram]

Figure 5.3: Miscellaneous Access

5.2.2 Access Operation

When a read or write access is triggered, an appropriate barrier method is executed. For the read or write barrier, an object reference will be passed by arguments, which can be captured and recorded. For the allocation barrier, an allocated object reference will be returned, which can be captured and recorded as well.

During reading or writing an object, an additional piece of source code can be hooked in the JVM to augment the original function of reading or writing to offer the desired functionality. A pseudocode example follows.
int TraceAccessBarrier::singleObjectReadU64(Thread *thread,
    Object *reference, int offset)
{
    int result = superClass::singleObjectReadU64(thread,
                 reference,offset);
    tagActiveObjectMap(thread,reference); /*!< this is code stub*/
    return result;
}

void TraceAccessBarrier::singleObjectStoreU64(Thread *thread,
    Object *reference, int offset, int value)
{
    superClass::singleObjectStoreU64(thread,
                  reference,offset,value);
    tagActiveObjectMap(thread,reference); /*!< this is code stub*/
}

5.3 Validating the Access Barrier

The Access Barrier provides a mechanism to capture the read/write access to the objects. But in some experiments, some read or write access to objects cannot be captured by the Access Barrier. The following section will explain
reasons and validate the Access Barrier.

5.3.1 The JIT

A Java program consists of classes, which contain platform-neutral bytecodes that can be interpreted by a JVM on many different computer architectures. The JIT compiler is used to improve Java execution efficiency instead of Java bytecode interpreter. The following sections will demonstrate the impact of the JIT on Access Barrier with several test cases.

5.3.2 Test Case 1: Single Read Instruction

The operation of a read U32 barrier is repeatedly performed 100,000 times with both the JIT-on mode and the JIT-off mode, respectively.

```
public class SingleOperation {
    public static void main(String[] args) {
        One anObject = new One();
        for(int i=0; i < 10000; i++) {
            anObject.get(i);
        }
    }
}

class One {
```
```java
private int count;
public void set(int in) {
    count = in;
}
public int get() {
    return count;
}
```}

Table 5.2: Read Barrier Execution Loops

<table>
<thead>
<tr>
<th>Measured objective</th>
<th>Description</th>
<th>Loops</th>
<th>read accesses under JIT=off</th>
<th>read accesses under JIT=on</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadU32 operations</td>
<td>Read a field of 32-bit in the object</td>
<td>100,000</td>
<td>100,000</td>
<td>51,547</td>
</tr>
</tbody>
</table>

The results in Table 5.2 show read barriers are performed 100,000 times with the JIT mode off. The read operations of the Access Barrier equal the times of reading the field in the program. That means the Access Barrier can capture all read operations when the JIT is off. But the read operations of the Access Barrier are dramatically lower when the JIT is enabled, which has only 51,547 read operations.

With the JIT turned on, every method is initially interpreted by the JVM and the Access Barrier can capture the read accesses. When the call count
of a method exceeds the JIT compilation threshold over time, this method will be compiled into native instructions, and the majority of Access Barrier operations will be taken out by the JIT compiler. The Access Barrier loses the read operations of optimized native instructions. Consequently, the Access Barrier can capture only partial read access operations, and not all read accesses.

5.3.3 Test Case 2: Single Write Instruction

The operations of a write U32 barrier are repeatedly performed 100,000 times with both the JIT on and the JIT off, respectively.

Table 5.3: Write Barrier Execution Loops

<table>
<thead>
<tr>
<th>Measured objective</th>
<th>Description</th>
<th>Loops</th>
<th>write accesses under JIT=off</th>
<th>write accesses under JIT=on</th>
</tr>
</thead>
<tbody>
<tr>
<td>WriteU32 operations</td>
<td>Write a field of 32-bit in an object</td>
<td>100,000</td>
<td>100,000</td>
<td>51,527</td>
</tr>
</tbody>
</table>

The results in Table 5.3 show that the write barrier is performed 100,000 times in the JIT-off mode, and the Access Barrier has captured 100,000 write accesses.

But the write operations of the Access Barrier are dramatically lower when the JIT is enabled, which have only 51,527 write operations. The reason is the same as explained in the preceding subsection. Because some methods are optimized and compiled into native instructions by the JIT compiler.
The majority of Access Barrier operations have been taken out by the JIT compiler. Consequently, the Access Barrier captured only partial write accesses.

5.3.4 Test case 3: running SPECjvm2008 Derby

The SPECjvm2008 benchmark Derby executes a great number of read/write access operations to objects. In this subsection Derby will be run for 5 minutes under both the JIT on and the JIT off, respectively. Both read accesses and write accesses are captured. The results are shown in Table 5.4.

<table>
<thead>
<tr>
<th>Measured objective</th>
<th>Description</th>
<th>Running time</th>
<th>Access times under JIT=off</th>
<th>Access times under JIT=on</th>
</tr>
</thead>
<tbody>
<tr>
<td>All read and write operations</td>
<td>SPECjvm2008 Derby java package</td>
<td>5 minutes</td>
<td>67,145,184,162</td>
<td>367,119,304</td>
</tr>
</tbody>
</table>

The number of access in the JIT-on mode has only 5.46% of the number in the JIT-off mode. Although the JVM executes more code with the JIT-on than JIT-off, obviously the Access Barrier loses the majority of read and write access operations under the JIT-on mode.
5.3.5 Summary

Whenever a read/write operation on an object occurs, it can be captured with an Access Barrier. The Access Barrier is triggered frequently, it is a costly operation. In J9, the JIT optimizes frequently used methods by translating them into native instructions. Only a few operations of the Access Barrier remain and are invoked with the JIT-on mode, such as, the operations of reading/writing a reference field remain. The majority of Access Barrier operations are taken away and are not executed in order to speed up the execution.

The above experimental results confirm that the Access Barrier is affected by the JIT. In real life applications, the JIT is enabled by default when Java programs run. That is, the Access Barrier does not work completely under a normal Java application running.

It is concluded that the Access Barrier is not a feasible approach to determine the activity of objects with the JIT-on mode. However, the Access Barrier works well under JIT-off mode.
Chapter 6

The Proportion of Cold Objects

The previous chapters have defined cold objects. One question that arises is how many objects are cold in real Java applications. If there are only a few cold objects in real Java programs, it is not worth researching this topic. This chapter will investigate the proportion of cold objects via the experimental results of benchmarks.

6.1 Introduction

In order to get information regarding the proportion of cold objects, there are two issues to be addressed. What are typical Java applications in the real world? That is, how can representative Java applications be identified? Secondly, how can we measure the proportion of cold objects?

Both SPECjvm2008 [15] and SPECjbb2005 [14] are software suites which
have been widely accepted by the information technology industry. They contain several real life Java applications. These benchmarks will be used to obtain insight into cold object behavior.

When these benchmarks are executed, cold objects will be identified, the number of cold objects will be counted, and the proportion of cold objects will be calculated.

6.2 Methodology

If an object is not accessed for a specified time, this object is considered to be cold. As mentioned earlier, there is no clear way to measure the coldness of an object. However, if all accessed objects in the heap can be captured, the remaining are un-accessed objects.

In Chapter 5, an approach to track the object’s activity using Access Barriers has been introduced. Whenever an object is accessed with a read/write operation, an Access Barrier can be triggered. As long as the JIT is turned off, the accessed object can be captured and marked.

6.3 Implementation

6.3.1 The Last Access Timestamp

The Access Barrier provides a good point to capture object activity. This section will discuss how to store the activity information of objects.
As discussed earlier, objects in the heap are always movable via copy-forward or compacting. One solution is similar to the previous Stack-based solution. In this approach some regions can be pinned and have an associated active-bit-map. This solution has some drawbacks. First, only objects in the pinned regions are tracked. Unpinned regions will not be measured. Consequently, the complete information regarding the whole heap will not be obtained. Secondly, whenever the active flag of an object is set in the active-bit-map, the active flag will remain unchanged until the region is unpinned. So, the active flag in the active-bit-map does not reflect the latest change of object activity. For example, an object in the pinned region has been accessed and marked its active flag with 1. Subsequently, this object is not touched, but its active flag is still kept at 1 until the region is unpinned. That means the active-bit-map has limited real-time information.

In order to reflect the real-time activity of objects, a good way is to use an absolute timestamp. Whenever the Access Barrier is triggered, the current access time is recorded with a timestamp, overriding any previous access time. Since the recorded time is the last access time, it reflects accurate access information. The value of the timestamp comes from the system clocks, which have high accuracy. Later, we can check the access time and if the elapsed time exceeds a preset threshold, this object is considered to be cold.

A new question is where to store the last access time? Since the method of the pinned region and the active-bit-map will not be considered, a feasible
way is to embed the timestamp in the object. No matter where the object moves, the timestamp is like a shadow that always accompanies the object. In the JVM, an extra field—timestamp—can be added into the header of an object. Each object has the same header structure; although, they have their own distinct body information. The following represents the layout of the extra field in red color:

![Diagram of an Object with a Timestamp Field](image)

Figure 6.1: Object with a Timestamp Field

### 6.3.2 Measurement

In the Balanced GC, the measurement method is added at the end of the main GC thread. After each GC, the measurement method will be executed. In the measurement method, all objects on the heap are walked, and the elapsed time is measured since the last access. If the elapsed time exceeds the preset threshold of cold objects, this object is considered to be cold. Figure 6.2 shows an example of how to identify cold objects. At any GC cycle, a current time can be obtained with the `now()` system function and
the elapsed time can be calculated. The red bar is the cold threshold. In this example, if the elapsed time of object 2, object 4, and object 5 exceed the length of the red bar, they are considered to be cold objects. However, object 1 and object 3 are not cold.

Figure 6.2: Measurement of Cold Objects

When all objects are walked, the statistical information of cold objects can be logged and output.

6.4 Experimental Results and Analysis

6.4.1 Experimental Environment

The experimental environment is the following configuration.

- 1.87 GHz 8 Core/16 Thread Xeon, 4x E7520, 32GB RAM
During runtime, the heap is divided into 1024-2048 regions. Young regions become tenured when they reach a certain age. New objects are usually allocated in the young regions. As we know, “most objects die young” [30] in Java. Objects in the young regions are not likely to be stable. Since tenured regions’ ages exceed 24, they are the most likely to be stable. Experiments in researching the proportion of cold objects will focus on objects in tenured regions. Objects in young regions are ignored.

### 6.4.2 Density of Cold Objects

The density of cold objects describes cold object proportion, which is defined as the size of cold objects to the size of total used regions. Total used regions are those regions which contain the objects. The regions which are in the Free state will not be considered.
Table 6.1: Density of Cold Objects

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th># of total used regions</th>
<th># of tenured regions</th>
<th>Size of total used regions (MB)</th>
<th>Size of tenured regions (MB)</th>
<th>Size of cold objects (MB)</th>
<th>Density of cold objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECjbb2005</td>
<td>695</td>
<td>440</td>
<td>1,390</td>
<td>546</td>
<td>285</td>
<td>20.50%</td>
</tr>
<tr>
<td>SPECjvm2008 compiler.compiler</td>
<td>745</td>
<td>95</td>
<td>1,490</td>
<td>162</td>
<td>28</td>
<td>1.88%</td>
</tr>
<tr>
<td>SPECjvm2008 compiler.sunflow</td>
<td>558</td>
<td>54</td>
<td>1,116</td>
<td>108</td>
<td>26</td>
<td>2.33%</td>
</tr>
<tr>
<td>SPECjvm2008 derby</td>
<td>1,076</td>
<td>935</td>
<td>2,152</td>
<td>1,769</td>
<td>213</td>
<td>9.90%</td>
</tr>
<tr>
<td>SPECjvm2008 sunflow</td>
<td>161</td>
<td>114</td>
<td>322</td>
<td>222</td>
<td>73</td>
<td>22.67%</td>
</tr>
<tr>
<td>SPECjvm2008 xml.transform</td>
<td>230</td>
<td>170</td>
<td>460</td>
<td>294</td>
<td>84</td>
<td>18.26%</td>
</tr>
<tr>
<td>SPECjvm2008 xml.validation</td>
<td>145</td>
<td>90</td>
<td>290</td>
<td>157</td>
<td>16</td>
<td>5.52%</td>
</tr>
</tbody>
</table>

Notes:

1. Column “Total used regions” is those regions which contain the objects.
2. Column “Tenured regions” are those regions whose ages exceed 24.

Table 6.1 shows that some benchmarks have a higher proportion of cold objects than others. It depends on the particular application. In the benchmark SPECjbb2005, SPECjvm2008 sunflow, and SPECjvm2008 xml.transform, the proportion of cold objects exceeds 18%. The highest proportion reaches
22.67%. The proportion of cold objects is quite encouraging.

### 6.4.3 Estimated Cold Regions

Once cold objects are collected and moved into cold regions, a special GC policy can be applied. For example, once cold objects are moved into cold regions, these regions need not perform the PGC. Doing so can reduce the pause time of the GC. *Estimated cold regions* describes the number of cold regions, which is defined as the size of cold objects to region size, here region size assumes 2MB.

**Table 6.2: Cold Regions**

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Total regions</th>
<th>Estimated cold regions</th>
<th>Ration</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECjbb2005</td>
<td>1,024</td>
<td>143</td>
<td>13.96%</td>
</tr>
<tr>
<td>SPECjvm2008 compiler.compiler</td>
<td>1,024</td>
<td>14</td>
<td>1.37%</td>
</tr>
<tr>
<td>SPECjvm2008 compiler.sunflow</td>
<td>1,024</td>
<td>13</td>
<td>1.29%</td>
</tr>
<tr>
<td>SPECjvm2008 derby</td>
<td>1,076</td>
<td>107</td>
<td>9.94%</td>
</tr>
<tr>
<td>SPECjvm2008 sunflow</td>
<td>1,024</td>
<td>37</td>
<td>3.61%</td>
</tr>
<tr>
<td>SPECjvm2008 xml.transform</td>
<td>1,024</td>
<td>42</td>
<td>4.10%</td>
</tr>
<tr>
<td>SPECjvm2008 xml.validation</td>
<td>1,024</td>
<td>8</td>
<td>0.78%</td>
</tr>
</tbody>
</table>

All estimated cold regions are shown in Figure 6.2. Given SPECjvm2005 as an example, its anticipated cold region count is 143, and total region count is 1024. If all cold objects are collected to cold regions, the proportion of cold region is 13.96% (143/1024). That means there are 13.96% regions which do not have to do PGC frequently. Therefore, the GC pause time could be
significantly reduced.

6.4.3.1 Cold Objects in SPECjbb2005 and SPECjvm2008

Figure 6.3 and Figure 6.4 show experimental data in SPECjbb2005 and SPECjvm2008, respectively. The blue color represents the size of the total used regions and the red color represents the size of the cold objects. The X-axis represents the time in seconds and the Y-axis represents the size of the cold objects in Megabytes. Figure 6.3 shows that the size of the objects in the heap keeps increasing and the size of the cold objects increases as well. Figure 6.4 shows objects’ changes in the heap like a sawtooth pattern, but the size of cold objects stays flat.

![Figure 6.3: Cold objects in SPECjbb 2005](image)

Figure 6.3: Cold objects in SPECjbb 2005
Cold objects in the region come from:

- New cold objects: These objects became cold objects since the current GC cycle.

- Continuing cold objects: These objects continue to keep cold since the last GC cycle.

- Moving cold objects: These objects are cold in another region, and at this cycle they are copied to the current region by Copy-forward or Compacting algorithms.

Figure E.1 in the Appendix E shows that the majority of cold objects are from “continuing cold” in SPECjbb2005.
6.4.5 Reactivation of Cold Objects

A cold object in the cold region has three states over time.

- continue to stay cold - Cold object stays alive and is still un-accessed.
- reactivate - Cold object is accessed by Java programs again.
- become garbage - Cold object becomes dead.

When a cold object is accessed, it is re-activated. The reactivation rate reflects the probability of cold objects being reactivated. Ideally, the lower the reactivation rate, the better. If too many cold objects in a cold region are accessed, this cold region has a mixture of cold objects and active objects, and the active objects will have to be moved out of the cold region. Reactivation rate describes the reactivation proportion, which is defined as the number of reactivated cold objects to the total number of cold objects.

Table 6.3: Reactivation of Cold Objects

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Min</th>
<th>Max</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECJbb2005</td>
<td>0.00%</td>
<td>6.45%</td>
<td>0.82%</td>
</tr>
<tr>
<td>SPECjvm2008 compiler.compiler</td>
<td>0.00%</td>
<td>0.54%</td>
<td>0.01%</td>
</tr>
<tr>
<td>SPECjvm2008 compiler.sunflow</td>
<td>0.00%</td>
<td>0.70%</td>
<td>0.00%</td>
</tr>
<tr>
<td>SPECjvm2008 derby</td>
<td>0.00%</td>
<td>3.65%</td>
<td>0.04%</td>
</tr>
<tr>
<td>SPECjvm2008 sunflow</td>
<td>0.00%</td>
<td>0.13%</td>
<td>0.00%</td>
</tr>
<tr>
<td>SPECjvm2008 xml.transform</td>
<td>0.00%</td>
<td>0.54%</td>
<td>0.01%</td>
</tr>
<tr>
<td>SPECjvm2008 xml.validation</td>
<td>0.00%</td>
<td>0.94%</td>
<td>0.02%</td>
</tr>
</tbody>
</table>
Table 6.3 shows that the average reactivation rate of cold objects in each experiment is less than 1%, and the maximum rate is quite low as well. Results show that once objects become cold, they rarely re-activate. Cold objects rarely warm up, which gives us more encouragement to harvest cold objects to cold regions with little concern of a penalty later if they were to become re-activated.

### 6.4.6 Garbage of Cold Objects

When a cold object is not reachable any more, it is dead. The garbage rate reflects the life-cycle of cold objects. *Garbage rate* describes the garbage cold object proportion, which is defined as the proportion of garbage cold objects to the total number of cold objects.

Table 6.4: Garbage Rate of Cold Objects

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Min</th>
<th>Max</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECJbb2005</td>
<td>0.00%</td>
<td>5.35%</td>
<td>0.12%</td>
</tr>
<tr>
<td>SPECjvm2008 compiler.compiler</td>
<td>0.00%</td>
<td>0.92%</td>
<td>0.01%</td>
</tr>
<tr>
<td>SPECjvm2008 compiler.sunflow</td>
<td>0.00%</td>
<td>0.95%</td>
<td>0.00%</td>
</tr>
<tr>
<td>SPECjvm2008 derby</td>
<td>0.00%</td>
<td>21.32%</td>
<td>0.06%</td>
</tr>
<tr>
<td>SPECjvm2008 sunflow</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>SPECjvm2008 xml.transform</td>
<td>0.00%</td>
<td>98.00%</td>
<td>0.46%</td>
</tr>
<tr>
<td>SPECjvm2008 xml.validation</td>
<td>0.00%</td>
<td>12.10%</td>
<td>0.09%</td>
</tr>
</tbody>
</table>

Table 6.4 shows that the average garbage rate of cold objects in the experiments is less than 1%, and the maximum rate is quite low as well, except for SPECjvm2008 xml.transform, which shows a maximum garbage rate of cold objects.
objects is 98.00%. Analyzing its experimental log, this exception is caused by a few samples. For example, there are only six objects, but five objects die. Otherwise, Table 6.4 shows that once objects become cold, they rarely become garbage.

6.5 Summary

In this chapter, the proportion of cold objects has been investigated. Whenever an object is accessed, a timestamp is placed in the header of the object. The elapsed time is calculated since the last accessed time to determine whether an object is cold or not. Seven groups of benchmarks are run, and the density of cold objects has been obtained, which shows that the proportion of cold objects can reach maximally 22.67% in some real life applications. The results show that the topic of cold objects is worth researching. Furthermore, the source of cold objects is analyzed as well. The re-activate rate and garbage rate of cold objects are researched, and some important observations are collected from experimental data. Once objects become cold, they continue to stay cold and are rarely reactivated or become garbage.
Chapter 7

Evaluation of the Stack-based Approach

The Stack-based approach periodically walks the stack frames, and each walk is called a sampling. Since sampling is not continuous, during the sampling interval some information will be missed. One question that arises is how many cold objects are missed because of discontinuous sampling. This chapter will present a measuring method and evaluate the Stack-based solution.

7.1 Methodologies

The Access Barrier can capture all read/write access operations, and does not miss any access information under the JIT-off condition. So, the Barrier-based solution is considered to be an ideal way to capture and mark active
objects. However, the Barrier-based solution has a restriction - it can only work in the JIT-off mode if all accesses are required to be captured. Since the Access Barrier is reliable in the JIT-off mode, the Access Barrier will be considered as a baseline to evaluate the Stack-based approach.

In order to make sure that the evaluation is valid, both the Barrier-based solution and the Stack-based solution should execute the same program for the same duration. This is necessary to obtain comparable data.

The following are the conditions under which the programs are run.

1. The same Java application.

2. The same running environment - Java Virtual Machine.

3. The same pinned regions (selection/deselection) algorithm.

4. The same cold object collection mechanism.

5. The same active-bit-map data structure, but just two instances - one is for the Stack-based solution, another is for the Barrier-based solution.

6. The same running parameters and options.

However, both approaches run under different mechanisms. They mark different active-bit-maps with their respective method. That is, when an object is accessed, the Access Barrier is triggered, and the Barrier-based marks active flags at its own active-bit map. The Stack-based walks references in the stack frame, and marks active flags in its own active-bit map.
7.2 Extension of the Running Duration Under JIT-off Mode

As mentioned earlier, the Access Barrier does not work in the JIT mode, and the running speed will be slower in the JIT-off mode than in the normal JIT-on mode. The Stack-based solution in Chapter 4 is executed under the JIT-on mode. In order to get close to the previous total of bytecodes executed, the running time of the Java program has to be prolonged. The extension of running time is subject to Running scale, which is defined as Java application performance in JIT-off mode to Java application performance in JIT-on mode.

Since both JIT-on and JIT-off change the memory access pattern in a simple linear scaling manner, Java program performance can reflect the amount of executed bytecode. Usually, performance is measured in operations per minute (Ops/m) in the SPECjvm2008. In order to make up the difference in the JIT-off mode, the running time will be extended in the JIT-off mode. For example, performance of the JIT-on mode is 100 Ops/m, and the performance of the JIT-off mode is 2 Ops/m. Then the running time will be extended by a factor of 50. Suppose a Java application runs 1 hour in the JIT-on mode, now it will have to run 50 hours in the JIT-off mode.
7.3 Matching of the Sampling Interval Under JIT-off Mode

When the running time is prolonged under the JIT-off mode, if the sampling interval is unchanged, then the sampling count will be increased by the same scale as well. In order to have an accurate comparison between the JIT-off mode and the JIT-on mode, they should have an identical sampling count even though they have different running times. The basic idea is that the sampling count should be kept identical and the sampling interval should be increased. The expected sampling count for JIT-off within the same running duration as JIT-on mode should satisfy $\text{Expected\_sampling\_count}$, which is defined as sampling count in JIT-on mode to $\text{Running\_scale}$.

Once the expected sampling count is obtained, the expected sampling interval can be acquired experimentally. The benchmark will be executed under the JIT-off mode for identical running time as the JIT-on mode. When multiple experiments can be done with different sample intervals, then the measured sampling count can be obtained. The final sampling interval will be found where the measured sampling count is close to the expected sampling count. Suppose the sampling count is measured to be 10,000 within 1 hour of running time in the JIT-on mode, and the $\text{Running\_scale}$ is 20, then the expected sampling count should be 500 within 1 hour running time in the JIT-off mode. After these are determined, many experiments should be performed. Finally, an experiment whose sampling count is close to 500 can be reached.
7.4 Terms

Several metrics are used to evaluate the Stack-based approach, this section will introduce them.

1. Collectible Pinned Region
   Collectible Pinned Region is a pinned region that has no new active references within the cold threshold.

2. AverageConvergenceTime
   The Convergence Time is the duration where a pinned region becomes a collectible pinned region. That is, since a region is pinned, how much time does it need to become a collectible pinned region? The AverageConvergenceTime is an average of Convergence Time, and it reflects the speed that a pinned region needs before collecting cold objects. The lower the AverageConvergenceTime is, the more efficient the identification of cold objects.

3. AllObjects
   AllObjects is the number of all objects in the collectible pinned regions.

4. MarkedObjects
   MarkedObjects is the number of objects that have been marked as active in the collectible pinned regions.

5. CollectibleColdObjects
   CollectibleColdObjects is the number of objects that have not been
marked as active in the collectible pinned region. That is, these cold objects can be collected.

6. AllObjectMass

AllObjectMass is the size of all objects in the collectible pinned regions.

7. MarkedObjectMass

MarkedObjectMass is the size of all objects that have been marked as active in the collectible pinned regions.

8. CollectibleColdMass

CollectibleColdMass is the size of all objects that have not been marked as active in the collectible pinned region. That is, these cold objects can be collected.

9. falseInactivity

False Inactive objects are those which are considered to be inactive with the Stack-based solution, but are marked as active with the Barrier-based solution. Because of non-continuous sampling, the Stack-based solution misses some objects’ activities, and these objects are considered to be un-accessed and inactive. Actually these objects’ activities are captured by the Access Barrier, and they have been accessed.

The falseInactivity is the ratio of false inactive object in all inactive objects. The falseInactivity reflects the missing of some active objects, which is a significant metric to evaluate the Stack-based solution. The lower this value is, the better the Stack-based solution is.
10. falseActivity

false active objects are those which are marked as active with the Stack-based solution, but are considered as inactive with the Barrier-based solution. For example, suppose a mutator thread is suspended before a region is pinned. Some references are already in this mutator thread. When this mutator thread is resumed, those references will be walked by the Stack-based solution and they are marked as active by the Stack-based solution, but the Barrier-based solution cannot capture them and they are marked as inactive by the Barrier-based solution. Another scenario is a parameter that is never used in the method. However, it can be identified in the stack by Stack-based, but it cannot be captured by Access Barrier. The falseActivity is defined as the number of false active objects to the number of all active objects.

11. hitRate

All marked objects in the Stack-based solution are the number of objects marked as active with the Stack-based solution. All marked objects in the Barrier-based solution are all objects which are marked as active with the Barrier-based solution.

The hitRate is defined as the marked object difference between in Barrier-based and in Stack-based to the marked objects in Barrier-based.
7.5 Experimental Environment

The experimental environment is the following configuration, which is a Virtual Box (TM) [16] environment.

- CPU: Intel(R) Core(TM) i7-3770 CPU @3.40GHz,
- Memory: 4.00 GB
- Linux OS: CentOS
- GC policy: Balanced GC
- Maximum pinned regions: 64
- JIT: Turn-off
- Heap Size: -Xms2048M -Xmx4096M

7.6 Experiments

7.6.1 Determination of Running Duration and Sampling Interval in Derby

In order to determine running duration and sampling interval for Derby, two groups of experiments are performed. The first group of experiments is executed with the JIT turned on, the Derby benchmark is run for 1 hour with a 1ms sampling interval, 6 minutes cold duration and 3 iterations. The
second group of experiments is executed with the JIT turned off, the Derby benchmark is run again for 6 minutes cold duration, 3 iterations, and 1 hour with different sampling intervals. The average statistical information is shown in Table 7.1.

Table 7.1: Performance Under JIT-on and JIT-off Mode in Derby

<table>
<thead>
<tr>
<th>JIT mode</th>
<th>Sampling interval</th>
<th>Average performance</th>
<th>Average sampling count</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT-on</td>
<td>1 ms</td>
<td>163.95 Ops/minute</td>
<td>2,239,633</td>
</tr>
<tr>
<td>JIT-off</td>
<td>1 ms</td>
<td>2.68 Ops/minute</td>
<td>2,896,085</td>
</tr>
<tr>
<td>JIT-off</td>
<td>90 ms</td>
<td>2.72 Ops/minute</td>
<td>51,456</td>
</tr>
<tr>
<td>JIT-off</td>
<td>100 ms</td>
<td>2.75 Ops/minute</td>
<td>39,210</td>
</tr>
<tr>
<td>JIT-off</td>
<td>120 ms</td>
<td>2.81 Ops/minute</td>
<td>32,840</td>
</tr>
</tbody>
</table>

Comparing the performance between the JIT-on and the JIT-off modes, the performance of the JIT-on mode is around 60 times faster than the performance of the JIT-off mode. So, the running duration will be prolonged 60 times. That is, the benchmark has to be run 60 hours under the JIT-off mode, which will be the equivalent to 1 hour under the JIT-on mode.

In this experiment, the Expected sampling count under JIT-off mode can be calculated according to definition in 7.3, which is 37,327 (2,239,633/60). The sampling count can be looked up in Table 7.1. The Average sampling count in the second last row in Table 7.1 is 39,210 - which is close to the Expected sampling count (37,327). So, the case with 100ms sampling interval is selected in the JIT-off mode.
7.6.2 Determination of Running Duration and Sampling Interval in Compiler.compiler

Similarly, two groups of experiments are performed in Compiler.compiler. The first group of experiments is executed with the JIT turned on, the Compiler.compiler is run for 1 hour with a 1ms sampling interval, 6 minutes cold duration and 3 iterations. The second group of experiments is executed with the JIT turned off, the Compiler.compiler is run again for 6 minutes cold duration, 3 iterations, and 1 hour with different sampling intervals. The average statistical information is shown in Table 7.2.

Table 7.2: Performance Under JIT-off Mode in Compiler.compiler

<table>
<thead>
<tr>
<th>JIT mode</th>
<th>Sampling interval</th>
<th>Average performance</th>
<th>Average sampling count</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT-on</td>
<td>1 ms</td>
<td>82.82 Ops/minute</td>
<td>2,576,145</td>
</tr>
<tr>
<td>JIT-off</td>
<td>1 ms</td>
<td>7.34 Ops/minute</td>
<td>2,158,904</td>
</tr>
<tr>
<td>JIT-off</td>
<td>10 ms</td>
<td>7.28 Ops/minute</td>
<td>345,930</td>
</tr>
<tr>
<td>JIT-off</td>
<td>12 ms</td>
<td>7.31 Ops/minute</td>
<td>288,591</td>
</tr>
<tr>
<td>JIT-off</td>
<td>15 ms</td>
<td>7.59 Ops/minute</td>
<td>224,587</td>
</tr>
</tbody>
</table>

In Compiler.compiler, a similar analysis as with Derby leads to the conclusion that a 12 hour running duration and a 15 ms sampling interval should be selected.
7.6.3 Number of Cold Objects

The original cold duration is 6 minutes in the JIT-on mode. In Derby the cold duration threshold in the JIT-off mode will be prolonged by 60 times, which is 360 (6 x 60) minutes. In Compiler.compiler, it will be prolonged by 12 times, which is 72 (6 x 12) minutes. At the same time, the running time will be extended 60 times and 12 times under the JIT-off mode as well. Table 7.3 shows experimental results, the data in the column “Barrier-based” is collected by the Barrier-based solution, while the data in the column “Stack-based” is collected by the Stack-based solution.

The metric—AverageConvergenceTime—reflects the convergence speed. The AverageConvergenceTime in the Barrier-based solution is 2.5 times faster than in the Stack-based solution in Derby, in Compiler.compiler the Barrier-based solution is 1.42 times faster than the Stack-based solution. The reason is that each access can be captured by the Access Barrier, while the Stack-based solution might miss some reference accesses. Therefore, the Stack-based solution needs more time to capture active objects.

It is not surprising that the Barrier-based solution can harvest more cold objects than the Stack-based solution. For example, in Derby the number of collectible pinned regions in the Barrier-based solution is 5 times more objects than in the Stack-based solution, the number of cold objects in the Barrier-based solution is 11.78 times more objects than that in the Stack-based solution, and the size of cold objects is 10.42 times larger in the Barrier-based solution than that in the Stack-based solution. The reason is that the
Table 7.3: Number of Cold Objects

<table>
<thead>
<tr>
<th>Items</th>
<th>Barrier-based</th>
<th>Stack-based</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derby with 1 x 60 hours running time and 6x60 minutes cold duration threshold</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collectible pinned regions</td>
<td>85</td>
<td>17</td>
<td>5 : 1</td>
</tr>
<tr>
<td>AverageConvergenceTime (in Seconds)</td>
<td>27,721.62</td>
<td>69,497.71</td>
<td>1 : 2.50</td>
</tr>
<tr>
<td>AllObjects</td>
<td>1,485,531</td>
<td>72,290</td>
<td></td>
</tr>
<tr>
<td>MarkedObjects</td>
<td>673,272</td>
<td>3,350</td>
<td></td>
</tr>
<tr>
<td>ColdObjects</td>
<td>812,259</td>
<td>68,940</td>
<td>11.78 : 1</td>
</tr>
<tr>
<td>AllObjectMass (in Bytes)</td>
<td>172,021,784</td>
<td>32,882,056</td>
<td></td>
</tr>
<tr>
<td>MarkedObjectMass (in Bytes)</td>
<td>129,221,416</td>
<td>28,773,560</td>
<td></td>
</tr>
<tr>
<td>ColdObjectMass (in Bytes)</td>
<td>42,800,368</td>
<td>4,108,496</td>
<td>10.42 : 1</td>
</tr>
</tbody>
</table>

Compiler.compiler with 1 x 12 hours running time and 6x12 minutes cold duration threshold

<table>
<thead>
<tr>
<th>Items</th>
<th>Barrier-based</th>
<th>Stack-based</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collectible pinned regions</td>
<td>64</td>
<td>28</td>
<td>2.29 : 1</td>
</tr>
<tr>
<td>AverageConvergenceTime (in Seconds)</td>
<td>4440.64</td>
<td>6311.96</td>
<td>1 : 1.42</td>
</tr>
<tr>
<td>AllObjects</td>
<td>1,390,739</td>
<td>376,824</td>
<td></td>
</tr>
<tr>
<td>MarkedObjects</td>
<td>141,772</td>
<td>279</td>
<td></td>
</tr>
<tr>
<td>ColdObjects</td>
<td>1,248,967</td>
<td>376,545</td>
<td>3.32 : 1</td>
</tr>
<tr>
<td>AllObjectMass (in Bytes)</td>
<td>129,565,792</td>
<td>53,923,840</td>
<td></td>
</tr>
<tr>
<td>MarkedObjectMass (in Bytes)</td>
<td>20,090,576</td>
<td>1,280,616</td>
<td></td>
</tr>
<tr>
<td>ColdObjectMass (in Bytes)</td>
<td>109,475,216</td>
<td>52,643,224</td>
<td>2.08 : 1</td>
</tr>
</tbody>
</table>

Barrier-based solution has less AverageConvergenceTime and it can harvest more collectible pinned regions and more cold objects.
7.6.4 Reliability Analysis of Stack-based Solution

Table 7.4, Table 7.5, and Table 7.6 show falseInactivity, falseActivity, and hitRate, respectively. They are captured by the Stack-based solution.

Table 7.4: FalseInactivity

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Inactive objects</th>
<th>falseInactivity</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derby</td>
<td>68,940</td>
<td>1,117</td>
<td>1.62%</td>
</tr>
<tr>
<td>Compiler.compile</td>
<td>376,545</td>
<td>1,191</td>
<td>0.32%</td>
</tr>
</tbody>
</table>

Table 7.5: FalseActivity

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Active objects</th>
<th>falseActivity</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derby</td>
<td>3,350</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Compiler.compile</td>
<td>279</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Table 7.6: HitRate

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Active objects in Barrier-based</th>
<th>Active objects in Stack-based</th>
<th>hitRate Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derby</td>
<td>4,467</td>
<td>3,350</td>
<td>74.99%</td>
</tr>
<tr>
<td>Compiler.compile</td>
<td>1,470</td>
<td>279</td>
<td>18.98%</td>
</tr>
</tbody>
</table>

In Derby, the falseInactivity ratio is 1.62%. In Compiler.compile, the falseInactivity ratio is 0.32%. Both of them are significantly low. It reflects the fact that there are few objects that are incorrectly classified as cold. The data
supports the hypothesis that cold objects can be identified by the Stack-based approach, which is encouraging. Furthermore, the falseActivity ratio is 0 in both Derby and Compiler.compiler in this experiment. In Derby the hitRate ratio is 74.99% and in Compiler.compiler the hitRate ratio is 18.98%, which shows that around 25% of objects in Derby and 81% of objects in Compiler.compiler are failed to be marked as active because of non-continuous sampling. Here is an explanation why hitRate in Compiler.compiler is low, but it obtains a low falseInActivity. There are 279 marked active objects and 1191 false inactive objects in the Stack-based, so actually there should have 1470 active objects. We notice application’s activity is significant low, which is 1470/376,825 = 0.39%, so even though a large fraction is missed (1191/1470=81%), it still has a low falseInActivity(0.32%), since 1191 is a small portion of 376,825.

7.6.5 Convergence Time Analysis

In Derby, the Barrier-based solution has 85 collectible pinned regions. That is, the Barrier-based solution has identified 85 regions where cold objects were found and can be collected, while the Stack-based solution has 17 collectible pinned regions. Although the Barrier-based solution has more collectible pinned regions than the Stack-based solution, the 17 collectible pinned regions in the Stack-based solution are completely included in the Barrier-based collectible pinned regions. That is, collectible pinned regions in the Stack-based solution are a subset of the collectible pinned regions in
the Barrier-based solution.

Figure 7.1: Convergence Time in Derby

Figure 7.1 shows a convergence time comparison between Stack-based and Barrier-based solutions. The X-axis represents 17 common collectible pinned regions, the Y-axis represents convergence time. In the Barrier-based solution, the maximum convergence time is less than 500 minutes, while in the Stack-based solution, the maximum convergence time reaches more than 2500 minutes. In Figure 7.1, an observation is that the Barrier-based solution converges significantly faster than the Stack-based solution.

Similarly, in Compiler.compiler the Stack-based solution found 28 correctable regions, which are included in the 64 found by the Barrier-based solution. In Figure 7.2, the maximum convergence time in the Barrier-based solution is less than 100 minutes. While in the Stack-based solution, the convergence
time in the majority of collectible pinned regions is less than 100 minutes as well, only 3 collectible pinned regions have a high convergence time.

### 7.7 Summary

When evaluating the Stack-based solution, three metrics—AverageConvergenceTime, falseInactivity and falseActivity—are the most significant.

1. **AverageConvergenceTime** reflects the collecting efficiency of the Stack-based solution, which means how fast the Stack-based solution identifies the cold objects.

2. **FalseInactivity Ratio** reflects the correctness of the Stack-based solution which means how many active objects will be incorrectly considered as
cold objects because of non-continuous sampling.

3. FalseActivity Ratio also reflects the correctness of the Stack-based solution which means how many inactive objects will be incorrectly considered as active objects.

In Derby, experimental data shows the AverageConvergenceTime in the Stack-based solution is 2.5 times longer than that in the Barrier-based solution. In Compiler.compiler, the AverageConvergenceTime in the Stack-based solution is less than 1.7 times longer than that in the Barrier-based solution. These results of the proportions are acceptable. Moreover, in Derby, experimental data shows there is 1.62% falseInactivity in the Stack-based solution. In Compiler.compiler, experimental data shows there is 0.32% falseInactivity. For both falseActivity is 0%. It can be concluded that identifying cold objects with the Stack-based solution is reasonable.

Table 7.7: Additional 4 Representative Benchmarks in SPECjvm2008 Suite

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>AverageConvergence Time ratio (Barrier-based : Stack-based)</th>
<th>FalseInactivity Ratio</th>
<th>FalseActivity Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler.sunflow</td>
<td>1 : 6.41</td>
<td>1.16%</td>
<td>1.13%</td>
</tr>
<tr>
<td>Xml.transform</td>
<td>1 : 10.61</td>
<td>0.23%</td>
<td>0.06%</td>
</tr>
<tr>
<td>Xml.validation</td>
<td>1 : 1.82</td>
<td>0.14%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Sunflow</td>
<td>1 : 6.61</td>
<td>0.61%</td>
<td>0.53%</td>
</tr>
</tbody>
</table>

Additionally 4 representative benchmarks in the SPECjvm2008 have been run, and results have been shown in Appendix A-D. The AverageConver-
vergenceTime ratio, FalseInactivity ratio and FalseActivity are shown in Table
7.7. It is not surprising that the AverageConvergenceTime in the Stack-based
solution is longer than in the Barrier-based solution. However, most falseIn-
activity ratios in Table 7.7 are less than 1%, the largest falseInactivity ratio
is 1.16%. Most of falseActivity ratio in Table 7.7 are less than 1% as well,
the worst falseInactivity ratio is 1.13%. The results confirm the suitability
of the Stack-based solution.
Chapter 8

Performance Optimization — Activity Bit Embedding

This chapter will present an approach which attempts to simplify the active-bit-map in the Stack-based solution.

8.1 Original Approach to Active-bit-map

In the previous Stack-based approach, when active objects are sampled, the mutator thread is responsible for walking its own stack frames, picking out active references that belong to pinned regions, and storing them in buffers. When a mutator thread finishes sampling, it waits for the next sampling signal from the daemon thread. The daemon thread periodically walks the buffers in the mutator threads. All references in the buffers are considered to
be active. The daemon thread checks references’ active-bit in the active-bit-map. If the reference’s active-bit is not set, then a corresponding active-bit is set.

Figure 8.1 shows the approach to active-bit-map: the procedure of walking the stack frames and marking the active-bit-map.

1. Mutator thread: each mutator thread has an internal reference buffer and its own stack. The mutator thread is responsible for walking its own stack frames and storing the references in a buffer. Only the references that are located in the pinned regions are stored in buffers,
the references that stay in non-pinned regions are discarded.

2. Daemon thread: the daemon thread walks distributed reference buffers in different mutator threads and marks the corresponding active bits in the active-bit-map.

The above procedure shows the marking procedure is divided into two steps. The first step is that the mutator thread walks the stack frames and stores the references in buffers. The second step is that the daemon thread walks reference buffers and marks the corresponding active bits.

### 8.2 An Approach to Embedded Activity Bit

The Active-bit has been described in the active-bit-map in Chapter 4. The active-bit-map is an independent data structure which is described in Section 4.4. Each 1-bit in the active-bit-map corresponds to 8-bytes in the heap. However, an active-bit-map does not have to be allocated to record the active-bit of references, instead the active-bit could be embedded in the body of the object. As only one bit is needed, the active-bit occupies a spare bit in the object. In the JVM it is feasible to find a spare bit in the object. When the mutator thread walks the stack frames, it checks the active-bit embedded in the objects. If the active-bit is not set, then active-bit is set explicitly, as shown in Figure 8.2. The references do not have to be buffered in this new approach. The daemon thread is not involved in walking the
buffers as well. In this embedded-activity-bit approach, there is only one step required.

The issue of synchronization has been resolved naturally in this specific scenario when multiple threads access embedded-activity-bit. The first is that all operations of setting the active-bit are constant write operations, which always write the constant bit of “1”. The second is that both write operations and read operations are sequential. The read operation only occurs in the GC phase, which will stop the world. During the GC phase, there are no write operations to set the active-bit. The write operation only occurs in the running phase of the mutator threads. During the running phase of the mutator threads, there are no read operations of the active-bit. So there is no lock for the access of the embedded-activity-bit in this approach.

Figure 8.2: Marking Active Objects Using Embedded-activity-bit
8.3 Experiment

The performance under the Active-bit-map approach is compared with the Embedded-activity-bit approach in this experiment. Furthermore, four metrics are recorded in this experiment.

- Application performance
- Walking Stack Time - time spent on walking stack frames.
- Number of walked references - total number of references in the stack frames that are walked.
- Number of setting active-bit references - total number of references that are newly active and will be marked with the active flag.

In the Active-bit-map approach, Walking Stack Time is only the time that the mutator thread samples the stack frames and stores the references in buffers. Walking Stack Time does not include the time where the daemon thread walks the buffers and sets the active-bit. The experimental hardware environment is the same as in Section 6.4.1, running time is 1 hour x 2 iterations with the JIT on. There are 16 Java application threads, and each CPU binds a Java thread and runs in parallel. The experimental results are shown in Table 8.1.
Table 8.1: Comparison Between Active-bit-map Approach and Embedded-activity-bit Approach

<table>
<thead>
<tr>
<th>Item</th>
<th>Application Performance (Ops/m)</th>
<th>Walking Stack Time (seconds)</th>
<th>Number of walked Reference</th>
<th>Number of setting Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active-bit-map approach</td>
<td>214.89</td>
<td>4,671</td>
<td>23,968,355,443</td>
<td>433,571</td>
</tr>
<tr>
<td>Embedded activity bit approach</td>
<td>157.40</td>
<td>30,641</td>
<td>26,802,439,071</td>
<td>453,626</td>
</tr>
</tbody>
</table>

In the “Active-bit-map approach” row in Table 8.1, the time where the daemon thread walks the buffers and sets the active-bit is not shown in the above table, which is 515 s.

The results show that the application performance has a sharp decrease of 26.75% from 214.89 Ops/m to 157.40 Ops/m. The Walking Stack Time in the Embedded-activity-bit approach is 6.56 times bigger than in the Active-bit-map approach (30,641 s vs 4,671 s).

### 8.3.1 The Computing Time

The computing time spent on walking stack frames can be calculated with the following formula.

\[
\text{Proportion of walking stack frames} = \frac{\text{Total time of walking stack frames}}{(2 \text{ hours}) \times (\text{numbers of cpu core})} 
\]

(8.1)
For two different approaches, the calculation looks at

1. Active-bit-map approach: \( \frac{4,671 \text{ s}}{2 \times 3600 \text{ s} \times 16} = 3.61\% \).

2. Embedded-activity-bit approach: \( \frac{30,641 \text{ s}}{2 \times 3600 \text{ s} \times 16} = 26.60\% \).

The computing time spent on walking stack frames is profiled with “perf” tool [11] in linux, and run for 30 minutes. The profiled results captured by perf tool show that cpu usage spent on walking stacks is 2.34% in active-bit-map, and 20.76% in embedded-activity-bit. The experimental results are close to the profile results - 3.61% vs 2.34% and 26.60% vs 20.76%. The results confirm the validation of the above experimental data. This group of data also shows the operation of marking active references in the Embedded-activity-bit approach is extremely costly, and CPU usage reaches 20.76%.

8.3.2 Time Cost in Each Reference Access Operation

The time expense in each reference access operation is calculated, which provides a quantified number to explain how much time is spent on walking each reference.

In the Active-bit-map approach, the time cost of walking each active reference is 0.19 \( \mu \text{s} \), while in the Embedded-activity-bit approach it is 1.14 \( \mu \text{s} \), which is 5.87 times bigger than in the Active-bit-map approach.
Table 8.2: Time Cost on Accessing Each Reference

<table>
<thead>
<tr>
<th>Item</th>
<th>Walking Stack Time (second)</th>
<th># of walked Reference</th>
<th>The cost spending on each reference (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active-bit-map approach</td>
<td>4,671</td>
<td>23,968,355,443</td>
<td>0.19</td>
</tr>
<tr>
<td>Embedded-activity-bit approach</td>
<td>30,641</td>
<td>26,802,439,071</td>
<td>1.14</td>
</tr>
</tbody>
</table>

8.4 Analysis

The Embedded-activity-bit approach causes a large drop in performance. Initially, the hypothesis of the Embedded-activity-bit approach is to just do one step for marking—marking the new active reference while walking the stack frames. The time can be saved and the application performance can be increased. But the experimental results show the opposite is true.

Although the mutator thread walks and buffers the references, both references in the stack frames and the buffers have good cache locality [32] because both the stack frame and the buffer have contiguous memory. When accessing references in the stack frames and writing the references to the buffers, the operations are significantly faster, because all of the contents are adjacent. The Active-bit-map approach leverages the cache and increases the performance. However, in the Embedded-activity-bit approach each reference access is randomly accessed in the heap, so it will have a great deal of
cache-misses [2] and the access expense will be costly. Because cache-miss or cache-hit cannot be measured for a specific module in this big project, there is no real experimental data to confirm this analysis, a simulated experimental will be made as future work.

8.5 Summary

The Active-bit-map approach samples the stack frame, stores references in buffers in the mutator threads, and concurrently sets the active-bit in the active-bit-map in the daemon thread. The Embedded-activity-bit approach samples the stack frames and at that time sets the active-bit which is embedded in the body of the object in the mutator threads. It seems that the Embedded-activity-bit approach only uses one step to finish the work of marking the active-bit instead. But the Embedded-activity-bit approach—which randomly accesses objects in the heap—is an extremely costly operation with 1.14 $\mu$s for each object. The application throughput has a sharp drop.

The Embedded-activity-bit approach is not an improvement, however, this attempt provides a quantified Walking Stack Time on accessing objects in the heap. The information is helpful for future research and design.
Chapter 9

Performance Optimization — Concurrent Region Activity

9.1 Introduction

In a multi-threaded programming model, parallel computing technology allows multiple threads to be simultaneously executed [4]. In this thesis, mutator threads, GC threads and the daemon thread co-exist at runtime. In the JVM, whenever a GC happens, it will stop the world. All threads except GC threads will be suspended.

In the Stack-based solution, mutator threads and the daemon thread are executed concurrently. Here the daemon thread is similar to the marking-thread in the Balanced GC. The marking-thread is a part of the GC, but the marking-thread is running concurrently with mutator threads, which aims to
minimize the pause times of the GC.

In short, mutator threads and the daemon thread co-exist and are executed concurrently. Both of them execute sequentially with the GC main thread.

The Selection and Deselection algorithms of cold objects in the Stack-based solution are implemented in the main GC thread. Obviously, their execution will affect the pause times of the GC. In order to minimize the pause times caused by the overhead of the implementation of cold objects, we will explore the probability of optimizing the cold objects’ implementation in the GC thread.

Figure 9.1 shows that the Selection, Deselection and the daemon thread are introduced to the current Stack-based solution. The daemon thread is an independent thread, and both Selection and Deselection are a part of the GC thread. This certainly will have an impact on the application performance and the pause times of the GC. This chapter will introduce an optimization.
9.2 Original Design—Sequential Region Activity

The metric Region
Activity has been discussed in subsection 4.7. It reflects the activity of objects in a region. The metric Region
Activity is an input factor in the Selection algorithm. Each mutator declares an array that stores every region activity. For example, if there are 1024 regions, then the size of the array will be 1024. Each field in the array corresponds to a region counter. Each mutator thread counts its own Region
Activity during walking its stack frames. During the GC phase, when the GC thread performs the Selection algorithm, the Selection algorithm requires Region
Activity of each region. At this point, the GC thread has to aggregate Region
Activity, which is located in every mutator thread.

In the original design, a program module is responsible for aggregating the Region
Activity for each region. The module is located at the Selection algorithm in the GC main thread. That means the module will be executed during the GC phase that stops the world. The execution of Region
Activity is considered to be sequential from execution order between the GC thread and mutator threads. In this chapter, sequential region activity is called SequentialRA.

Suppose there are 32 mutator threads, and the heap consists of 1024 regions, the module will loop 32 x 1024 times when the Selection algorithm requires Region
Activity. Furthermore, the Selection algorithm will be executed once
for every GC cycle. That is, the module of aggregating the Region_Activity will be executed once for every GC cycle. It is very costly.

9.3 New Design—Concurrent Region Activity

Each mutator thread still counts its own Region_Activity during the walking of its stack frames. However, the module of aggregating Region_Activity is moved to the daemon thread from the GC thread. The daemon thread will aggregate Region_Activity once for every GC cycle instead. Since the sum of Region_Activity represents the last cycle’s Region_Activity, this modification does not affect the Selection algorithm. When the GC thread performs the Selection algorithm and requires the Region_Activity, the GC thread requests Region_Activity, which has already been prepared ahead by the daemon thread.

There are no synchronization and lock issues regarding access to Region_Activity. Firstly, when the daemon thread aggregates the Region_Activity from a spe-
cific mutator thread, this corresponding mutator thread has already stopped walking their stack-frame for marking active references. Secondly, when the GC thread requires Region_Activity, the daemon thread and mutator threads are suspended, both the GC thread and the daemon thread are executed sequentially. In this chapter, concurrent region activity is called _concurrentRA_.

![Figure 9.3: The Modified Design - ConcurrentRA](image)

### 9.4 Comparison Between SequentialRA and ConcurrentRA

The experimental hardware environment is the same as in Subsection 6.4.1, running time is 1 hour x 4 iterations with the JIT turned on. Two benchmarks—both Derby and Xml.transform—are run. The experimental results are shown in Table 9.1, which is a comparison between SequentialRA and ConcurrentRA. In Table 9.1, the “Application performance” column is from the output of running the benchmark. There are 3 pause time metrics (mean, maximum and total pause times), which are recorded by the IBM GCMV tool [9]. “Harvest count” is an output of cold object feature implementation.
Table 9.1: Comparison Between SequentialRA and ConcurrentRA

<table>
<thead>
<tr>
<th>Item</th>
<th>Application Performance (Ops/m)</th>
<th>Mean pause time (sec)</th>
<th>Maximum pause time (sec)</th>
<th>Total pause time (sec)</th>
<th>Harvest count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derby</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SequentialRA</td>
<td>231.91</td>
<td>0.03</td>
<td>0.69</td>
<td>2,036</td>
<td>9,211,218</td>
</tr>
<tr>
<td>ConcurrentRA</td>
<td>234.24</td>
<td>0.03</td>
<td>0.67</td>
<td>2,001</td>
<td>9,318,071</td>
</tr>
<tr>
<td>Xml.transform</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SequentialRA</td>
<td>246.09</td>
<td>0.05</td>
<td>5.85</td>
<td>1,167</td>
<td>10,447,137</td>
</tr>
<tr>
<td>ConcurrentRA</td>
<td>260.64</td>
<td>0.04</td>
<td>0.45</td>
<td>1,005</td>
<td>10,335,320</td>
</tr>
</tbody>
</table>

In Derby, after the calculation of the aggregation of Region.Activity is moved from the GC main thread to the daemon thread, both the application throughputs and pause times benefit from this optimization. The Java application throughput has the increase of 1.00%. The mean pause time remains unchanged. The total pause time has a decrease of 1.72%, while the Harvest count has an increase of 1.16%.

Likewise, in Xml.transform, both the application throughputs and pause times benefit from optimization. The Java application throughput has an increase of 5.91%. The mean pause time has a decrease of 20%. The total pause time has a decrease of 13.88%, while the Harvest count a decrease of 1.07%.

The results show that optimization is positive effect, it can improve Java application performance and reduce pause times.
9.5 Increasing the Workload in the GC Thread

In order to measure the impact caused by the overhead of the GC, an artificial workload is increased in the GC main thread. The change trends of the pause times and application throughputs will be observed and analyzed. A group of experiments is performed on the basis of no Region_Activity aggregating optimization. The overhead—artificial workload—is lengthened by 30 and 60 times. Eventually, in Derby the computing time of workload is 438.47 seconds and 866.24 seconds, respectively. In Xml.transform the computing times of workload are 371.09 seconds and 843.16 seconds, respectively. Three groups of experimental data are obtained as shown in Table 9.2.

<table>
<thead>
<tr>
<th>Item</th>
<th>Time of workload (s)</th>
<th>Performance (Ops/m)</th>
<th>Pause Time (s)</th>
<th>Harvest Count</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Derby</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x_workload</td>
<td>17.95</td>
<td>231.91</td>
<td>2,036</td>
<td>9,211,218</td>
</tr>
<tr>
<td>30x_workload</td>
<td>438.47</td>
<td>220.76</td>
<td>2,465</td>
<td>9,074,587</td>
</tr>
<tr>
<td>60x_workload</td>
<td>866.24</td>
<td>215.98</td>
<td>2,911</td>
<td>8,526,984</td>
</tr>
<tr>
<td><strong>Xml.transform</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x_workload</td>
<td>9.34</td>
<td>246.09</td>
<td>1,167</td>
<td>10,447,137</td>
</tr>
<tr>
<td>60x_workload</td>
<td>371.09</td>
<td>244.27</td>
<td>1,538</td>
<td>10,175,891</td>
</tr>
<tr>
<td>120x_workload</td>
<td>843.16</td>
<td>242.69</td>
<td>1,941</td>
<td>9,814,244</td>
</tr>
</tbody>
</table>

Table 9.2 shows that by increasing the workload in the GC, the application throughput has been decreased and the pause times of the GC has been increased. The results are not surprising. However, a concern is how much
both the pause times and application throughputs are affected by the increasing the workload in the GC. Based on Table 9.2, the differences for the pause times and application throughputs are calculated.

Table 9.3: Differences with 30x and 60x Workload in the GC Thread

<table>
<thead>
<tr>
<th>Diff workload (s)</th>
<th>Diff pause Time (s)</th>
<th>Diff pause time ratio</th>
<th>Diff performance (Ops/m)</th>
<th>Diff performance ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derby</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>420.52</td>
<td>429</td>
<td>102.02%</td>
<td>-11.15</td>
<td>-4.81%</td>
</tr>
<tr>
<td>848.29</td>
<td>875</td>
<td>103.15%</td>
<td>-15.93</td>
<td>-6.87%</td>
</tr>
<tr>
<td>Xml.transform</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>361.75</td>
<td>371</td>
<td>102.56%</td>
<td>-1.82</td>
<td>-0.75%</td>
</tr>
<tr>
<td>833.82</td>
<td>774</td>
<td>92.83%</td>
<td>-3.4</td>
<td>-1.39%</td>
</tr>
</tbody>
</table>

The results show that the time of workload in GC is completely contributed to pause time, which is consistent with the definition of GC pause time.


9.6 Increasing the Workload in the Daemon

In order to measure the impact caused by the overhead of the daemon thread, the workload is artificially increased in the daemon thread. The change trends of the pause time and application throughput are observed and analyzed. Another group of experiments is performed on the basis of Region_Aggregation optimizing as well. The overhead—artificial workload—is increased by 30 and 60 times. In Derby the computing times of workload are 664.02 seconds and 1331.28 seconds. In Xml.transform the computing times of workload are 751.17 seconds and 1,552.56 seconds, respectively. Three groups of experimental data are obtained as shown in Table 9.4.

<table>
<thead>
<tr>
<th>Item</th>
<th>Overhead Time(s)</th>
<th>Performance (Ops/m)</th>
<th>Pause Time(s)</th>
<th>Harvest Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derby</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x_workload</td>
<td>17.91</td>
<td>234.24</td>
<td>2,001</td>
<td>9,318,071</td>
</tr>
<tr>
<td>30x_workload</td>
<td>664.02</td>
<td>228.72</td>
<td>2,139</td>
<td>8,667,998</td>
</tr>
<tr>
<td>60x_workload</td>
<td>1,331.28</td>
<td>226.41</td>
<td>2,202</td>
<td>7,951,658</td>
</tr>
<tr>
<td>Xml.transform</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x_workload</td>
<td>9.54</td>
<td>260.64</td>
<td>1,005</td>
<td>10,335,320</td>
</tr>
<tr>
<td>60x_workload</td>
<td>795.17</td>
<td>256.14</td>
<td>1,019</td>
<td>9,662,654</td>
</tr>
<tr>
<td>120x_workload</td>
<td>1,552.56</td>
<td>254.07</td>
<td>1,026</td>
<td>9,039,788</td>
</tr>
</tbody>
</table>

Extensive research on the impact by different workloads in the daemon thread has been done. Based on Table 9.4, the differences for the pause times and application throughputs are calculated.
Table 9.5: Differences with 30x and 60x Workload in the Daemon Thread

<table>
<thead>
<tr>
<th>Diff workload (s)</th>
<th>Diff pause Time (s)</th>
<th>Diff pause time ratio</th>
<th>Diff performance (Ops/m)</th>
<th>Diff performance ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derby</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>646.11</td>
<td>138</td>
<td>21.36%</td>
<td>-5.52</td>
<td>-2.36%</td>
</tr>
<tr>
<td>1,313.38</td>
<td>201</td>
<td>15.30%</td>
<td>-7.83</td>
<td>-3.34%</td>
</tr>
<tr>
<td>Xml.transform</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>785.63</td>
<td>14.00</td>
<td>1.78%</td>
<td>-4.5</td>
<td>-1.76%</td>
</tr>
<tr>
<td>1,543.02</td>
<td>21.00</td>
<td>1.36%</td>
<td>-6.57</td>
<td>-2.57%</td>
</tr>
</tbody>
</table>

Figure 9.5: Differences with 30x and 60x Workload in the Daemon Thread

From Table 9.5 it can be observed that the workload in the daemon thread actually affects the pause times, but the effect is not larger than in the previous example. In Derby, a 21.36% workload computing time and a 15.30% workload computing time are converted to the pause time. In Xml.transform, a 1.78% workload computing time and a 1.36% workload computing time...
are converted to the pause times. The reason is that the daemon thread is executed in parallel with mutator threads, while the daemon thread shares computing resources, rather than exclusively occupying all computing resources.

9.7 Summary

When the aggregation of Region_Activity is executed in the GC thread, since the GC will stop the world, and the aggregation module will exclusively occupy computing resources during aggregation, so the aggregation is a much costlier operation. While the daemon thread calculates the Region_Activity, the aggregation module is executed in parallel with mutator threads, since the aggregation shares computing resources, so the overhead of the aggregation is relatively low.

In ConcurrentRA approach of Derby, there is a 1.00% increase of application throughput and a 1.16% decrease of total pause time. While in the.Xml.transform, there are a 5.91% increase of application throughput, a 20% decrease of the mean pause time, and a 13.88% decrease of the total pause time. In comparison with the SequentialRA approach, the ConcurrentRA approach shows benefits of both application throughput and pause time.

With increased workloads in the GC thread and the daemon thread, respectively, the changes of the pause time under different workloads are shown in this chapter. In the GC thread, the computing time of additional work-
loads in the main GC thread is converted completely to the total pause time. While in the daemon thread, the computing time of additional workloads is converted partially to the pause time in the daemon thread. In Derby, a 21.36% and a 15.30% of workload computing time in the daemon thread are converted to the total pause time. While in Xml.transform, there are 1.78% and 1.36% of computing time workload in the daemon thread are converted to the total pause time. It depends on the applications.
Chapter 10

Collection of Cold Objects

The previous chapters have discussed the problem of identifying cold objects and how to optimize the performance of detecting them. In the Stack-based solution, the functionality of the collection of cold objects has been implemented. This chapter will briefly discuss the collection of cold objects.

10.1 Introduction

In the Balanced GC policy, object allocation and garbage collection can be performed on any individual region. The natural hierarchy of the Balanced GC facilitates creation and reclamation of cold regions. When cold regions are created, they are taken from the heap pool. When cold regions are freed, they are released to the heap pool.

This chapter will not discuss the scenario where cold objects are swapped
out to external memory, such as a SSD. The scenario discussed is where cold objects are moved to cold regions, which are physical regions in virtual memory.

10.2 Creation of Cold Regions

The previous chapters have mentioned region states. Each region has a property that represents a region’s state, such as Free region, Managed region, Pinned region, Collectible pinned region, and Cold region.

When the JVM enables the “cold objects” feature, the JVM will create cold regions. The initial number of cold regions is specified through the JVM command options.

When a Free region is changed to Cold region, it means a cold region has been created successfully. When the capacity of cold regions is not sufficient, new cold regions can still be created dynamically.

10.3 Gathering of Cold Objects

Collectible cold objects are retained in a Collectible pinned region. Cold objects are harvested by separating cold objects from active objects.

The approach to harvest cold objects depends on the proportion of collectible cold objects in the current region. In the Balanced GC policy, the cost of moving a cold object is the same as the cost of moving an active object. If
the proportion of collectible cold objects is quite low, it is not expensive to move collectible cold objects to an existing Cold region, as shown as Figure 10.1. Otherwise, active objects are moved to a Managed region, as shown as Figure 10.2. For example, suppose a collectible pinned region has 20% active objects and 80% collectible cold objects. The most efficient way is to move active objects to other Managed region, while the remaining collectible cold objects will still retain in the current region, and change the current region’s property from Collectible pinned region to Cold region.

Figure 10.1: Copy-forward Cold Objects; Black are Cold Objects, Red are Active Objects

Figure 10.2: Copy-forward Active Objects; Black are Cold Objects, Red are Active Objects
When collectible cold objects are moved from the Collectible pinned region to the Cold region, the existing copyingforward algorithm for generational GC in JVM can be leveraged. The collectible cold object moving procedure is similar to the way Java objects ordinarily move during a generational GC. The unique point is that destination region here is a Cold region instead. After collectible cold objects are moved, their references relationships will be maintained in the generational GCs.

10.4 Re-activating of Cold Objects

When a cold object in the cold region is accessed, this cold object is re-activated. It is unavoidable for cold objects to be re-activated in real-life Java applications. When dead objects accumulate in cold regions, a GC should be performed to reclaim dead objects.

10.5 The GC Policy on Cold Regions

In Balanced GC, there are the PGC, the Compactation, the GGC and the Marking-phase, which co-exists with the mutator threads concurrently. These algorithms are performed to accommodate different needs. Ideally there are no active objects and no dead objects in the cold regions, so the PGC and the Compactating do not have to be performed on the cold regions. The Marking-phase is used to mark the live objects. It is essential work for
the GC. So the Marking-phase should be retained for cold regions.
The GGC does garbage collection for the whole heap. Usually the GGC rarely happens because it is a very costly operation. As time goes on, cold objects in the cold regions have the probability of re-activating and are accessed. That means it is likely that there are dead objects in the cold regions over time. The GGC is an opportunity to collect garbage in the cold regions, but the frequency of performing GGC on the cold regions can be reduced. So the GGC should be retained for cold regions.

10.6 Freeing Cold Regions

After the GGC happens on a cold region, this cold region’s state should have one of two states. One is to continue to retain the Cold region state. Another is to change to Free region state. The choice of states depends on the proportion of the remaining cold objects. When the proportion of cold objects is greater than a threshold (usually a high proportion), cold objects do not move and still stay in this region, active objects will move out from this cold region. Therefore, this region still remains a Cold region. When the proportion of cold objects is less than a threshold (usually a low proportion), active objects will be moved to other managed regions while cold objects will moved to other cold regions, then this region can be reclaimed to Free region.
10.7 Cost considerations

In the collection of cold objects, there is additional overhead to the standard Balanced GC. This is addressed in this section. Regarding creation of cold regions, it happens when the JVM starts. Cold regions are a reserved resource from existing regions on the heap, and the number of cold regions is specified as JVM options. Therefore, there is not too much runtime overhead during collection of cold objects.

When cold objects are harvested from collectible pinned regions, or when freeing a cold region, copy-forward will take place. Regardless of moving out cold objects or active objects it will produce overhead. In the standard Balanced GC, there is a standard copy-forward procedure, and the copy-forward of cold objects is similar to standard copy-forward. The PGC in the Balanced GC is a very frequent operation. When PGC happens, a set of selected region candidates are performing copy-forward, and copy-forward on collectible pinned regions takes place in this phase as well. The cost of copy-forward on cold objects is estimated to be equivalent to the cost in standard copy-forward, that means we have an extra subset of copy-forward. Therefore, the cost of harvesting cold objects depends on how many collectible pinned regions are performed copy-forward.

Regarding GGC and GC marking, cold objects do not cause extra overhead, it is a common expense for the whole heap.
10.8 Summary

This chapter briefly discusses the creation of cold regions, two policies of harvesting cold objects in collectible pinned regions, some changes of Balance GC applied in cold regions, and freeing cold regions. Harvesting cold objects will leverage the existing Balanced GC policy, which will make the collection simple and efficient.
Chapter 11

Conclusions

11.1 Overview

An investigation of the proportion of cold objects has been presented in this thesis. Several benchmarks that reflect real life Java applications were analyzed. The results show that cold objects do exist in real life Java applications. SPECjbb2005 can obtain a density of cold objects at 20.50%, and SPECjvm2008 Sunflow can obtain a density of cold objects at 22.67%. The results are encouraging, and the topic of cold objects is worth studying extensively.

In this thesis, a previous work — Stack-based solution to identify cold objects— has been reviewed. The Stack-based solution samples the mutator thread stack, marks the active objects, and harvests the cold objects. However, since the sampling of the Stack-based solution is non-continuous, it might
miss some active reference information. For these reasons, a Barrier-based solution has been proposed. The Access Barrier can capture all read/write operations when objects are accessed, so the Access Barrier will not miss any active references. However, the experimental results show that the functionality of the Access Barrier is affected when the JIT turns on. When Java applications are run, the JIT turns on by default. The Barrier-based solution cannot become a feasible solution to identify cold objects in the JIT-on mode, but the Access Barrier can work in the JIT-off mode, so the Barrier-based solution can still be an accurate measuring tool for cold objects under the JIT-off mode.

Because the Stack-based solution has a non-continuous sampling, this thesis is concerned with the correctness and efficiency of identifying cold objects with Stack-based solution. The Barrier-based solution is used to evaluate and validate the Stack-based solution. In order to validate a scenario in the JIT-on mode, the running duration and the sampling interval are prolonged in the JIT-off mode, respectively. In the JIT-off mode, SPECjvm2008 Derdy was run with 60 hours running duration, 100 ms sample interval, and 6x60 minutes cold threshold. The results show that the Access Barrier solution can harvest more cold objects and has shorter convergence time than the Stack-based solution, which is not surprising. Furthermore, the results show the falseInactitiveness ratio is quite low at 1.62%. In the experiment of SPECjvm2008 Compiler.compile, the results show that the falseInactitiveness ratio is still low at 0.32%. These experimental results show that the correct-
ness and efficiency of the Stack-based solution are convincing. Furthermore, an additional 4 representative benchmarks have been run, their results also support this point. Therefore, the Stack-based solution is an acceptable solution.

In order to minimize the overhead caused by the Stack-based solution, two attempts to optimize performance have been made in this thesis. The first attempt was an embedded-activity-bit approach. The original approach—Stack-based solution—samples the stack frames, stores references in buffers in the mutator threads, and concurrently sets the active-bit in the active-bit-map in the daemon thread. The embedded-activity-bit approach samples the stack frames and at that time sets the active-bit, which is embedded in the body of the object in the mutator threads. It seems that the embedded-activity-bit approach only uses one step to finish the marking work and saves the time of buffering references. However, the embedded-activity-bit approach—which is randomly accessed to objects in the heap—is an extremely costly operation with 1.14$\mu$s for each object, while the Active-bit-map approach cost is 0.19$\mu$s. The embedded-activity-bit approach is not an improvement.

The second attempt was a concurrent Region Activity calculation. The Region Activity calculation is moved from the GC main thread to the daemon thread, which runs concurrently with the mutator threads. The experimental results show that the total pause time has been decreased and the Java application throughput has been increased. In order to extensively study
the impact caused by the implementation of cold objects, an artificial workload was increased in the GC main thread and the daemon thread. The experimental results show positive improvement.

11.2 Future work

The efficiency of harvesting cold objects depends on the quality of pinned regions. Some regions have high activity, while other regions have low activity. Furthermore, some regions have higher proportions of new active objects than other regions. The regions with higher proportions of new active objects have a higher probability of obtaining less convergence time. The Selection algorithm determines which regions are pinned. Furthermore, once some pinned regions have a long convergence time, there should be a more efficient deselection algorithm. Proposed future work is to optimize the algorithm of Selection and Deselection. These algorithm will improve the efficiency of harvesting cold objects.

When cold objects are gathered, two copy-forward approaches have been discussed in Chapter 10. In previous work, an approach to copy cold objects into cold regions has been implemented, but the approach to copying active objects has not been implemented, this work could be considered to improvement as a future work.
Bibliography


Appendix A

compiler.sunflow

Running scale is 13, running duration is 13 hours, sample interval is 16 ms.

<table>
<thead>
<tr>
<th>Items</th>
<th>Barrier-based</th>
<th>Stack-based</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collectible pinned regions</td>
<td>11</td>
<td>4</td>
<td>2.75 : 1</td>
</tr>
<tr>
<td>AverageColdDuration (in Seconds)</td>
<td>2,596</td>
<td>16,641</td>
<td>1 : 6.41</td>
</tr>
<tr>
<td>AllObjects</td>
<td>379,309</td>
<td>150,779</td>
<td></td>
</tr>
<tr>
<td>MarkedObjects</td>
<td>93,498</td>
<td>532</td>
<td></td>
</tr>
<tr>
<td>ColdObjects</td>
<td>285,811</td>
<td>150,247</td>
<td>1.90 : 1</td>
</tr>
<tr>
<td>AllObjectMass (in Bytes)</td>
<td>38,589,816</td>
<td>10,183,872</td>
<td></td>
</tr>
<tr>
<td>MarkedObjectMass (in Bytes)</td>
<td>11,129,104</td>
<td>560,768</td>
<td></td>
</tr>
<tr>
<td>ColdObjectMass (in Bytes)</td>
<td>27,460,712</td>
<td>9,623,104</td>
<td>2.85 : 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inactive objects</th>
<th>falseInactivity Objects</th>
<th>falseInactivity Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>150,247</td>
<td>1,738</td>
<td>1.16%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active objects</th>
<th>falseActivity Objects</th>
<th>falseActivity Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>532</td>
<td>6</td>
<td>1.13%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active objects in Barrier-based</th>
<th>Active objects in Stack-based</th>
<th>hitRate Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>532</td>
<td>2,174</td>
<td>24.47%</td>
</tr>
</tbody>
</table>
Running_scale is 69, running duration is 69 hours, sample interval is 120 ms.

<table>
<thead>
<tr>
<th>Items</th>
<th>Barrier-based</th>
<th>Stack-based</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collectible pinned regions</td>
<td>33</td>
<td>18</td>
<td>1.83 : 1</td>
</tr>
<tr>
<td>AverageColdDuration(in Seconds)</td>
<td>37,520</td>
<td>399,300</td>
<td>1 : 10.64</td>
</tr>
<tr>
<td>AllObjects</td>
<td>911,730</td>
<td>418,761</td>
<td></td>
</tr>
<tr>
<td>MarkedObjects</td>
<td>76,278</td>
<td>15,807</td>
<td></td>
</tr>
<tr>
<td>ColdObjects</td>
<td>835,452</td>
<td>402,954</td>
<td>2.07:1</td>
</tr>
<tr>
<td>AllObjectMass(in Bytes)</td>
<td>88,660,776</td>
<td>37,980,384</td>
<td></td>
</tr>
<tr>
<td>MarkedObjectMass(in Bytes)</td>
<td>14,705,208</td>
<td>2,619,288</td>
<td></td>
</tr>
<tr>
<td>ColdObjectMass(in Bytes)</td>
<td>73,955,568</td>
<td>35,361,096</td>
<td>2.09 : 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inactive objects</th>
<th>falseInactivity Objects</th>
<th>falseInactivity Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>402,954</td>
<td>913</td>
<td>0.23%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active objects</th>
<th>falseActivity Objects</th>
<th>falseActivity Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>15,807</td>
<td>9</td>
<td>0.06%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active objects in Barrier-based</th>
<th>Active objects in Stack-based</th>
<th>hitRate Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>15,807</td>
<td>36,711</td>
<td>43.06%</td>
</tr>
</tbody>
</table>
Appendix C

xml.validation

Running scale is 66, running duration is 66 hours, sample interval is 120 ms.

<table>
<thead>
<tr>
<th>Items</th>
<th>Barrier-based</th>
<th>Stack-based</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collectible pinned regions</td>
<td>6</td>
<td>5</td>
<td>1.20 : 1</td>
</tr>
<tr>
<td>AverageColdDuration (in Seconds)</td>
<td>18,884</td>
<td>34,356</td>
<td>1 : 1.82</td>
</tr>
<tr>
<td>AllObjects</td>
<td>91,541</td>
<td>60,326</td>
<td></td>
</tr>
<tr>
<td>MarkedObjects</td>
<td>12,234</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>ColdObjects</td>
<td>79,307</td>
<td>60,231</td>
<td>1.32 : 1</td>
</tr>
<tr>
<td>AllObjectMass (in Bytes)</td>
<td>8,516,872</td>
<td>4,609,896</td>
<td></td>
</tr>
<tr>
<td>MarkedObjectMass (in Bytes)</td>
<td>1,952,232</td>
<td>6,328</td>
<td></td>
</tr>
<tr>
<td>ColdObjectMass (in Bytes)</td>
<td>6,564,640</td>
<td>4,603,568</td>
<td>1.43 : 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inactive objects</th>
<th>falseInactivity Objects</th>
<th>falseInactivity Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>60276</td>
<td>85</td>
<td>0.14%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active objects</th>
<th>falseActivity Objects</th>
<th>falseActivity Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active objects in Barrier-based</th>
<th>Active objects in Stack-based</th>
<th>hitRate Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>135</td>
<td>37.04%</td>
</tr>
</tbody>
</table>
Appendix D

sunflow

Running scale is 87, running duration is 87 hours, sample interval is 130 ms.

<table>
<thead>
<tr>
<th>Items</th>
<th>Barrier-based</th>
<th>Stack-based</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collectible pinned regions</td>
<td>24</td>
<td>16</td>
<td>1.50 : 1</td>
</tr>
<tr>
<td>AverageColdDuration (in Seconds)</td>
<td>82,255</td>
<td>543,310</td>
<td>1 : 6.61</td>
</tr>
<tr>
<td>AllObjects</td>
<td>698,888</td>
<td>469,872</td>
<td></td>
</tr>
<tr>
<td>MarkedObjects</td>
<td>6,304</td>
<td>1,880</td>
<td></td>
</tr>
<tr>
<td>ColdObjects</td>
<td>692,584</td>
<td>467,992</td>
<td>1.48 : 1</td>
</tr>
<tr>
<td>AllObjectMass (in Bytes)</td>
<td>42,392,160</td>
<td>27,081,504</td>
<td></td>
</tr>
<tr>
<td>MarkedObjectMass (in Bytes)</td>
<td>1,459,680</td>
<td>668,480</td>
<td></td>
</tr>
<tr>
<td>ColdObjectMass (in Bytes)</td>
<td>40,932,480</td>
<td>26,413,024</td>
<td>1.55 : 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inactive objects</th>
<th>falseInactivity Objects</th>
<th>falseInactivity Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>467,992</td>
<td>2,840</td>
<td>0.61%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active objects</th>
<th>falseActivity Objects</th>
<th>falseActivity Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,880</td>
<td>10</td>
<td>0.53%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active objects in Barrier-based</th>
<th>Active objects in Stack-based</th>
<th>hitRate Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,880</td>
<td>16,680</td>
<td>11.27%</td>
</tr>
</tbody>
</table>
Appendix E

Source of Cold Objects in
SPECjbb 2005
Figure E.1: Source of Cold Objects in SPECjbb 2005
Vita

Candidate’s full name: Baoguo Zhou

University attended:

Hubei University of Technology (1989 - 1993)
Bachelor of Science in Computer Science

University of New Brunswick (2013 - 2015)
Master of Science in Computer Science