Liveness Based Garbage Collection for Java

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

Master of Technology

by
Vilay Kandi
Roll No. : 12111025

under the guidance of
Dr. Amey Karkare

Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur
July, 2014

CERTIFICATE

It is certified that the work contained in this thesis entitled " **Liveness Based Garbage Collection for Java** ", by **Vilay Kandi (Roll No. 12111025)**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare 19/6/2014

(Dr. Amey Karkare)
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur
Kanpur-208016

June, 2014

Abstract

Garbage collection is one of the most important features of programming languages. It frees the programmer from the burden of manual deallocation of memory. The most common technique used to traverse objects over heap is reachability tracing. Garbage Collector collects only those objects which are not reachable. There are still number of objects on heap which are reachable but not live. To collect such objects, liveness of objects and their references need to be checked instead of reachability alone while tracing on heap.

To check liveness of object and their references, liveness information should be available at runtime in garbage collector. We have achieved it with the help of infrastructure like Soot and JikesRVM. Soot transfers information to class file and serialized file. JikesRVM takes care of this information at runtime. Performance of garbage collector is measured in terms of number of objects live on heap against number of objects reachable. We got reasonable improvement in garbage collection with some of the standard benchmarks. Results show that liveness based garbage collection always performs better or equal to reachability based garbage collection in terms of memory.

Dedicated to
my dear family, advisor and friends

Acknowledgement

I wish to express my sincere gratitude and regards to Dr. Amey Karkare for his constant motivation, support and encouragement throughout my thesis work. His invaluable suggestions and high spirited mentorship always held me high on positive energy. which also helped me in doing a lot of Researchs and I came to know about so many new things.

I would like to thank staff and faculty of Department of Computer Science and Engineering, IIT Kanpur for providing the state-of-art facilities.

I would like thank my colleague Nikhil and friends Shrikant, Pankaj, Prashant, Sudhanshu for suggestions and ideas throughtout my thesis work.

Vilay Kandi

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Garbage Collector(GC) tries to claim those objects in memory which are no longer used in the program. The basic work of GC is to find out which objects are not in use and reclaim the resources used by those objects. This frees the programmer from handling object allocation and deallocation manually. GC was first design by John McCarthy [1] to solve problems in Lisp. Now-a-days GCs are being used by most programming languages like Java, Python, Perl etc.

The most common technique used in GC is reachability tracing. A GC collects only those objects on heap which are either not live or not reachable through live variables [2]. An object is said to be reachable if it is a root variable or there is reference to the object from a root set of variables. Some objects are not going to be used in the program even though they have a reference from the root set of variables either directly or through a chain of reachable objects. To remove those objects on heap, liveness of object should be checked instead of reachability [3] [4] [5]. We will discuss about such an example in subsequent section.

1.1 Objective

To make objects live on heap based on liveness instead of reachability, root set of objects with their liveness information needs to be passed from compile time to runtime. Steps to achieve the required goal are as follows.

- Make a design so that we can pass particular root set of objects at run time.
- Dump liveness information in serialize file.
- Access both the information at runtime for better performance.

1.2 Motivational Example

To understand the need of GC based on liveness instead of reachability, look at the example shown in Fig 1.1, which contains a Java program, liveness information of object x in the form of automaton and structure of object x on heap at line number 3.

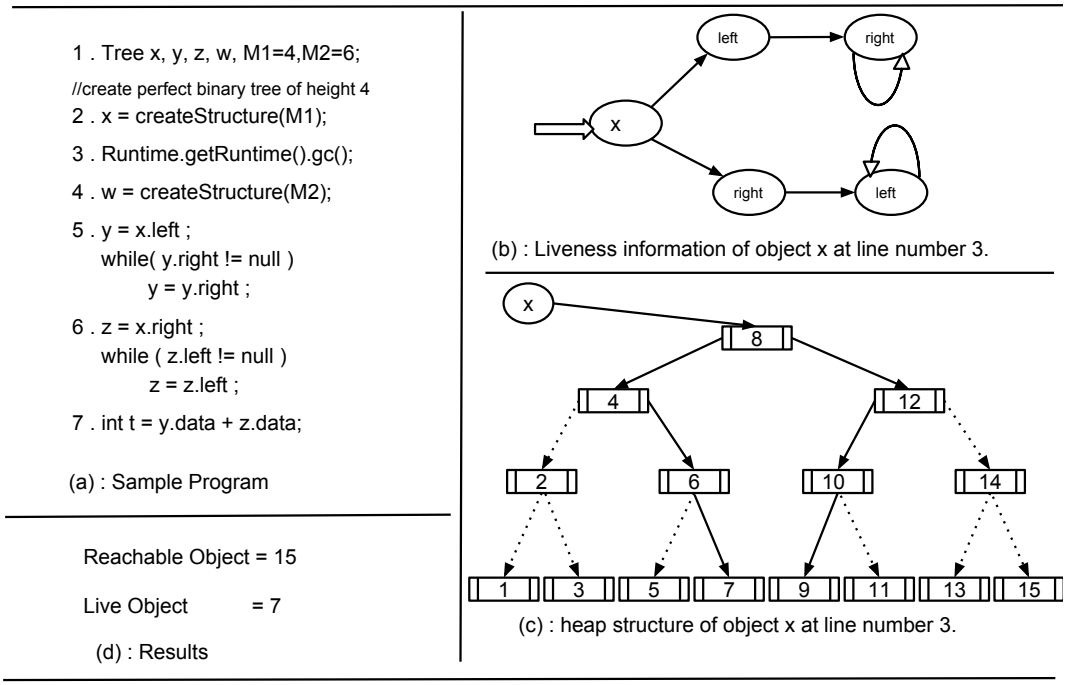

Figure 1.1: Motivational Example

The Program in Fig 1.1(a) defines objects at line number 1. At line number 2 it calls method createStructure(4), which creates perfect binary tree of height 4 having data value as their inorder traversal number and returns address of root node. Line 5 and 6 are code for finding predecessor and successor respectively. If we look at line number 3 in the program, the only live variable present is x. The behavior of

x on heap as a finite automaton at runtime in the form of liveness information is shown in Fig 1.1(b). If we apply given automaton on a heap structure of object x and traverse through it by marking reference edges as solid, it gives heap structure of x as shown in Fig 1.1(c). Looking at the heap structure mentioned in Fig 1.1(c) we can notice the following things:

- Object reachable through solid line are live.
- Object reachable through dashed line are reachable but not live, these are garbage and can be collected.

GC based on liveness rather than reachability can collect such objects. Taking into account the above scenario, we can collect more number of garbage objects.

1.3 Contribution

Our contribution to this thesis can be described as follows:

- Passing information from compile time to runtime.
  We have made required changes in Soot [6] and JikesRVM [7] so that we can transfer object information through newly created instruction.
- Combining both the objects and its liveness information in the form of automaton at runtime which improve garbage collection.

1.4 Outline of Thesis

The rest of thesis is organized as follows. Chapter 2 gives background details such as tools used, related work, approach towards solution etc. Chapter 3 discusses about design and algorithms used for counting reachable as well as live objects. Chapter 4 gives details about changes made at compile time. Chapter 5 describes changes made at runtime. Chapter 6 presents the experimental results. Chapter 7 gives conclusion and future work.

Chapter 2

Background

This chapter gives the background details of terminology, infrastructure and related work to place the research contribution of this thesis in context. Section 2.1 begins with terminology used in this thesis. Section 2.2 describes Soot framework and JikesRVM as an infrastructure tools used in thesis. Section 2.3 explores about the explicit live reference analysis, which was used as a black box.

2.1 Terminology

Heap is a region of memory dynamically allocated to program at runtime. Blocks of memory allocated on heap are referred as *objects*. Reclaiming those object is called *collection*. Reclaiming such unused objects is known as *garbage collection(GC)*. Process which does this job is called *garbage collector*. Working thread, which does the process of allocation and deallocation is known as *mutator*. GC which checks liveness of all object on heap called as *full heap collector*. Current types of GC can be found out by its *plan*. The plan gives details of GC such as mutator, collector, constraints, tracing policy etc. Objects having reference from the root set of objects are called as *reachable objects*. Now onwards *live objects* on heap are only those objects which are used in program. References of such live objects should not be considered as live unless it is used in further execution of program. Java bytecode is referred as bytecode only. Tracing of objects based on reachability is also called

as *transitive closure tracing*. Garbage collector based on reachability is referred as *RGC*. Garbage collector based on liveness is referred as *LGC*. If we talked about some program then it is *Java program* only unless it is explicitly mention. Java provides a mechanism, called object serialization where an object can be represented as sequence of byte that includes the object's data as well information about the type and the type of data store in object [8].

2.2 Frameworks Used

We use the following frameworks for our thesis :

- Soot[6] as a compile time framework to generate class file.
- JikesRVM[7] as a runtime framework to run this class file.

2.2.1 Soot framework

Our work is done in context with Soot optimizing framework developed by Sable Research Group[6]. Soot is an extensible tool with a powerful API, which allows users to write high level program analysis and transformation easily [9]. Soot provides tree intermediate representations for Java namely, *Jimple, Baf, Jasmin* [10]. These intermediate representations allow different APIs for various optimization techniques. In this project we are going to extend the basic bytecode instruction set to add a new instruction *objref* . For this we have to extend existing Soot IR to incorporate this new instruction. Soot version used in this work is soot-2.5.0.

Jimple is 3-address intermediate representation that has been design to simplify analysis and transformation of java bytecode [11]. **Baf** [10] is composed of low level instructions similar to Java bytecode. **Jasmin** [10] is a Java assembler interface. It takes ASCII descriptions for Java classes and converts them into binary Java class files suitable for loading into a Java interpreter.

Architecture

Soot is an optimization framework which internally goes through several transformation of intermediate representations(IR). The flow of internal transformation is shown in Fig 2.1[10].

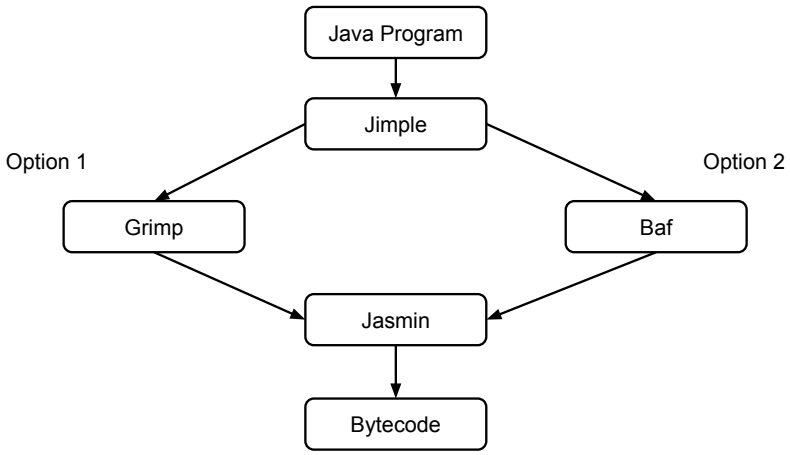

Figure 2.1: Soot Framework

It takes java program as input and first converts it to Jimple IR which can be optimized in several ways and later converts it to bytecode. Transformation from Jimple to bytecode can be done in two different ways. One is through $Jimple \rightarrow Grimp \rightarrow Jasmin \rightarrow byteCode$ and another is $Jimple \rightarrow Baf \rightarrow Jasmin \rightarrow byteCode$. Even though both the ways are equivalent, in this work second approach has been explored due to it's simplicity and ease of use.

2.2.2 JikesRVM

Architecture of JikesRVM can be divided into Core Runtime Services, Magic, Compilers, Memory Manager and Adaptive Optimization System [12]. We deal with two of its components, namely Compilers and Memory Manager. Compiler generates executable code from bytecode while Memory Manager take care of allocation and deallocation of objects.

Compilers

Compilers in *JikesRVM* are of three types(*baseline, optimizing, JNI*). These compiler generate executable machine code from bytecode. *Baseline compiler* generate code without optimizing bytecode. *Optimizing compiler* generate optimized bytecode through various analysis on bytecode like constant propogation, dead code elimination etc. Java Native Interface*JNI* compile native method by generating code. Out of these compiler we are using *baseline compiler* to generate executable machine code because of its simplicity and ease of use. We can select this compiler at the time of building JikesRVM. Default compiler used in JikesRVM is baseline compiler.

Memory Manager

Memory Manager handles the object allocation and deallocation. Memory manager class is located at org.jikesrvm.mm.mminterface.MemoryManager in JikesRVM. It provide mutator object for a current plan of garbage collector.

Object Model

Java object is composed of four pieces, JavaHeader, GCHeader, MiscHeader and instance fields. Every object contains these portions. JavaHeader support language level functions such as locking, hashcodes etc as shown in Fig 2.2.

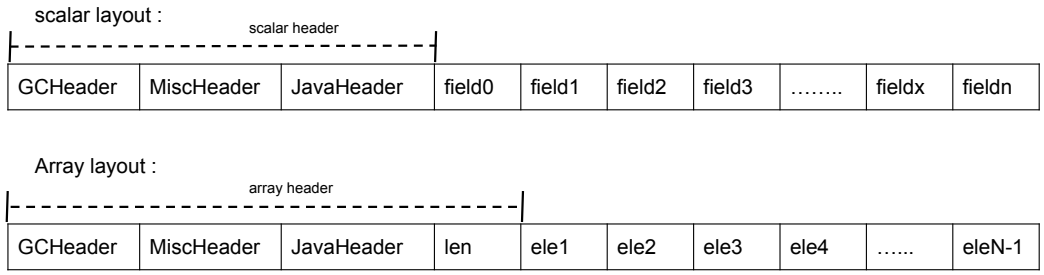

Figure 2.2: Object Model

Object Header

The default object model uses a two word header. One word holds TIB pointer and other one is shown in Fig 2.3.

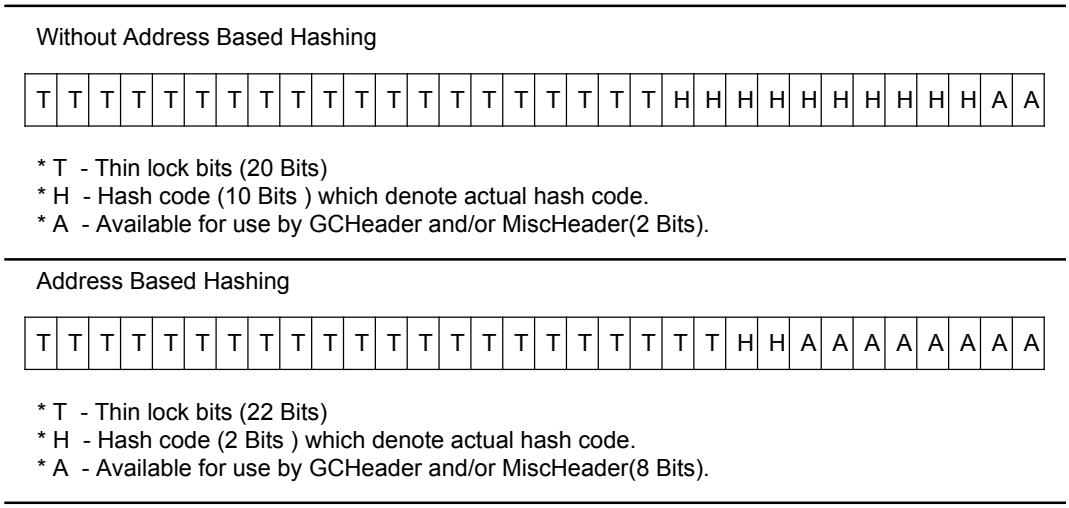

Figure 2.3: Java Header

The word contains an inline thin lock, hash code and available bits. If Address based hashing is used 2 hash bit denote 3 state of object *(Hashed, UnHashed, Hashed_and_Moved)* to uniquely identify objects. In case of without address based hashing all 10 bits are used for hash code. Available bits are used for GCHeader and/or MiscHeader. Default model used in JikesRVM is address based hashing. Users can change this hashing technique while building JikesRVM. Address based hashing is used in our approach, as more number of bits are available to use. These bits can be used to mark objects for counting, reachability check, liveness check etc.

2.3 Explicit Live Reference Analysis (Heap Reference Analysis)

This is end-to-end solution for discovering live references of an object at compile time [3][page 1:5]. It is an intra-procedural analyzer which predicts the inter-procedural flow of data value. Data flow values represent the sets of automatons for all live

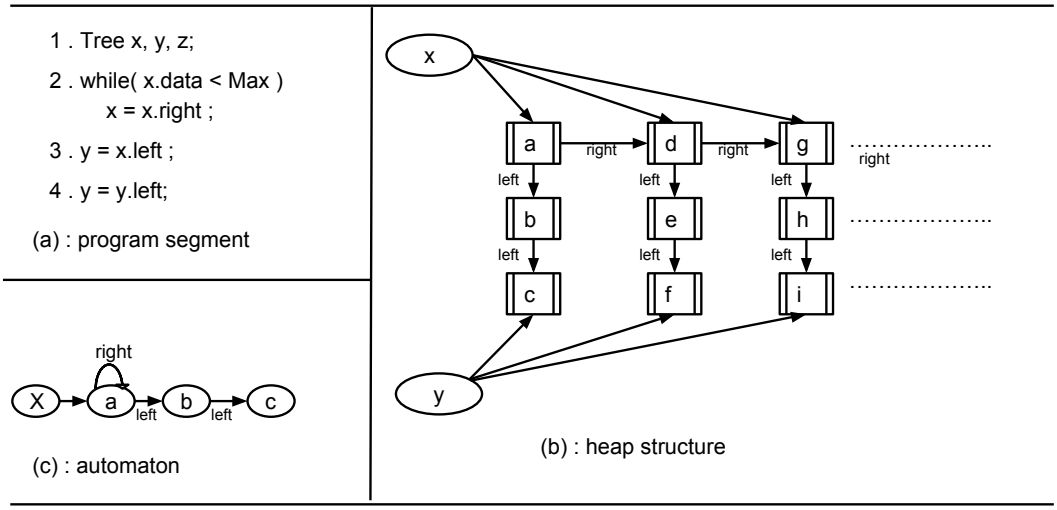

Figure 2.4: Discovering explicit live references of an object at compile time

variable at each program point. To understand how this analysis calculates live references of objects, look at the Fig 2.4. It contains code segment (Fig 2.4(a)) and behavior of object x on heap (Fig 2.4(b)).

Looking at program segment, we can examine that x may point to objects *a, d, g* at runtime depending on how many times while loop executes. After execution of while loop, two left references of an object are accessed as shown in Fig 2.4(b). If we try to express such behavior of an object with the help of an automaton then we will get following kind of access expressions:

$$
\begin{aligned}
&[x \rightarrow left \rightarrow left] \\
&[x \rightarrow right \rightarrow left \rightarrow left] \\
&[x \rightarrow right \rightarrow right \rightarrow left \rightarrow left] \\
&[x \rightarrow right \rightarrow right \rightarrow right \rightarrow left \rightarrow left] \\
&..... \\
&..... \\
&.....
\end{aligned}
\tag{2.1}
$$

If we represent this information in the form of regular expression [13], then it

looks like

$$x \rightarrow (right)* \rightarrow left \rightarrow left$$

Finite automaton to represent above regular expression is shown in Fig 2.4(c). This is the output of explicit live reference analysis for all objects in the program at each program point. How to know liveness of program variable at compile time can be found at [14] [13].

Chapter 3

Design

This chapter gives design of our complete application and describes algorithms used in it. Design model is shown in Fig 3.1.

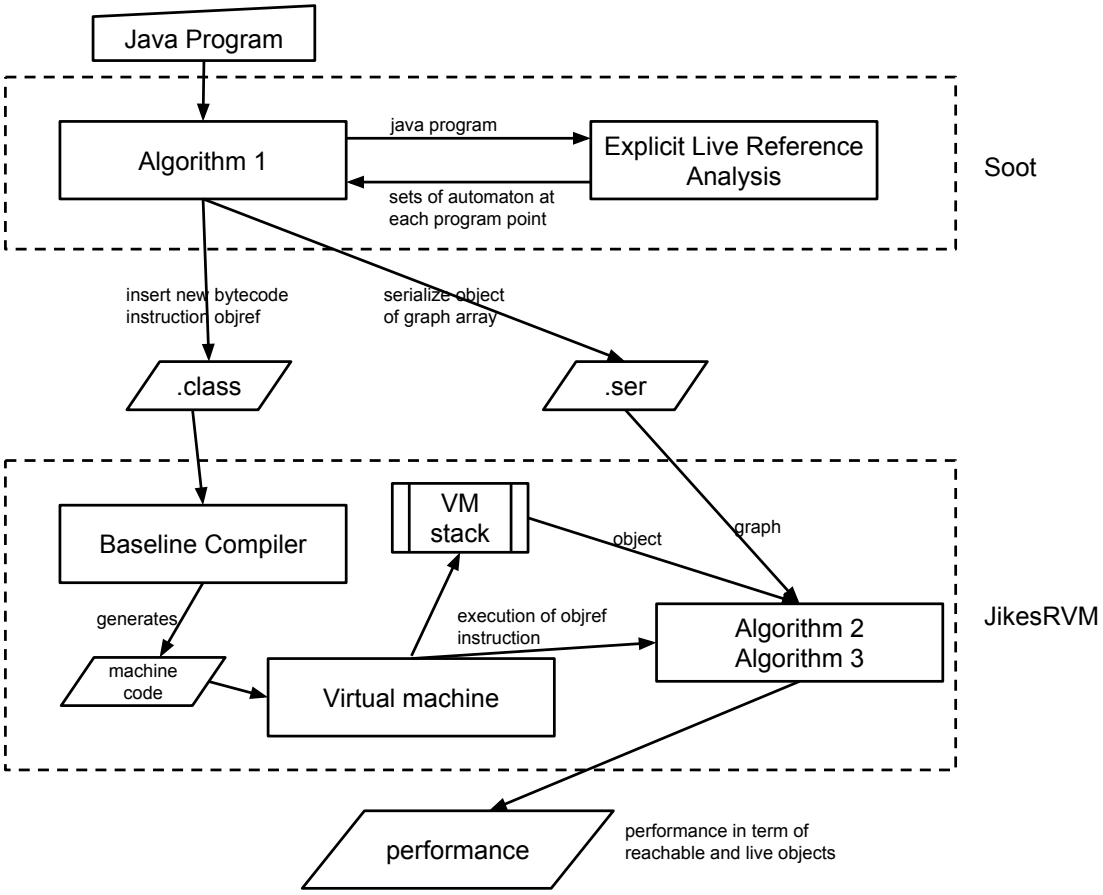

Figure 3.1: Design

Our application takes Java program as an input and gives performance as an

output in terms of number of live objects on heap against number of reachable object. We have implemented this in Soot framework for the transformation of bytecode as shown in Algorithm 1. It transforms bytecode by inserting *objref* instruction and dumps liveness information in serialized file. It uses Explicit Live Reference Analysis as a black box from earlier work [3], which gives the behavior of object on heap in the form of an automaton. *JikesRVM* takes class file as a input generated by Soot. Components of JikesRVM are shown in Fig 3.1. Changes are made in JikesRVM components to interpret the new bytecode instruction inserted in class file. Generated machine code *(including machine code for the new instruction objref)* from baseline compiler gets executed on virtual machine. Execution of *objref* calls a method in virtual machine which takes an object as an argument. This method accesses the liveness information from serialize file and gives performance as output.

3.1 Generation of class file and serialized file of graph array

This transformation analysis runs in whole program mode [15] for transformation of bytecode. Transformation of bytecode is performed under scene transformer [10]. Output of Explicit Live Reference Analysis gives data flow values at each unit statement. These data flow values are sets of automatons for each live variables representing their behavior on heap [3]. The program finds GC call and replaces it with new *objref* V_i statement for all live variable V_i. This statement gets inserted into class file through various intermediate representations shown in Table 4.1. Chapter 4 gives details of this new instruction and its transformation through various intermediate representations. This instruction passes the root set of variables from compile time to runtime. However data flow values of analysis are dumped in serialized file. Name of these files are derived from line numbers of GC call in source code as a unique value. This information is accessed in JikesRVM. Next section explains about processing of this information.

Algorithm 1: Generate appropriate .class file

Input : Java file as input
Output: .class file containing appropriate objref instruction and .ser file containing liveness information of object.

```
1  ela = output of Explicit Live Reference Analysis ;
2  for each method M in java file do
3      for each statement S in method M do
4          if S = Runtime.getRuntime().gc() then
5              //get data flow values before statement S. graphset =
               ela.getFlowBefore(S)
6              for all live variable V_i at program point S do
7                  ObjectRefStmt a1 = Jimple.v().newObjectRefStmt(V_i);
8                  units.insertBefore(a1, S);
9                  graphArray.add(graphset.get(V_i))
10             end
11             //dump graphArray into LineNO.ser file
               file.writeObject(graphArray);
12         end
13     end
14 end
```

3.2 Implementing LGC

We have a .class file and a .ser file as an output of Soot framework. These files are fed as an input to JikesRVM. Figure 3.1 shows how these files are processed. In this section we discuss tracing of an object using Algorithm 2 and Algorithm 3. Most of the GCs today trace the object using transitive closure, which is described in Algorithm 2. A LGC traces the object with the help of graph as a liveness information. It is described in Algorithm 3. As we discussed in the Introduction, with the help of an example, that Algorithm 3 marks less number of object as live compared to Normal tracing algorithm. As a result we can optimize heap memory more efficiently. Performance of GC is measured in terms of number of live objects in both these algorithms.

Algorithm 2: Trace objects using transitive closure

Input : Root set of objects

Output: objects marked based on transitive closure

```
1  for all elements E in root set do
2      obj = E.getObject();
3      queue.insert(obj);
4  end
5  while ( !queue.isEmpty() ) do
6      obj = queue.remove();
7      mark(obj);
8      fields = obj.getAllReference();
9      for f in fields do
10         child = f.load();
11         if (child != null) and (testMarkBit(child) != true) then
12             queue.insert(child);
13         end
14     end
15 end
```

For implementation of this algorithm we have used 3 available bits of object model shown in Fig 2.2. Out of the 8 bits available in addressed based hashing, last 3 bits are used because first few bits are used by GC for internal purpose. Last bit is used for marking liveness of object. Next two bits are used to count reachable and live objects with the help of standard DFS algorithm [16]. Algorithm 3 uses three types of values in queue for marking objects as live. First one represent object

itself. Second and third represent graph and current root node respectively.

Algorithm 3: Trace objects using graph.

Input : Root set of objects and their Graphs

Output: object marked based on graph

```
1  for all element E in root set do
2      object = E.getObject();
3      graph = E.getGraph();
4      currRoot = graph.getRoot();
5      queue.insert(pair¡object,pair¡graph,currRoot¿¿);
6  end
7  while ( !queue.isEmpty() ) do
8      element = queue.remove();
9      object = element.first;
10     mark(object)
11     graph = element.second.first;
12     currRoot = element.second.second;
13     fields = object.getAllReference();
14     targets = graph.getTargets(currRoot);
15     for f in fields do
16         for t in targets do
17             if f==t then
18                 child = f.load();
19                 if (child != null) and (testMarkBit(child) != true) then
20                     queue.insert(Pair¡child,Pair¡graph,t¿¿)
21                 end
22             end
23         end
24     end
25 end
```

Chapter 4

Implementation

This chapter explores the implementation details and issue related with it. Section 4.1 describes changes required at compile time to generate class file. Section 4.2 discuss implementation issue related with JikesRVM to run generated class file.

4.1 Changes at compile time

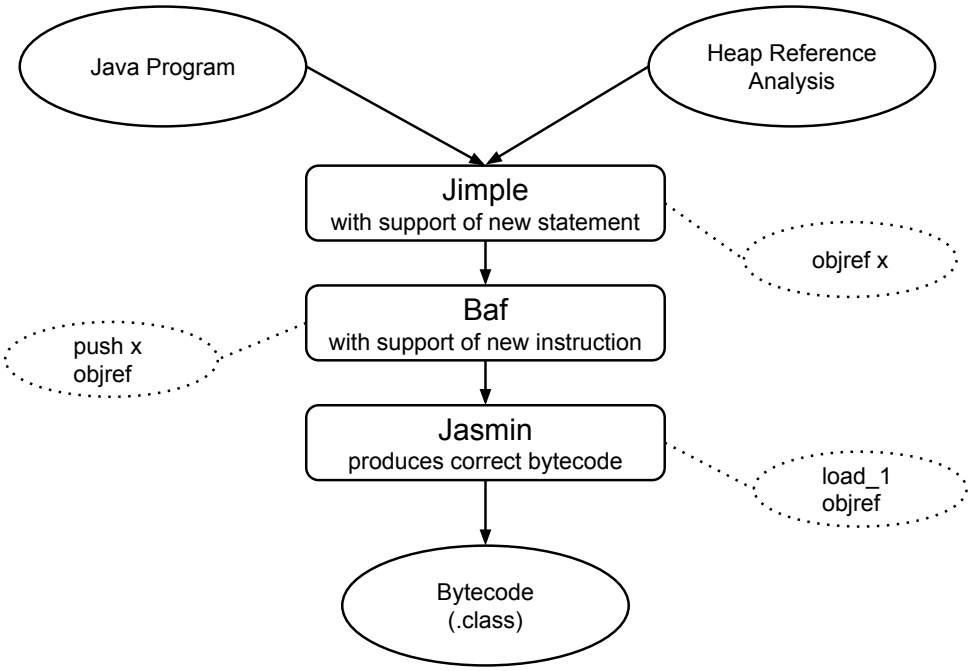

Figure 4.1: Soot Changes

We are using Soot as an compile time framework. As discussed earlier, passing

information from compile time to runtime is one of the challenges. In order to resolve it, the object must be passed from compile time to runtime. We created new instruction to pass this object. Intermediate representation of Soot framework needs to be modify to parse this instruction. Modification required in intermediate representations are shown in Fig 4.1 [10]. These changes are discussed in following subsections.

4.1.1 Modification in Baf

As new statement is created in the grammar of Jimple IR, there is a need for support in Baf IR. Therefore, same kind of changes are required in Baf IR. It requires addition of two new functions, one for creating Baf instruction and other as an interface between Baf and Jasmin Assembler as shown in code below.

```
-----------------------------------------------------------------
//support for adding new instruction in Baf
public ObjectRefInst newObjectRefInst(Type opType, Local l) {
        return new BObjectRefInst(opType,l);
}
//Interface between Baf and Jasmin
public void caseObjectRefInst(ObjectRefInst i) {
        emit("objref");
}
-----------------------------------------------------------------
```

4.1.2 Modification in Jasmin

Like Jimple and Baf, Jasmin also needs to be modified to support the new instruction. The modifications are shown in the code below. A new opcode opc_objref with number 203 is generated. Numbers upto 202 are already reserved. Information of new opcode is added to runtime constants of Jasmin using function addInfo of

InsnInfo.java. Then new instance of this instruction is created and added to the existing Jasmin code by the funtion addInsn in ClassFile.java.

```
-----------------------------------------------------------------
//Add Runtime constant in Jasmin. Add its opcode name and length.
public static final int opc_objref = 203;
//Add information about new opcode in InsnInfo.java
addInfo("objref",RuntimeConstants.opc_objref,"objref");
//Generate corresponding code in ClassFile.java
Insn inst = new Insn(RuntimeConstants.opc_objref);
_getCode().addInsn(inst);
-----------------------------------------------------------------
```

4.1.3 Adding new instruction to Soot

This section explains about adding new instruction in Soot through analysis. Code to add statement "*objref* local" in Jimple, before and after unit statement is as follows.

```
-----------------------------------------------------
//Adding new Statement in SOOT through Analysis
ObjectRefStmt o = Jimple.v().newObjectRefStmt(local);
//Adding Instruction before and After unit statement.
units.insertBefore(o, unit);
units.insertAfter(o, unit);
-----------------------------------------------------
```

After adding this instruction in soot, its insertion in different intermediate representations is shown in Table 4.1.

Jimple	Baf	Jasmin	Bytecode
objref local	load.r args *objref* .r	load_0 *objref*	aload_0 *objref*

Table 4.1: Instruction in different intermediate representation

4.1.4 Correctness of bytecode

Java Bytecode Editor(JBE) [17] is a tool for reading and verifying bytecode. Bytecode is modified to contain the new bytecode instruction. JBE is used to verify bytecode. JBE is standerd Java byteCode reader, so similar changes are required in JBE to interpret newly created instruction.

4.2 Changes at runtime

As we have made changes to Soot which is a compile time framework, similar changes are required at runtime so that the modified class file will be executed on virtual machine. To accomplish this task JikesRVM [7] is used as a runtime Virtual Machine(RVM).

4.2.1 Machine code generation

Baseline Compiler

JikesRVM takes *.class* file as input, which contains bytecode and converts it to executable machine code through two types of compilers :*(baseline, optimizing)*. Machine code executes on VM. In this approach we explored baseline compiler because of its simplicity and ease of use. Most of the code in **baseline compiler** is machine dependent. It contains *BaselineCompiler, TemplateCompilerFramework, BaselineCompilerImpl* as main files [7]. Baseline compiler does code generation process. As we get new bytecode instruction, baseline compiler will execute the following code to generate executable machine code.

/**

```
*  LiveGC Extension : emit machine code in baseline compiler
*  1 . copy parameter from operand stack to register.
*  2 . generate call to absolute address in entrypoint.
*/
protected final void emit_objref() {
  // pass 1 parameter word
  genParameterRegisterLoad(asm, 1);
  //offset of method in VM which will be call by this instruction.
  Offset methodOffset = Entrypoints.objrefMethod.getOffset();
  //absolute address of method.
  Offset absOffset = Magic.getTocPointer().plus(methodOffset);
  asm.emitCALL_Abs(absOffset);
}
```

Stack Manipulation

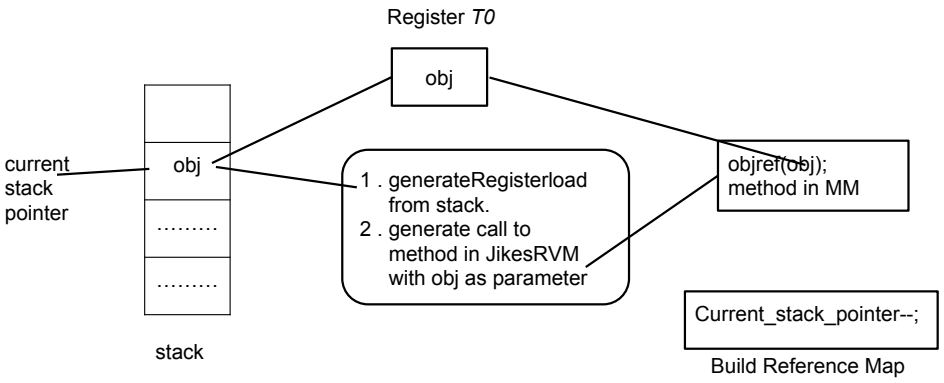

Figure 4.2: Stack Manipulation

Originally, statement inserted in Jimple of Soot framework is *objref* X, and it's bytecode form in class file is *load_0 : objref*. We have two machine instructions after code generation process by baseline compiler as shown in code above. First instruction generates register load for object and second instruction calls method in JikesRVM by passing argument loaded in register. There is a reference available on

stack for the object passed. To remove this reference there is a need to change the reference map, located at *BuildReferenceMap* file. Details of stack manipulation can be seen in Fig 4.2.

4.2.2 Support for newly created instruction

To support a class file containing extra instructions, JikesRVM has to be modified. Modules with their corresponding changes are described below.

Bytecode Constant

Bytecode constant is the information about Java bytecode that appear in the "code" attribute of a .class file. A new unused bytecode information has to be inserted corresponding to bytecode instruction *objref* .

$$\begin{aligned} &int \;\; JBC\_objref \;\; = 203; \\ &int \;\; JBC\_length[203] = 1; \\ &String \;\; JBC\_name[203] = "objref" \end{aligned} \tag{4.1}$$

BuildReferenceMap

It works with baseline compiler in calculating the GC maps for local variables and the java operand stack. These GC maps abstractly finds which stack slots and local variable contain references at the start of bytecode. When *objref* instruction is parsed in BuildReferenceMap, it will decrement stack top pointer and skip the instruction.

Entry Point

Entry Point represent fields and methods of the virtual machine that are needed by compiler-generated machine code or C runtime code. In this module we declared the method which is going to be called by code generated through baseline compiler.

Runtime Entry Point

These "helper functions" are called from machine code emitted by BaselineCompilerImpl. They contain definitions of fields and methods declared in Entry Point. We define our method named "*objref* " with argument as object in this module, which eventually is the method in memory manager.

Memory Manager

At this location we have objects which are passed from compile time. Next, we access the corresponding graph file which was dumped at compile time.

Chapter 5

Experimentation

In order to show the effectiveness of LCG, we have calculated the number of objects live on heap with two different algorithms. One is by marking objects and their references based on transitive closure. Other is by marking objects and their references based on automaton as a liveness information.

5.1 Experimental methodology and measurements

LGC collects more number of garbage objects, which gives the evidence of improvement over RGC. Experiments were performed on $Intel^{(R)}core^{TM}i7-4770\ \ CPU@3.40GHz\times$ 8 processor with *15.3 GiB* of memory and *64-bit 14.04 LTS* Ubuntu operating system. Benchmarks used were BiSort, Loop, DLoop, Reverse, TreeAdd and BST. Performance of these benchmarks is shown with the help of graphs. X-axis in the graphs represent instance of program point. Instance of program points are those points where explicit call of GC is triggered. These explicit calls are triggered after some threshold amount of statement executions. Overall performance of benchmark programs are shown in Table 5.1.

BiSort

This is a class that represents values to be sorted by the BiSort algorithm. It performs a bitonic sort based upon the Bilardi and Nicolau algorithm [18]. BiSort

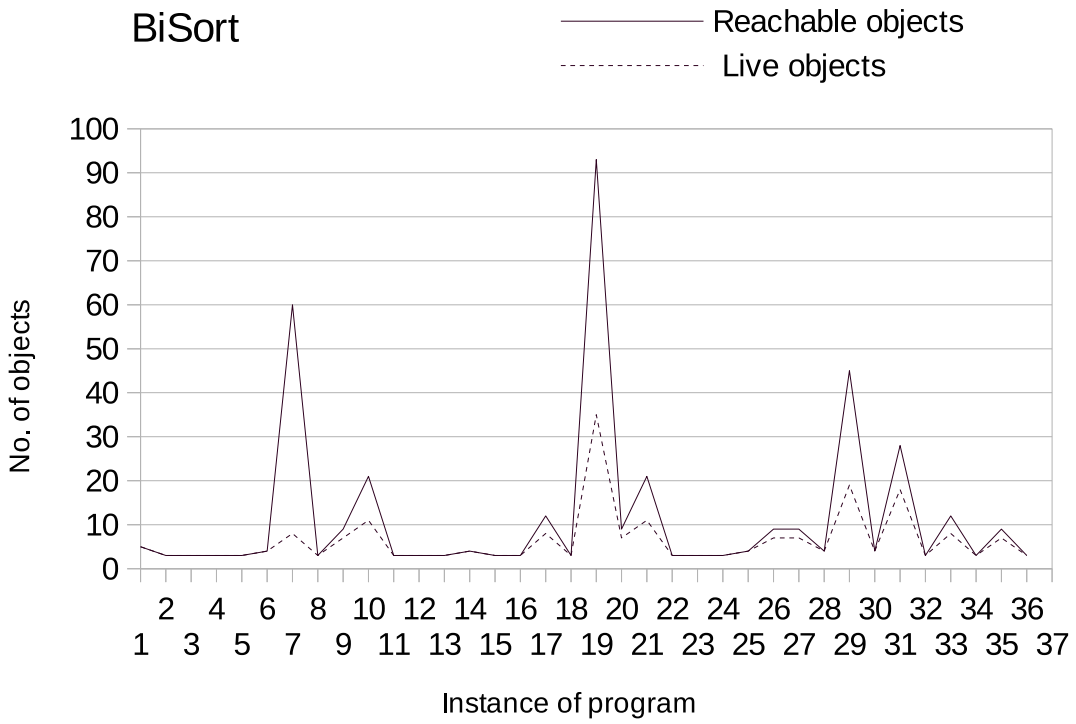

Figure 5.1: Bitonical Sort

algorithm was implemented in variety of architectures like shuffle-exchange [19], binary cube [20], the mesh [21] etc. In BiSort algorithm, initially it creates a tree structure of all such objects having bitonic sequence and then it sorts using bitonical sort. Performance of this algorithm with respect to number of live objects and reachable objects are shown in Fig 5.1.

Loop and DLoop

Loop and DLoop are the classes which represent single and double linked lists respectively. These programs are used in most of the architectures and application in recent times[22]. Loop and DLoop creates a linked list and traverses through it. Running these programs on our application will give optimal solution for the number of live objects on heap as shown in Fig 5.2 and Fig 5.3.

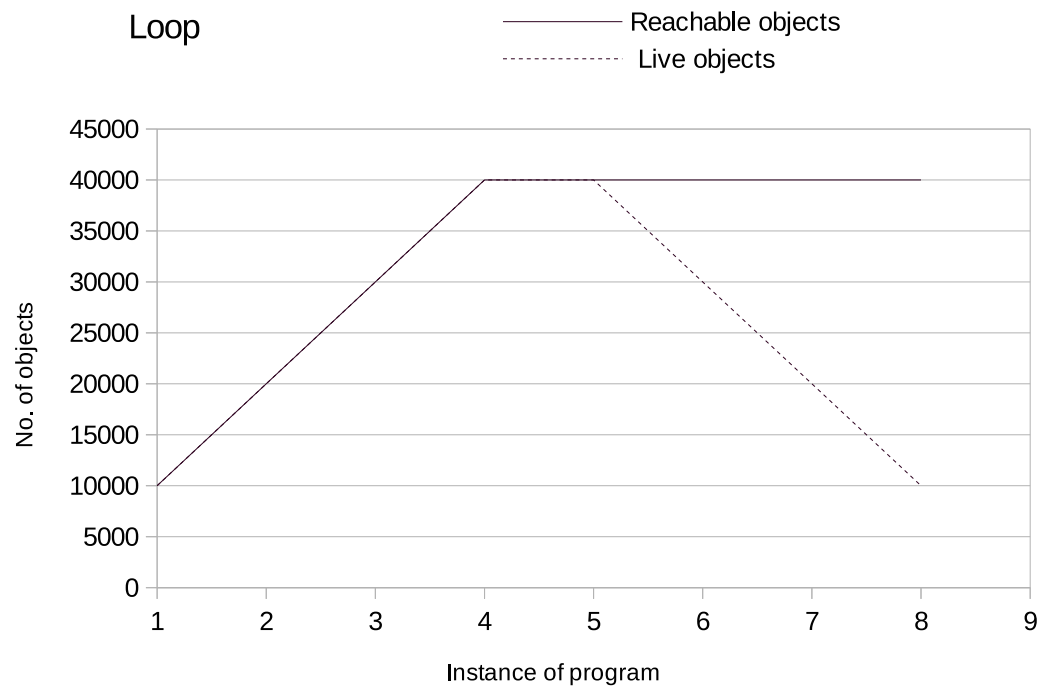

Figure 5.2: Loop

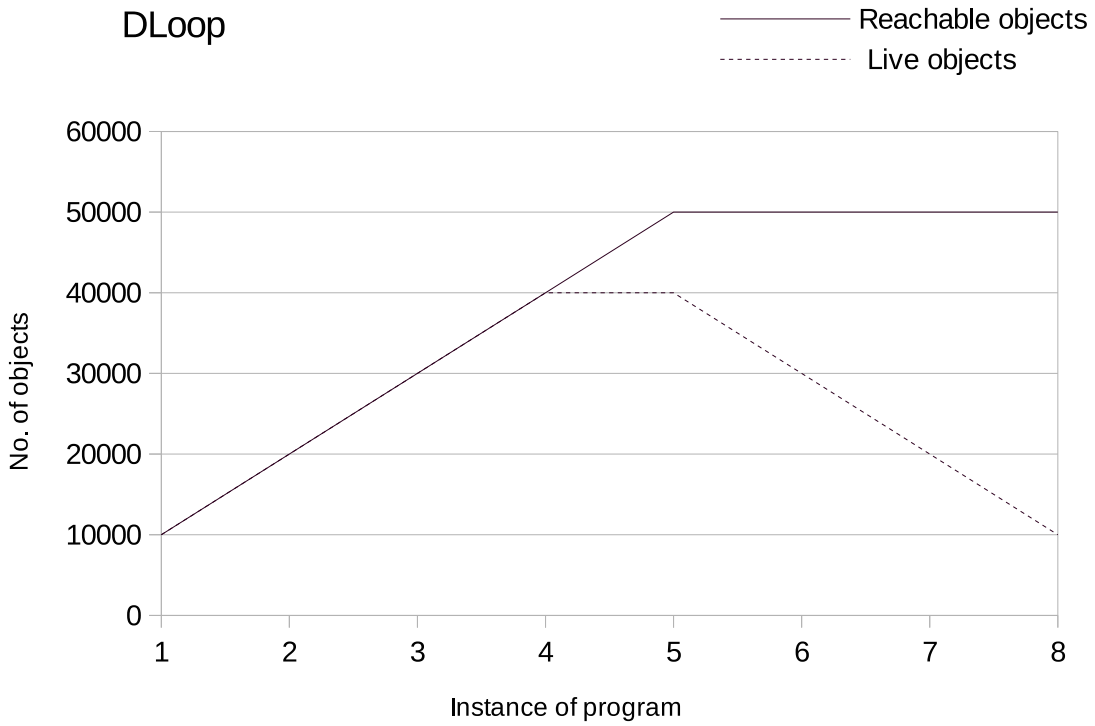

Figure 5.3: DLoop

Reverse, TreeAdd and BST

Reverse is a class representing linked list which creates a linked list and then reverse it. From graph shown in Fig 5.4, we can observe that after creation of list, RGC doesn't collect those objects which are not live. TreeAdd is program which contains a tree (as a data structure) and a function which will calculate total number of tree node. TreeAdd calculates left subtree first and then right subtree. Performance of LGC for Tree add is as shown in Fig 5.5.

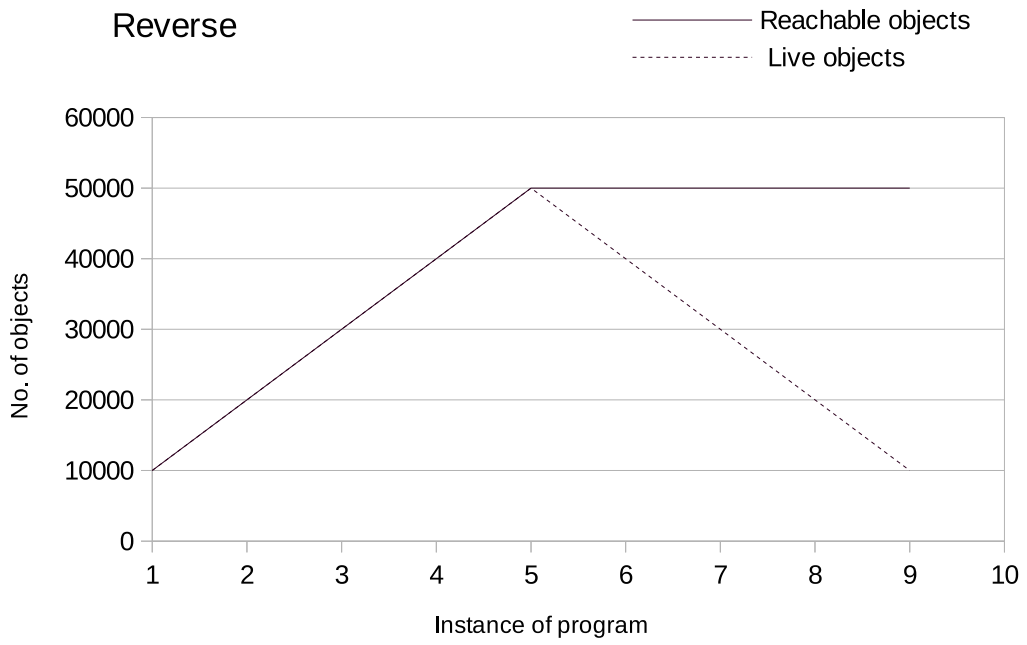

Figure 5.4: Reverse

BST is a class which represents tree structures on heap. It creates a binary tree and searches on it with the help of a recursive algorithm. Its performance with respect to memory is shown in Fig 5.6. From the figure, we can see that performance of heap structure is growing till the tree structure is being created. After that, we are not able to collect garbage from such cases as the whole tree structure is live. It also contains a recursive function and HRA analysis doesn't predict behavior of such objects on heap as discussed in section 2.3 .

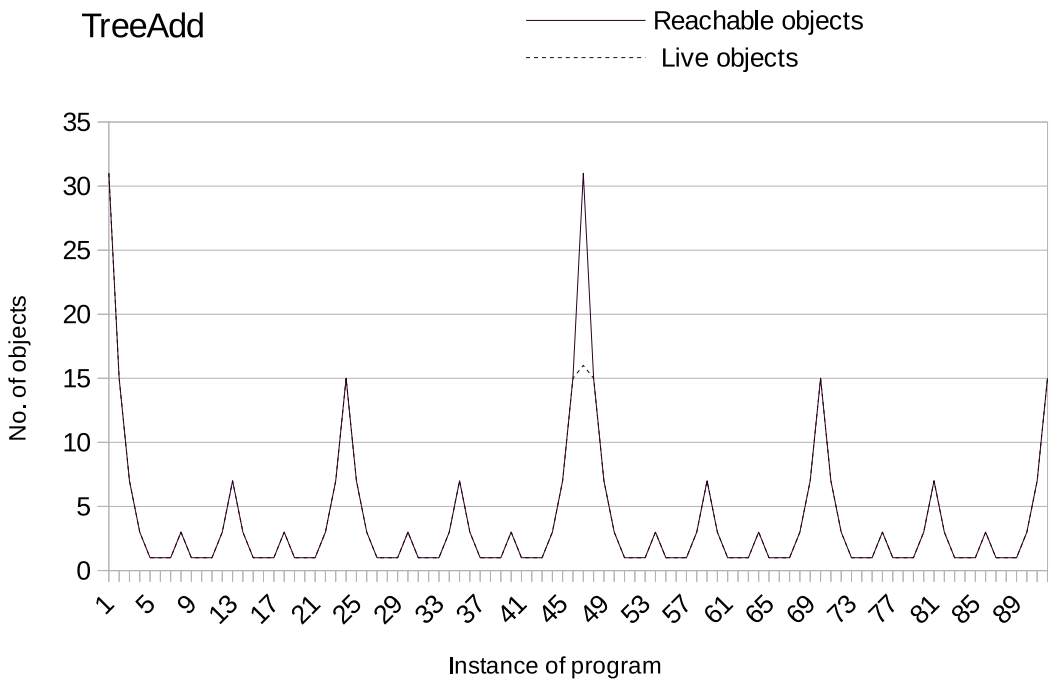

Figure 5.5: TreeAdd

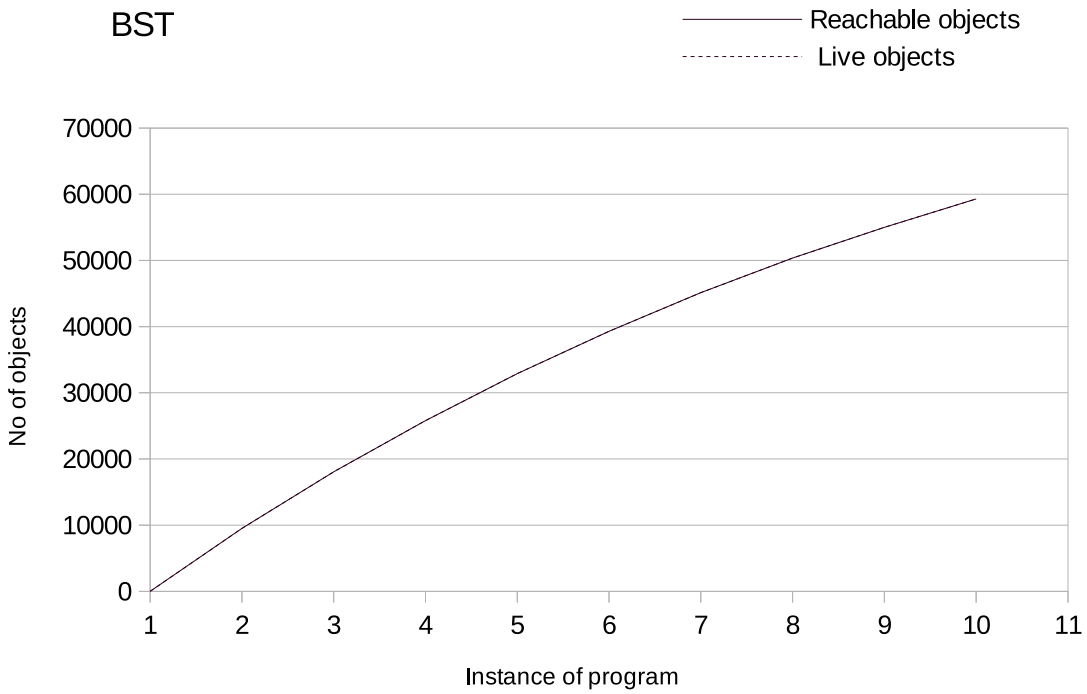

Figure 5.6: BST

5.2 Overall performance of program

We now calculate performance of the complete program instead of taking snapshots at each point. Performance of complete program is calculated in terms of observation values and percentage of extra garbage that can be collected by LGC. To analyze it we have set a count variable for statements in each benchmark program for which there is a change in memory. If count variable reaches some threshold value then we are taking a snapshot of all reachable objects and live objects on heap. Such points where snapshot is taken are referred as program points. At each program point, snapshot values are taken as reachable count and live count for RGC and LGC respectively.

Average number of objects present on heap for RGC and LGC having n program points is calculated by equation 5.1

$$\begin{aligned} RGC_{avg} &= \frac{\sum_{i=1}^{n} reachable\_count_i}{n} \\ LGC_{avg} &= \frac{\sum_{i=1}^{n} live\_count_i}{n} \end{aligned} \tag{5.1}$$

Percentage of extra garbage present for a program(P) at a point(i) is calculated as follows:

$$G_{P_i} = \frac{reachable\_count_i - live\_count_i}{reachable\_count_i} \times 100$$

Average of percentage of extra garbage collected for complete program having n program points is:

$$G_P = \frac{\sum_{i=1}^{n} G_{P_i}}{n}$$

Computation of all these values for benchmark programs described earlier are shown in Table 5.1.

We have taken running times for RGC and LGC which are based on tracing algorithms discussed in Algorithm 2 and Algorithm 3. The running times for different

SrNo	Program	Avg number of objects on heap		Time(in ms)		Percentage of extra garbage
		RGC	LGC	RGC	LGC	
1	BiSort	11.47	6.36	0.2	31	14.32
2	Loop	32500	25000	46	1653	18.75
3	DLoop	37999	25000	77	1666	24.99
4	Reverse	38889	27778	88	1837	22.22
5	TreeAdd	3.87	3.7	2	63	1.02
6	BST	33534	33534	94	7832	0

Table 5.1: Average performance of LGC over complete program

benchmarks are shown in Table 5.1. It shows that, we have achieved more garbage collection from the program at the expense of time. We can improve the running time of LGC using graph reachability which is discussed in the accompanying thesis [23].

Chapter 6

Conclusion and Future Work

This work fulfills the basic objective of passing object references and the information about live references at runtime to trace the object based on liveness instead of reachability. This liveness information is useful to run garbage collector at runtime for better memory footprint. We have traced the objects at runtime with the liveness information provided in the form of automatons. The results are obtained based on count value of the number of objects present on heap for both RGC and LGC. Looking at the results shown in observation Table 5.1, we can say that LGC performs better or near about equal to RGC in all benchmarks. In LGC for the programs *Loop, DLoop and Reverse*, average increase in garbage collected(G_P) for complete program is close to 20%, which is a significant improvement against existing RGC. In worse case if all the objects and their references are live till the end of the program, then graph tracing algorithm traverses like an RGC, as shown in TreeAdd example Fig 5.5. We can say that the number of live objects are always less than or equal to number of reachable objects. Therefore LGC will always perform at least as much as RGC, if not better.

This thesis work shows that the results in terms of number of objects that are live on heap instead of actual snapshot of memory after running the garbage collector. Current garbage collectors work with mutator program objects and objects created for internal purpose in VM. We don't have liveness information of such VM

objects because Explicit Live Reference Analysis runs at compile time. Therefore for collecting VM objects, GC traces objects based on transitive closure. Heap memory in JikesRVM is divided into spaces and the memory manager of JikesRVM allocates different objects in different spaces depending on the size of objects allocated. Each space has a different collection policy. We need a different policy for VM objects and mutator program objects. To run graph based tracing algorithm, objects in program have to be allocated in a separate single space. LGC can be run with a single space by changing allocation and deallocation policy of VM. Immediate future work would be to create a separate VM space for mutator programs which uses liveness based tracing for its garbage collection policy. We have dumped liveness information in the serialized file and accessed it at runtime. In future it can be dumped to class file itself and appropriate changes in VM can be made.

Bibliography

[1] John McCarthy. History of programming language. In *History of programming language*, pages 1–2. ISBN 0-12-745040-8, 1981.

[2] Amey Karkare, Amitabha Sanyal, and Uday Khedker. Heap reference analysis for functional programs. *arXiv preprint arXiv:0710.1482*, 2007.

[3] Uday P Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1):1, 2007.

[4] Amey Karkare, Uday Khedker, and Amitabha Sanyal. Liveness of heap data for functional programs. *arXiv preprint cs/0703155*, 2007.

[5] Rahul Asati, Amitabha Sanyal, Amey Karkare, and Alan Mycroft. Liveness-based garbage collection.

[6] Soot - a java optimazation framework. `http://www.sable.mcgill.ca/soot/`.

[7] Jikesrvm. `http://www.jikesrvm.org/`.

[8] Java - serialization. `http://www.tutorialspoint.com/java/java_serialization.htm`.

[9] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing java using attributes. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 8. IBM Press, 2000.

[10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, pages 35,36. IBM Press, 1999.

[11] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[12] Jikesrvm architecture. `http://jikesrvm.org/Architecture/`.

[13] Alfred V Aho and Jeffrey D Ullman. *Principles of compiler design*. Addision-Wesley Pub. Co., 1977.

[14] Alfred V Aho and Jeffrey D Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., 1972.

[15] Soot - a java optimazation framework - usage. `http://www.sable.mcgill.ca/soot/tutorial/usage/`.

[16] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[17] Java bytecode editor. `http://cs.ioc.ee/~ando/jbe/`.

[18] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal on Computing*, 18(2):216–228, 1989.

[19] Harold S Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, 20(2):153–161, 1971.

[20] Marshall C Pease. The indirect binary n-cube microprocessor array. *Computers, IEEE Transactions on*, 100(5):458–473, 1977.

[21] Clark D Thompson and Hsiang Tsung Kung. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, 20(4):263–271, 1977.

[22] Ran Shaham, Eran Yahav, Elliot K Kolodner, and Mooly Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. *Science of Computer Programming*, 58(1):264–289, 2005.

[23] Nikhil Pangarkar. Improving liveness based garbage collection in java. Master's thesis, IIT Kanpur, 2014.