SOS: Save our Object Space

Student Name: Aniya Aggarwal

IIIT-D-MTech-CS-MT-12-034
Nov 25, 2014

Indraprastha Institute of Information Technology
New Delhi

Thesis Committee
Rahul Purandare (Advisor)
Vinayak Naik (Internal Examiner)
Mohan Dhawan (External Examiner)

Submitted in partial fulfillment of the requirements
for the Degree of M.Tech. in Computer Science,
with specialization in Data Engineering

Keywords: Android, Escape Analysis, Memory Optimization, Garbage Collection, Loops, Heap, Static analysis

Certificate

This is to certify that the thesis titled **"SOS: Save our Object Space"** submitted by **Aniya Aggarwal** for the partial fulfillment of the requirements for the degree of *Master of Technology* in *Computer Science & Engineering* is a record of the bonafide work carried out by her under my guidance and supervision in the Information Management and Data Analytics group at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Rahul Purandare

Indraprastha Institute of Information Technology, New Delhi

Abstract

Each Android application (app) runs in its own virtual machine (VM), with every VM allocated a limited heap size for creating new objects. The heap size is scarce and device dependent. The more heap space an app uses, the more work the garbage collector (GC) would have; the more work the GC has, the bigger is the pause time for collection of un-referenced objects. To avoid frequent garbage collection, the objects should be allocated wisely. In this work, we propose a tool called *SOS* to help the developers to control and reuse memory allocated to objects on the heap. In this work, we target objects allocated in loops and identify them by leveraging static program analysis techniques. With the intention to reuse the heap space allocated to these objects, we further perform program transformation. As a case study, we take Android apps and manifest the benefits that SOS can provide in terms of reduction in pause times and reduction in heap space used. We show the trends in pause times, number of GC invocations and heap space freed, as a function of number of temporary objects.

Acknowledgments

I would like to thank Dr. Rahul Purandare for constantly motivating us and pushing us to try and delve further and further. I would like to sincerely thank Dr. Vinayak Naik for sharing his valuable insights about the Android platform and his valuable suggestions. I would also like to thank Steven Arzt for sharing his expertise in Soot. A special thanks to all my friends for quizzing me and spurring me on throughout the course of the thesis.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Android Activity Lifecycle 3

1.3 Android Best Practices 5

1.4 Our Contribution 6

2 Related Work **9**

2.1 Related Work 9

3 SOS Architecture **11**

3.1 Architecture 11

3.2 Why not AspectJ? 12

4 Our Approach **13**

4.1 Overview 13

4.2 Modules 15

4.2.1 Interprocedural Escape Analysis Module (IEAM) 16

4.2.2 Hoistable Sites Identifier Module (HSIM) 19

4.2.3 Bytecode Instrumentation Module (BIM) 21

4.3 Heuristics 25

5 Evaluation **26**

5.1 Evaluation Overview 26

5.2 Testbed 26

5.2.1 Application Description 26

5.2.2 Phone Specifications 27

5.3 Methodology of Collecting Data and Results 28

6 Discussion **31**

6.1 Strengths 31

6.2 Weaknesses 32

7 Conclusion and Future Work **33**

7.1 Conclusion 33

7.2 Future Work 34

List of Figures

1.1 Android Bootup 2

1.2 Activity Lifecycle Pyramid 4

3.1 SOS Architecture 11

4.1 Call graph overlaid on a CFG and Escape Analysis for SlideShow app 17

4.2 Resulting Jimple code for class A after instrumentation 23

4.3 Resulting Jimple code for createObjects() of MainActivity class after instrumentation 24

5.1 Pause Time v/s Iterations of Loop 29

5.2 Freed Memory v/s Iterations of Loop 29

5.3 GC Invocations v/s Iterations of Loop 30

List of Tables

5.1 Evaluation Results 28

Chapter 1

Introduction

1.1 Background

A key feature of modern smartphone platforms is the availability of centralized app stores like Google Play, Apple's App Store and Windows Phone Store, which provide the consumers the convenience of downloading any app at any time from anywhere. Google Play alone stocked about 1.3 million applications by the end of September, 2014 [6]. Undoubtedly, over time, Android has emerged as the most pervasive platform.

Android apps acquire coarse permissions to access user's sensitive data or to get control of various phone components, such as GPS, SD card, camera, etc. These permissions restrict the app to use only a limited number of resources on the phone. This gives the user a sense of assurance that only those resources that are requested will be used. But amidst all these permissions, the permission to access the Random Access Memory (RAM) of the device is granted by default. Even though every app has a quota of heap allocated to it, there is nothing that prevents an app from using this allotted quota inefficiently, and just like work expands to fill the available time (Parkinson's Law), programs expand to fill the available memory.

A vital component of the Android software stack is the Dalvik VM (DVM), which is a process virtual machine that provides platform-independent programming environment for apps. When a system boots, the boot loader loads the linux kernel into the memory. The kernel runs the

init program which is the parent process of all the processes. The *init* process in turn starts the *Zygote* process other daemon processes. The *Zygote* creates a parent Dalvik process which spawns DVM instances, one for every app that wants to start. The *Zygote* also initializes a BSD read socket for listening to DVM instance requests. Figure 1.1 shows this process. The solid lines denote the flow and the dashed lines denote the system calls.

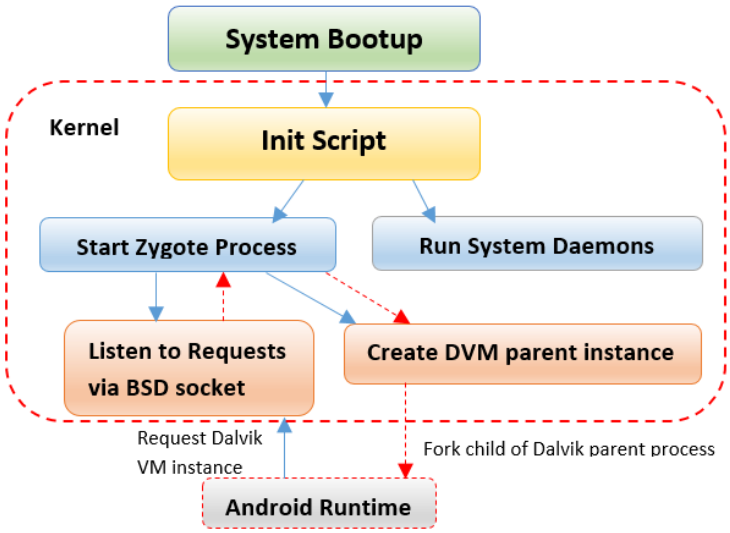

Figure 1.1: Android Bootup

The RAM is a precious resource for any computing device, and it becomes invaluable when one considers resource constrained devices like mobile phones. Although, there is a Dalvik garbage collector to collect the objects that are no more referenced, the developers should be careful of not introducing memory leaks so that the collector can collect these objects. To maintain a functional multi-tasking environment, Android sets a hard limit on the heap size for each app. The exact heap size limit varies among devices based on the overall availability of RAM. If the app reaches the heap capacity and is unable to reclaim memory allocated to objects, it will throw an *OutOfMemoryError* [15].

Android does not offer swap space. Any memory that is modified remains in the RAM and is not swapped out. Thus, the only way to release memory is to release references to the objects, so that all unreachable objects can then be collected by the GC. The GC's execution, however, comes with the cost of pausing the app to collect objects. Although the introduction of *Concurrent Garbage Collection* [4] has reduced the stoppage time, jitters are still prevelant among apps.

Humans are acutely sensitive to jitters and can even notice the smallest of them. Moreover, the more often the GC is invoked, the more battery it will drain. Hence, one should be cautious while allocating objects in the first place and reduce dependence on Dalvik GC to replenish memory.

In addition, the Dalvik heap does not defragment memory to remove the holes. It shrinks only when there is unused space at the end of the heap. The physical memory used by the heap can shrink but only after GC has occured and the unused pages are returned by Dalvik using the *madvise* system call. This makes reclaiming memory from small allocations inefficient because the pages used by small allocations may still be shared with other allocations that have not yet been freed.

Allocating more than what is required has another drawback. When the user switches between apps, Android keeps the processes that are not in the foreground in an LRU (Least Recently Used) cache. If the cached process retains its allocated memory, it restrains the performance of the system by holding on to valuable memory which could have been used by the foreground processes. Hence, in a situation where the apps in the foreground require more space, a process on the LRU gets killed based on the least recently used policy and the amount of memory the object was holding is freed. Thus, in order to avoid an app from getting picked for termination one should pay heed to smart object allocation.

1.2 Android Activity Lifecycle

Activities are the most frequently used component in Android applications. An *Activity* represents a single screen with a user interface. Since we target Android applications in this work for performing optimizations, it becomes imperative to know about the lifecycle of Android activities.

Unlike Java, where applications have a *main()* method, Android sports callback methods that move an activity from one state to another in its lifecycle. When a user navigates in to, out of or back to an app, the Activity instance of the app transitions between different states. These states are depicted in Figure 1.2.

The different states can be shown as a step-pyramid where each state of the activity lifecycle is a separate step on the pyramid. As the activity gets initiated the system calls a sequence

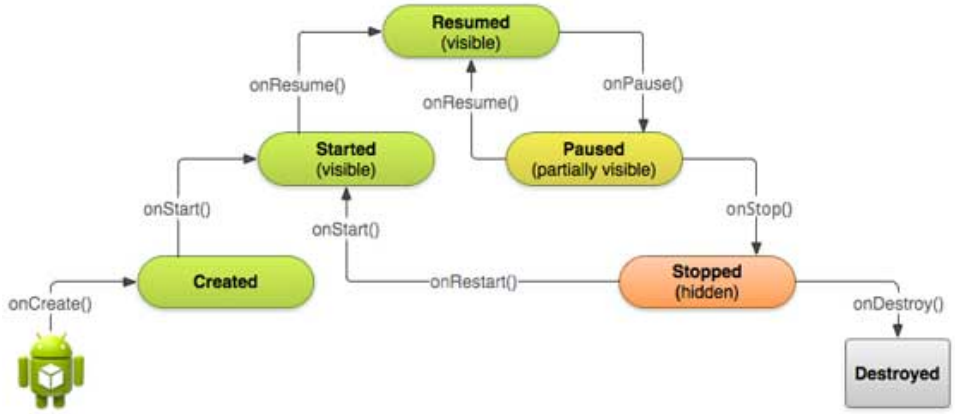

Figure 1.2: Activity Lifecycle Pyramid

of callback methods to move the activity one step towards the top of the pyramid. When an activity is at the top, it is said to be in the foreground and the user can interact with it.

As the user starts to leave the activity, the system calls a sequence of callback methods that bring the activity down the pyramid. The activity does not need to come down to the lowest level of the pyramid to start again. It can move back to the 'resumed' state from a 'paused' or a 'stopped' state. When the activity is in the 'stopped' state it is completely hidden from the user and the user cannot interact with it in any way. The activity instance and all its information is stored in a *Bundle* and retained, however, the app cannot execute any code at this point of time.

In the 'paused' state, the activity is obscured by another activity. The other activity might be semi-transparent or it might cover a part of the screen. The paused activity can neither execute any code nor get a user input at this point in its lifetime. The other states shown in the Figure are transient and the activity quickly moves through them by calling the callback methods.

Apart from Activities, there are three other components, namely, *Services, Content Providers* and *Broadcast Receivers*. A service is component that works in the background to perform long running tasks. Services also have callbacks which move a service from one state to another. However, unlike Activites, they do not have a user interface. Content Providers are used to store and share app data. They provide the functionality of a database. Finally, broadcast receivers respond to system-wide broadcast messages and can act on them by triggering some tasks. Apps can even initiate own broadcast messages.

A unique feature of Android apps is that a component of one app can start a component of another app. Therefore unlike other apps, Android apps have multiple entry points.

1.3 Android Best Practices

Google has listed a number of best practices for managing memory on their website [14]. Some of them are as follows.

(a) *Services* should be used sparingly and should be stopped when its work is complete. The process of a service is retained on the RAM and thus reduces the number of processes that can be kept on the LRU cache, this impacts the app switching time and could even cause thrashing.

(b) They also recommend freeing resources held by an app when it goes in the background. To save heap space, the bitmaps loaded in memory should be scaled down to an appropriate resolution that the device supports.

(c) Android discourages excessive use of code abstraction as it requires more code to be mapped to the memory.

(d) They also disapprove use of dependency injection frameworks which again burdens the RAM by scanning the code for annotations [21].

Even if one follows the aforementioned best practices there is still a chance that the developers can introduce memory bugs or memory leaks. A developer must be wary of the allocations made by the app and must take steps to minimize the memory used. Google has mentioned some useful performance tips at [13]. These tips are based on the following two principles: 1) Avoid doing work that you do not need to do, and 2) Do not allocate memory if it can be avoided. We enumerate some of the tips next.

(a) Prefer use of *static* methods over *virtual*. This makes invocation 15% to 20% faster.

(b) Use of *static final* for constants avoids a field lookup and uses a relatively inexpensive "string constant" instruction for accessing *Strings* and can access *integer* variables directly from the static field initializers of the *dex* file.

(c) Avoiding use of *getters* and *setters* saves the overhead incurred in making *virtual* method calls, instead one should directly access fields.

(d) Use enhanced for loops whenever possible.

(e) Avoid use of *floating point* variables as they are 2 times slower than *ints*.

(f) Finally, avoid creating unnecessary objects, for instance, use *StringBuffer* to avoid creation of intermediate *String* objects which are immutable; and slice multi-dimensional arrays into parallel one dimensional arrays.

In the current discourse, we extend the last guideline by allowing the objects to be reused.

1.4 Our Contribution

Most of the time, garbage collection gets enforced due to the creation of tons of small and short lived objects. Neither Dalvik garbage collector nor Dalvik virtual machine is capable of optimizing such repetitive allocations by itself. The creation of such objects in performance critical paths of an app can cause severe bottlenecks. This is because firstly, such objects could be holding onto heap space even though these objects are not needed anymore, i.e. they are waiting to be collected, and secondly, they increase the overhead on the GC.

In general, the *Object Oriented Programming* (OOP) paradigm encourages programmers to create objects without having apprehensions about the cost that is incurred. However, such a practice can be devastating for Android apps and can degrade the performance and scalability. Android apps, in contrast to other OOP language based apps, advocate creating objects only when it is inevitable. Therefore, in this work, we assist the developers and the app users to transform the memory intensive code to a light-weight version.

We leverage the power of a static Java bytecode analysis framework called *Soot* [3] to perform a *Summary based Interprocedural Escape Analysis*, [1] which identifies objects allocated inside loops that never *escape* the scope of the loop. A "summary" is an abstract element associated with each method that models the effect of calling the method. In a summary-based analysis, the summary of a method can be computed using the summary of all the methods it calls. A summary does not depend upon the context in which a method is called.

The inter-procedural analysis interacts with an intra-procedural analysis that is able to compute the summary of one method, given the summary of all the method it calls, thus, the inter-procedural analysis calls the intra-procedural analysis in a reverse topological order of method dependencies. The intra-procedural analysis works by maintaining an abstract value that represents the 'effect of the method' from its entry point up to the current point. At the entry point, this value is empty. The summary of the method is then the merge of the abstract values at all its return points.

After identification of hoistable sites using the summary based interprocedural analysis, we transform the code to move the allocation statements out of the loop, and hence facilitate reuse of space allocated to the objects. Ofcourse, the objects would have to be re-initialized appropriately before entering the loop in the subsequent iterations. We use customized reset methods, instrumented in the application; these act like constructors for the subsequent iterations of the loop.

As a case study, we take one third party and one custom made Android app and show a detailed evaluation of the benefits that SOS can provide in terms of reduction in pause time and reduction in heap space utilized by the target objects. We also show the trends in the pause time, number of GC invocations and heap space freed due to GC as the number of objects created increases. SOS shows promising early set of results on these apps. In a nutshell, the contributions of our work are:

1. Identification of object allocation sites that can be hoisted using a context sensitive, flow sensitive, summary based interprocedural escape analysis.

2. Automated transformation of code to hoist identified allocation statements outside the loop and reset class fields appropriately for reuse.

3. Evaluation on one third party and one custom Android app to validate our claim that such transformations can yield dramatic reductions in heap space usage and pause times.

4. Showing trends in the number of GC invocations, pause times and memory freed as a function of the number of objects created.

The purpose of our work is not to hack the app; the modified APK would need to be re-signed before installation anyway. The work is also not aimed to gain monetary benefits, rather it is targeted to show that such optimizations are possible by leveraging static analysis and code instrumentation, without the need for source code. There has been a lot of work on optimizing memory utilization for Java apps. However, there has been none for automatically optimizing code of Android apps for efficient memory utilization. In this context, our work elicits a novel approach to detect and deal with memory related issues and equips the developers with a tool that can assist in developing memory efficient apps.

In Chapter 2, we discuss the related works that served as a motivation for pursuing this work. Chapter 3 describes the architecture of SOS. We explain our technique for optimizing memory usage in Chapter 4. In Chapter 5, we evaluate SOS on two Android apps and highlight its benefits. We also show the trends in the garbage collection as the number of objects allocated increases. Then we discuss the strengths and weaknesses of our tool in Chapter 6 and mention the ongoing work in Chapter 7.

Chapter 2

Related Work

2.1 Related Work

In our previous work [5], we optimized apps for mitigating various performance issues related to energy consumption, data consumption and cost to the users. We leveraged static analysis and instrumentation techniques to equip the 'consumers' of an app with the power of controlling the way in which the app could use the granted permissions. Similar to our previous work, we target another performance issue here - optimal use of heap space. However, in contrast to it, this work depends heavily on data flow analysis and can act as a support system for the developers too.

Java objects are created and stored on the heap. On the other hand, local variables of a method are added on the stack after encapsulation in a stack frame. Memory allocation and deallocation on the stack is cheaper and easier, however, a developer cannot decide which objects to allocate on the stack. In the seminal work, *Escape Analysis for Java* [9], the authors propose and implement an algorithm for Escape analysis to infer whether an object can be allocated on the stack. Nowadays, this analysis is performed by the JVM as a standard practice.

In their work [22], Sălcianu and Rinard present a purity and side-effect analysis by leveraging pointer analysis [23] and escape analysis [1]. A 'Pure function' is that which does not modify the state of the program when it is executed. Our analysis is similar to a Purity analysis, however,

here we are concerned only about leveraging Escape analysis to target loops rather than entire methods.

Xu et al. [24] try to detect loop invariant data structures using Soot. They focus on data models and try to identify logical data structures that can be hoisted out of the loop. In contrast to our work, they do not transform the code, instead they restrict themselves to report the *hoistability* of a data structure based on a *dependence-based hoistability metric*. Our analysis, on the other hand, tracks object creation sites, data flows, and assignments to locals and fields. This information enables our analysis to identify objects allocated within a loop that do not escape its scope and hence allow their allocated memory to be re-initialised and reused. We go a step ahead and perform the transformation to reuse object space.

Bhattacharya et al. [8] target Java applications to reuse temporary objects inside loops. They perform source to source transformations and leverage dynamic analysis to prioritize sites that are exercised more than the others. We, in contrast, are targetting Android apps and their inherent multiple entry point and event driven nature renders the dynamic analysis meaningless because user behavior can vary widely from one user to another. Thus, if the apps are lengthy and there arises a need to prioritize optimizable sites, we leverage static analysis and use certain heuristics to prioritize the hoistable sites. In contrast to their work, the transformations we make are at the bytecode level, this becomes important because source code may not be available all the time. Moreover, mobile apps run in a constrained environment on phones with limited resources and providing a smooth UI experience becomes a challenge in itself. This puts more pressure on the $GC\_CONCURRENT$ and GC_FOR_ALLOC. In this context, optimizing Android apps for efficient memory usage has been and will remain an elusive problem.

Chapter 3

SOS Architecture

3.1 Architecture

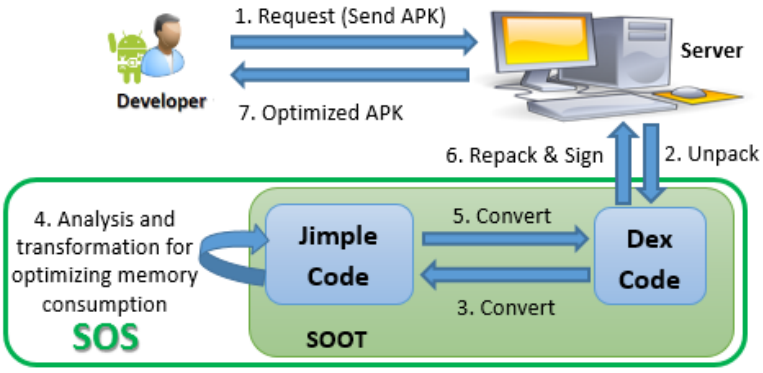

Figure 3.1: SOS Architecture

The architecture of SOS is shown in Figure 3.1. The developer or the user places a request (Step 1) by sending an APK to the server. The sent APK is then unpacked (Step 2) by SOS running on the server into Dalvik bytecode (*dex*). The Dalvik bytecode is converted (Step 3) to Jimple using Soot. SOS, then performs an interprocedural escape analysis in the 'Transformation phase' of Soot (Step 4). After the Transformation phase identifies the target constructs that can be optimized, instrumentation is done to the Jimple code (Step 4). Finally, the Jimple code is converted back to Dalvik bytecode (Step 5) and the *dex* file is packed along with other resources

of the app into an APK. This optimized APK is re-signed and zipaligned [18] (Step 6) before sending to the developer in Step 7.

SOS can accept both an APK (Application Package) or source code as input. One usage scenario of SOS is to equip the developers so that they can use it to optimize their code. Other scenario is that SOS can be used by app users. In the latter case, only the APK would be available. SOS has been designed so that all the required information for the analysis can be collected in a single pass over the code. This helps to speed up the analysis. Moreover, the instrumentation is done carefully so that no side effects are introduced.

3.2 Why not AspectJ?

Soot [3] was used, vis-a-vis AspectJ, as the framework of choice. This design decision was taken based on the following weaknesses of AspectJ:

(a) With AspectJ it is not possible to intercept assignment statements. This renders custom-developed analysis infeasible.

(b) Using AspectJ, one can only go "around" a piece of code but cannot remove it from the app's binary permanently.

(c) Addition of an object allocation statement in the app's binary would not be possible with AspectJ.

These limitations of AspectJ diverted our attention to Soot. The work horse of SOS is Soot's [3] modification known as Dexpler [7], which takes an APK as input, unpacks it, finds the Dalvik bytecode and converts it directly into Jimple. Jimple is an internal three address representation of the code used by Soot and has many advantages. Unlike Dalvik bytecode, it has only fifteen different kinds of statements. Unlike Java bytecode, it is not stack oriented, instead, it uses variables and is closer to a high level language. Furthermore, it is free from the nested structures that are usualy observed in a high level language. This makes Jimple a convenient representation of the code to work with.

Chapter 4

Our Approach

4.1 Overview

After extracting the Dalvik bytecode and converting it to Jimple, Soot's Transformation phase kicks in. In this phase, the application's Jimple code is analyzed by our *interprocedural summary based escape analysis* to detect target constructs of interest (i.e. object allocations inside loops that do not escape the scope of the loop). In other words, this step identifies those object allocation sites which can be hoisted out of a loop. The identification step is followed by a suitable bytecode instrumentation to move the relevant Jimple statements out of the loop, and at the same time, it ensures that no side-effects result. The instrumentation also adds code to invoke constructor-like reset methods to re-initialize object fields. Next, the Optimization phase optimizes the code and finally the Annotation phase performs Soot's built-in analysis. The resultant Jimple code is optimized for heap usage. This eventually leads to a better memory management scheme that advocates reuse of objects.

The overall process comprises of three modules, namely *Interprocedural Escape Analysis Module*, *Hoistable Sites Identifier Module*, and *Bytecode Instrumentation Module*. To illustrate the working of the three modules we consider the following code snippet from our custom made app known as `SlideShow`. The app is available at [2] for reference.

Application Description

This app presents a slide show of 9 images by picking the images one at a time and displaying them in an *ImageView* [16] based on the iterations of the loop. If the number of iterations is greater than 9, then we take a modulo 9 to stay in the range of available images. We could have added more images to avoid repetition but the aim here is to illustrate reuse of objects and a few images were enough for the purpose. For every new image, a new *class A* object is created and the image is encapsulated inside it in the `img` field. `arr_i` and `arr_f` are instance fields that make *class A* objects heavier. The code excerpt given below has a few methods omitted from the "MainActivity" class, so as to emphasize on the relevant portions of the app only.

```
1  class A{
2    int arr_i[] = new int[100000];
3    float arr_f[] = new float[100000];
4    int num; Bitmap img;
5    public A(){
6      num=10;
7    }
8    public String printNum(){
9      return ""+num;
10   }
11 }
12 class B{
13   private A a;
14   public String doSomething(A a, A aa){
15     this.a = a;
16     System.out.println(aa.num);
17     return a.printNum();
18   }
19 }
20 public class MainActivity extends Activity{
21   TextView tv ; ImageView iv;
22   protected void onCreate(Bundle savedInstanceState){
23     super.onCreate(savedInstanceState);
```

```
24      setContentView(R.layout.activity_main);
25      iv = (ImageView) findViewById(R.id.imageView1);
26      tv = (TextView) findViewById(R.id.textView1);
27      final Button button = (Button) findViewById(R.id.btn_heap);
28      button.setOnClickListener(new View.OnClickListener(){
29          public void onClick(View v){
30                createObjects();
31        }
32      });
33    }
34    public void createObjects(){
35      A aObj;
36      B bObj = new B();
37      TypedArray imgs = getResources().obtainTypedArray( R.array.image_ids);
38      int i = 0;
39      while(i<10){
40        aObj = new A();
41        A aa = new A();
42        int resid = imgs.getResourceId(i%9, -1);
43        aa.img = decodeSampledBitmapFromResource( getResources(), resid, 100,100);
44        tv.setText(bObj.doSomething(aObj, aa));
45        new UpdateImages().execute(aa.img);
46        i++;}
47    }
48    //Remaining method/inner class definitions
49  }
```

Listing 4.1: Code snippet from SlideShow app

4.2 Modules

A detailed description about the functions of each of these modules is as follows:

4.2.1 Interprocedural Escape Analysis Module (IEAM)

The process kicks off by generating a precise call graph of the target Android app by leveraging the capabilities of FlowDroid [11]. FlowDroid models the complete lifecycle of Android and its callback methods very precisely. Since Android apps do not have a single entry point (and focus more on callbacks), FlowDroid generates a special dummy main method which emulates all the possible flows and thus acts as an entry point for an app's call graph.

Figure 4.1 shows a part (relevant for further discussion) of the generated call graph overlaid on a CFG for `SlideShow` app. The dotted arrows represent the inter-procedural method invocations, while the solid ones mark the order of execution within a method. The numbers mentioned at the right end of each rectangular enclosure denote the order in which the corresponding statement is encountered in the control flow. The *dummyMainMethod()* (not shown) generated by FlowDroid calls the *onCreate()* method (not shown) of MainActivity which in turn invokes the *createObjects()* method of the same class when the user clicks on the button having ID *btn_heap*. The method *createObject()*, as evident from the figure, makes an interprocedural call to *doSomething()* method of *class B* which further calls *printNum()* method of *class A*.

After the call graph generation, we perform a *Summary-based Inter-procedural Escape Analysis* (SIEA) in order to identify those objects that *escape* the scope of a method. An object is said to *"escape"* a method if the lifetime of that object is not restricted to the method in which it is defined. In other words, if that object can be accessed from other methods within the application, then it is said to have *"escaped"* the method where it is defined. SIEA is implemented by extending the abstract class *AbstractInterProceduralAnalysis* defined in Soot API [3]. The analysis requires that every method in a call graph be associated with a *"summary"*. A *summary*, in our context, refers to a set of objects that escape the scope of a given method.

The key feature of our analysis is that the summary of a method can be computed using the summaries of the callee methods. The *Intra-procedural Escape Analysis* (IEA) is responsible for computing the summary of a particular method, given the summary of all the methods invoked by it. Thus it becomes mandatory for the SIEA to call the IEA in a reverse topological order of

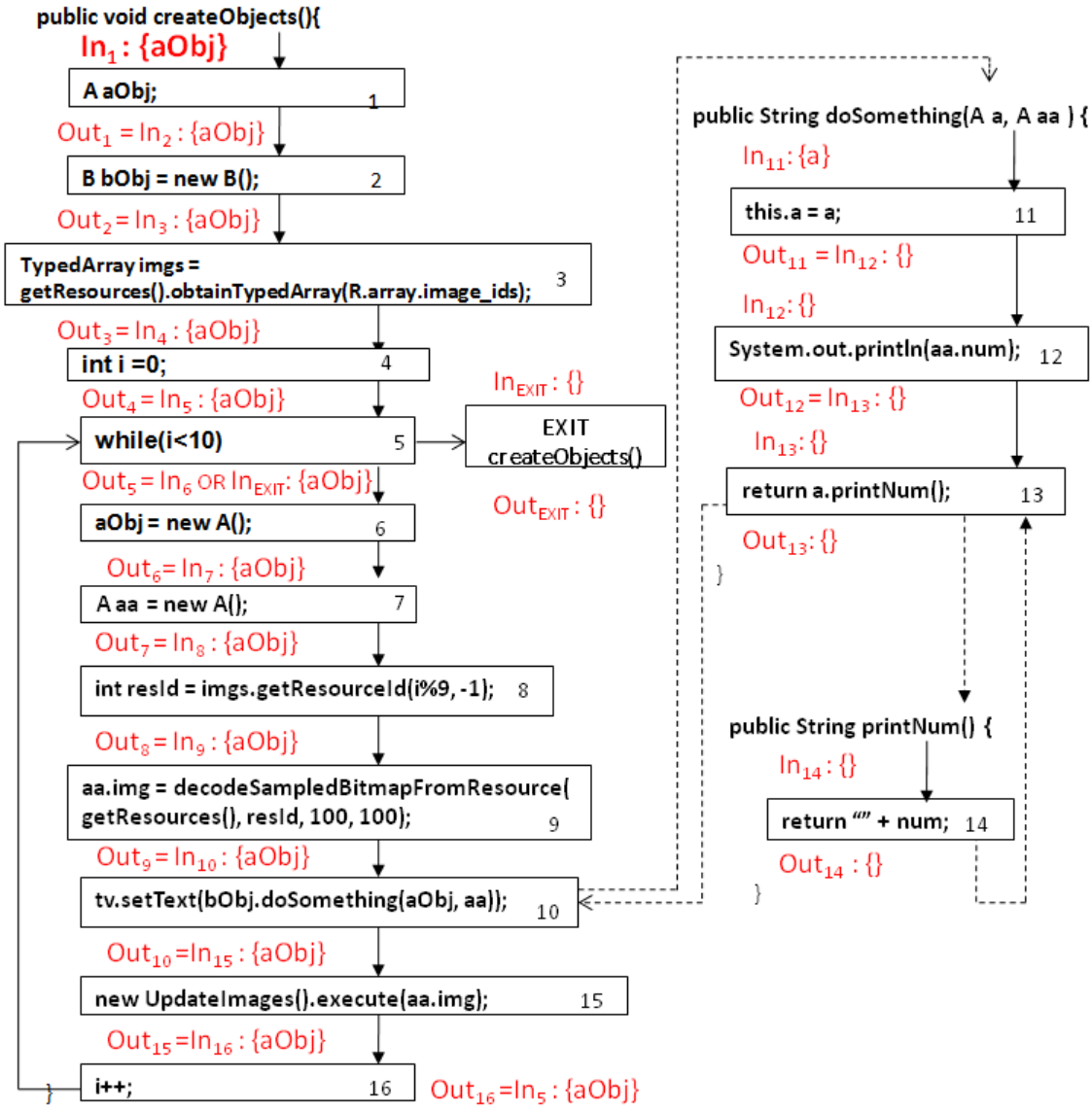

Figure 4.1: Call graph overlaid on a CFG and Escape Analysis for SlideShow app

method dependencies so that the summaries of callee methods is known at the time of performing IEA for the concerned method.

IEA itself is implemented as a backward flow analysis [19] which marks an object as *"escaped"* if it is assigned to a static or an instance field of any class. The variables that are directly or transitively assigned to already deemed escaped variables are also considered as to have escaped.

For every method call encountered during analysis, we fetch the summary for the callee method. The formal arguments that escape from the callee method are mapped to the corresponding

actual arguments. The identified arguments thus form the set of escaped variables and are added to the *In* set for the call site. For each program point inside a method, IEA determines the variables that *must* have escaped on *some* path from that point. We describe the set of escaped variables at entry and exit of every statement S_i in a method as a pair denoted by In_i and Out_i respectively. These sets are initialized and then propagated through the *unit graph* [3] in a backward manner until a *fixed point* [19] is reached.

The flow functions for IEA are expressed as

$$Out_{\mathrm{i}} = \begin{cases} \phi & \text{if } S_{\mathrm{i}} \text{ is exit node in CFG} \\ \bigcup\{In_j \mid \mathrm{S_j} \in succs(\mathrm{S_i}) & \text{otherwise} \end{cases}$$

$In_i = Out_i \cup Esc_i$; where

$$Esc_{\mathrm{i}} = \begin{cases} \{y\} \mid \mathrm{S_i}\text{: } x = y & \text{if x is static/instance field} \\ \{y\} \mid \mathrm{S_i}\text{: } x = y & \text{if x has already escaped} \\ \{y\} \mid \mathrm{S_i}\text{: } x.func(y) & \text{if x} \in \text{java.util.Collection is (alias of) field} \\ \{p_{\mathrm{i}}\} \mid \mathrm{S_i}\text{: } x = f(\mathrm{p_1},\mathrm{p_i}) & \text{if } \mathrm{p_i} \in \text{summary}_{\mathrm{f}} \end{cases}$$

Here, Esc_i corresponds to the set of variables that have escaped due to the presence of code statement S_i in a method. If f is a method, then "$\text{summary}_{\mathrm{f}}$" denotes the summary of method *f*. The summary of a method will be the *In* set of the first statement of that method, i.e. the escaping variables of that method. Figure 4.1 shows the corresponding In_i and Out_i sets for each statement S_i in the call graph where i is the number at the extreme right of the rectangular enclosures. Please note that our escape analysis framework also marks all those objects as 'escaped' which are added/assigned to any field reference of type belonging to java.util.Collection package such as Set, List, Map, Vector etc. The method $func()$ used in the definiton of Esc_i thus may refer to any java library specific method pertaining to the java.util.Collection package which is responsible for adding an object to a collection object. Examples of such library methods include add(), addAll(), put(), putAll(), etc.

After computing the summary of a particular method by using the above flow functions, we use the Soot Pointer Analysis Research Kit (SPARK) framework for assuring that all the aliases of reference variables in the summary are also contained in the summary of the method in concern. SPARK is an accurate flow-insensitive and context-sensitive points-to analysis framework which is shipped with Soot. The flow-insensitive nature of the tool thus renders our approach conservative.

4.2.2 Hoistable Sites Identifier Module (HSIM)

This module runs in parallel with the IEAM. At the time of identifying the set of escaped objects for a method; we also detect the presence of loop constructs, if any, in that method. *LoopFinder* of Soot API [3] was used to detect loops and extract the loop body. After identification of loop constructs in the method, we find those objects which are allocated memory using *new* operator within the loops. All these newly allocated objects are then checked for containment in the *In* set of the method's first statement. The absence of the object in the *In* set indicates that it does not escape the scope of that method and hence the corresponding memory allocation statements can be safely hoisted out of the loop.

Please note that the objects escaping the method will be a superset of the objects escaping any loop in that method. Hence, instead of checking containment in the loop's first statement's *In* set, it suffices to check for containment in the *In* set of the first statement of the method. We mark all such identified object allocation sites as "hoistable" and refer to such sites as "hoistable sites".

The *createObjects()* method in Figure 4.1 contains a while loop within which two objects of *class A*, namely, `aObj` and `aa` are allocated memory. The set of objects that escape from this method, as revealed by SIEA, contains `aObj` but not `aa` because `aObj` gets assigned to a field in *doSomething()* method and hence escapes, however, `aa` does not. Thus, only the object allocation `A aa = new A()` is marked as fit to be hoisted out of this while loop.

Tracking objects that escape the scope of the loop via local variables of the method

Please note that an object allocated inside a loop can also escape the scope of the loop via a local variable. This might happen if inside the loop, the object gets assigned to a reference variable of *Collection* type that is declared outside the loop, for instance, objects allocated in a loop can be collected in a *Collection* declared outside the loop. Such a *Collection* may either be an instance of array or the classes belonging to java.util.Collection package of Java library. We track such cases in our analysis and deem them as "non-hoistable". One such case is manifested in the code given below, where the *ClassB* object escapes the scope of the loop via the local reference variable `hisObj`.

```
1 private void callme() {
2     Set<ClassB> hisObj = new HashSet<ClassB>();
3     for(int i=0;i<10;i++)
4     {
5       ClassB myObj = new ClassB();
6       hisObj.add(myObj);
7     }
8     System.out.println(hisObj);
9   }
```

Listing 4.2: Code snippet: Object escaping the loop's scope via a local variable of the method

At the first look, detecting such a situation might seem quite straight forward. One might think of it as separating the variables declared inside loops from those declared outside the loops and then tracking asssignments (of objects created inside loops) to the variables declared outside the loops. However, this is trivializing a bigger problem. This is because Jimple declares all variables at the beginning of a method even if the variables were declared inside loops in the original source code.

In order to track objects that escape the scope of the loop via a local variable, we use the fact that any object that does not escape the scope of the loop will not be used after the loop ends. Contrapositively, if a reference variable, that refers to an object created inside a loop, is used outside the loop, specially after the end of loop's life, then the object can be said to have escaped.

We examine the loop statements to detect if any object which is allocated memory inside the loop or its alias, say 'obj' gets added to any *Collection* object via put(), putAll(), add(), addAll(), or an assignment statement in case of the *Collection* object being an array. Next we check if that *Collection* object or any of its alias is used in any of the statements that follow the loop construct in execution flow.

If the *Collection* object or its alias is used outside the loop in the method then the newly allocated object inside the loop ie 'obj' is said to escape the scope of the loop. Thus, such an object allocation site cannot be optimized.

4.2.3 Bytecode Instrumentation Module (BIM)

Once all the *hoistable sites* are identified, SOS performs automatic code injection to move all identified sites out of their respective loops. Soot [3] facilitates code injection at the bytecode level by converting Java source code to Jimple and allowing modifications to the Jimple code. Loop hoisting causes the object to be allocated memory only once in its lifetime, i.e. before the control enters the loop, and hence the object loses the chance for re-initialization inside the loop. In order to ensure safe reuse of objects, we propose that a new *reset method* be invoked on them before their reuse inside the loop. This reset method basically simulates the constructor to restore the state of the object back to the one that it had just after memory allocation.

Let us call the constructor invoked at the hoistable object allocation site as *Constructor C* which has a corresponding *<init>()* method I in the equivalent Jimple code. The reset method has exactly the same method signature as that of the *Constructor C*, with a difference that the method name is different. It is named automatically as *reset_i* where i is the number of arguments the *Constructor C* takes as input. Some essential features of our reset method's implementation are as follows:

(a) It contains all the statements of method I except the *special invoke* statement for *<init>()* method of *java.lang.Object* class, which is responsible for actual memory allocation for any object. This avoids allocation of space to objects in the subsequent iterations of the loop.

(b) Our implementation for the reset method also ensures that all those fields which are primitive and have not been assigned any value in the *Constructor C* are assigned a default value in accordance with the Java compiler.

(c) We do not reset static and final instance fields. Moreover, we initialize with *null* those fields which are not initialized in the constructor and are of a type defined by the user.

(d) In case the constructor initializes a field of user-defined type with a *new* statement, we call the reset method corresponding to that object's class instead of the *new* statement.

(e) If the constructor initializes a field of array type, we restore the state of such fields to the Java default state using Arrays.fill() method.

(f) We also handle the cases where fields are of type belonging to java.util.Collection package and java.util.StringBuilder class. The objects of Collection are restored back to their original state by invoking the library method clear() on the object. Similarly, the field of StringBuilder type can be emptied by invoking delete() method on the object which is again provided by Java library. The fields of all other Java/Android library specific type with a *new* statement, we retain the same initialization of such an object in the reset method also. This means that even while resetting the fields of such types, new memory is allocated and thus ensures partial reuse.

(g) We also initialize variables that are inherited from a super class which is one level higher in the inheritance hierarchy. We do not initialize other reference variables derived from the super class because if the super class is a class belonging to the Android library, then its implementation is not present in the android.jar, rather the methods/constructors are only defined as stubs. Hence, we try to synthesize *reset()* method on a best effort basis since a precise resetting of the fields of an object would require availability of the Android API implementation, which currently resides on the devices only and is not a part of the android.jar.

Thus, our technique presents a precise, conservative but unsound model to automatically generate the reset() method functionality based on the implementation of the constructor.

```
class com.example.toyexample.A extends java.lang.Object{
    float[] arrf; int[] arri;
    android.graphics.Bitmap img; int num;
    public void <init>(){
        com.example.toyexample.A $r0;
        int[] $r1; float[] $r2;
        $r0 := @this: com.example.toyexample.A;
        specialinvoke $r0.<java.lang.Object: void <init>()>();
        $r1 = newarray (int)[100000];
        $r0.<com.example.toyexample.A: int[] arri> = $r1;
        $r2 = newarray (float)[100000];
        $r0.<com.example.toyexample.A: float[] arrf> = $r2;
        $r0.<com.example.toyexample.A: int num> = 10;
        return;
    }
    public void reset_0(){
        com.example.toyexample.A $r0;
        int[] $r1; float[] $r2;
        $r0 := @this: com.example.toyexample.A;
        $r0.<com.example.toyexample.A: android.graphics.Bitmap img> = null;
        $r1 = $r0.<com.example.toyexample.A: int[] arri>;
        staticinvoke <java.util.Arrays: void fill(int[],int)>($r1, 0);
        $r2 = $r0.<com.example.toyexample.A: float[] arrf>;
        staticinvoke <java.util.Arrays: void fill(float[],float)>($r2,0.0F);
        $r0.<com.example.toyexample.A: int num> = 10;
        return;
    }
    public java.lang.String printNum(){
```

Figure 4.2: Resulting Jimple code for class A after instrumentation

Figure 4.2 and 4.3 present snapshots of the Jimple code for *class A* and *MainActivity* respectively which are generated after instrumentation.

In Figure 4.3, the loop is identified by `label1`. After the analysis, $\$r2$ was declared as *hoistable* from the loop. Note that $\$r2$ is the temporary variable generated corresponding to the object *aa* of type *A* during the automatic conversion from source code to Jimple. The original memory allocation site for $\$r2$ inside `label2` of *createObjects()* now invokes *reset_0()* on $\$r2$ instead of *<init>()*. Further, *<init>()* along with *new statement* for $\$r2$ is invoked just before the control enters `label1`.

Figure 4.2 shows a newly added method with signature `public void reset_0()` which is a partial replica of `public void <init>()`. This method, in contrast to *<init>()*, does not contain statement for *specialinvoke* to *<init>()* of *java.lang.Object* on $\$r0$.

```
public void createObjects(){
    com.example.toyexample.MainActivity $r0;
    com.example.toyexample.A $r1, $r2;
    com.example.toyexample.B $r3;
    //Remaining variable declarations
    $r0 := @this: com.example.toyexample.MainActivity;
    $r3 = new com.example.toyexample.B;
    specialinvoke $r3.<com.example.toyexample.B: void <init>()>();
    $r5 = virtualinvoke $r0.<com.example.toyexample.MainActivity:
                        android.content.res.Resources getResources()>();
    $r6 = virtualinvoke $r5.<android.content.res.Resources:
        android.content.res.TypedArray obtainTypedArray(int)>(2130968576);
    $i0 = 0;
    $r2 = new com.example.toyexample.A;
    specialinvoke $r2.<com.example.toyexample.A: void <init>()>();
label1:
    if $i0 < 10 goto label2;
    return;
label2:
    $r1 = new com.example.toyexample.A;
    specialinvoke $r1.<com.example.toyexample.A: void <init>()>();
    $r1.<com.example.toyexample.A: int num> = 5;
    virtualinvoke $r2.<com.example.toyexample.A: void reset_0()>();
    $i1 = $i0 % 9;
    $i1 = virtualinvoke $r6.<android.content.res.TypedArray:
                            int getResourceId(int,int)>($i1, -1);
    $r5 = virtualinvoke $r0.<com.example.toyexample.MainActivity:
                        android.content.res.Resources getResources()>();
    $r4 = staticinvoke <com.example.toyexample.MainActivity:
    android.graphics.Bitmap decodeSampledBitmapFromResource(
        android.content.res.Resources,int,int,int)>($r5, $i1, 100, 100);
    $r2.<com.example.toyexample.A: android.graphics.Bitmap img> = $r4;
    $r7=$r0.<com.example.toyexample.MainActivity:android.widget.TextView tv>;
    $r8 = virtualinvoke $r3.<com.example.toyexample.B: java.lang.String
    doSomething(com.example.toyexample.A,com.example.toyexample.A)>($r1, $r2);
```

Figure 4.3: Resulting Jimple code for createObjects() of MainActivity class after instrumentation

The modified APKs generated as a result of the instrumentation need to be re-signed by a certificate created using a private key before its deployment, otherwise the new app will not get installed. The optimized APK should also preferably be *zipaligned*. This causes all uncompressed data like images or raw files to be aligned on 4-byte boundaries.

4.3 Heuristics

For large apps, the number of hoistable sites can reach a large number. Hence, if we are to remain within a given time budget or wish to perform the optimizations to only a few high impact sites, we would have to find out the top contributors to object allocations that are optimizable.

In [8], the authors resort to dynamic analysis techniques, however, the cost of dynamic analysis is an overkill and in the context of Android apps could even be useless because different people have different behaviors when it comes to using apps. Hence, we rely on the following heuristics for prioritizing hoistable sites statically:

(a) New allocations can occur indirectly inside loops. A method that allocates objects can be called from within a loop. In such a case detecting the correct instrumentation point becomes non-trivial. In this work, we do not ignore such cases (we propose a technique to detect them) because they form a strong candidate for optimization, however we leave the detection of correct instrumentation point for future work.

The technique for detecting such a case is as follows. We maintain a set of methods that have new allocation statements directly (not in a loop) in them. We also maintain a set of methods that call another method from within a loop. Now we perform a reachability test between every combination in the two sets. If the two methods in consideration are reachable then there is scope for optimization.

(b) Betweenness Centrality: We calculate betweenness centrality for every method in the call graph generated by FlowDroid. Betweenness centrality of a node (in a graph) quantifies the number of times that particular node appears as a bridge along the shortest path between other two nodes. While calculating betweenness centrality, we also include paths which have the node in question as their terminal or initial node. Methods which get high betweenness scores should be given priority when selecting a subset of the hoistable sites for optimization. We leave the instrumentation point detection as future scope.

Chapter 5

Evaluation

5.1 Evaluation Overview

In this chapter we validate our hypothesis that SOS is helpful in reducing the memory consumption and reducing garbage collector invocations, thereby reducing the pause times. We manifest the empirical study conducted on two Android apps, one of which is our own custom developed app and the other a third party application. The details about the testbed and the methodology of collecting results are presented first. Then we present the results from the two Android apps and show the improvements that SOS can yield. Finally, we take one of the apps and show the trends in the pause times due to garbage collector invocations, the memory freed as a result, and the number GC invocations as a function of the number of loop's iterations.

5.2 Testbed

5.2.1 Application Description

We used a tailor made app - `SlideShow` available at [2], and a third party app - `Custom ListView` [10]. We will call them `SS` and `CLV` respectively. The `SS` app displays a slide show in which the images displayed in an *ImageView* [16] keep changing. This app stores each image encapsulated in its own new *class A* object and optimize the app to reuse the first allocated

object to store the subsequent images and display them. This app rescales images so as to avoid excess memory consumption by Bitmaps. If this is not done, loading images would cause even more GC and increase pause times even more. We go a step further and use this app to highlight the trends of garbage collection as the number of iterations (slides) are increased. As mentioned earlier, if the number of iterations of the loop is more than 9, then we perform a modulo 9 to the integer iterator variable and display the image with the resultant *Resource ID*.

The `CLV` app is a tutorial that illustrates how to make apps that use *ListView* class [16]. The app contains three classes described as follows:

(a) *CustomListViewAndroidExample* class - which creates an *ArrayList* of 11 items and sends it to the *CustomAdapter* class

(b) *CustomAdapter* class - which build the view holder for these 11 items and displays them

(c) *ListModel* class that represents a list item

We tweak this app to create three such *ArrayLists* of *ListItem* so that we could elicit reuse of the *ListView* items. A point to note here is that since all the 11 items appear on screen at once, we cannot reuse one item to create another item in the same list. This is because the earlier 10 objects will end up having the same instance values as that of the 11^{th} one. However, we can reuse one entire *ArrayList* of items to create another one.

5.2.2 Phone Specifications

We performed our experiments with these two apps on two Android smartphones - a) *Sony Xperia P LT22i* (dual-core, 1 GHz Cortex-A9 processor, 1 GB RAM, Android version 4.1.2), b) *Moto G* (quad-core, 1.2 GHz Cortex-A7, 1 GB RAM, Android version 4.4.4). The heap size allowed for the apps was 64 MB on Xperia P and 96 MB on Moto G. However, Android starts off every app with a lower heap limit and grows the heap whenever more space is required.

Table 5.1: Evaluation Results

App	Device	Metric	U-OPT	OPT	Saving	Saving in %
Slide Show	XperiaP	S_{avg} (KB)	15625	8594	7031	45
		$T_{avg}(ms)$	138.7	50.2	88.5	63.8
	MotoG	S_{avg} (KB)	15625	8594	7031	45
		$T_{avg}(ms)$	67	23.8	43.2	64.48
Custom ListView	XperiaP	S_{avg} (KB)	42970	4297	38673	90
		$T_{avg}(ms)$	169	0	169	100
	MotoG	S_{avg} (KB)	42970	4297	38673	90
		$T_{avg}(ms)$	83.8	0	83.8	100

Here:

U-OPT : Un-optimized

OPT : Optimized

S_{avg} : Average heap space consumed by the target objects in KB where the average is over 10 readings

T_{avg} : Average pause times for GC in ms where the average is over 10 readings

Saving in % = (Saving/U-OPT)*100

5.3 Methodology of Collecting Data and Results

We ran two versions (unoptimized and optimized) of each app on the phones and measured the *Pause times due to GC* and *Memory consumption* due to allocation of the target objects, where the *'target objects'* are those objects that can be hoisted out of the loop. The optimized (OPT) and unoptimized (U-OPT) versions of the apps were ran 10 times each and the results were averaged over the readings.

The logs from the *logcat* [12] were used for measuring the total pause times. We built a small parser to accumulate results from the logs automatically. The DDMS, packaged with Android SDK, [17] features an *Allocation Tracker* [12] which we used to measure the space allocated to the target objects on the heap.

Table 5.1 tabulates the time and space measurements obtained from both the phones for both the apps and highlights the savings that SOS yields. These readings were taken for 10 iterations of the loops in both the `SS` and the `CLV` app. The GC pause times mentioned in the table are a sum of the pause times for GC_FOR_ALLOC and $GC\_CONCURRENT$ types [12]. Other types of GCs were not encountered in the apps we profiled.

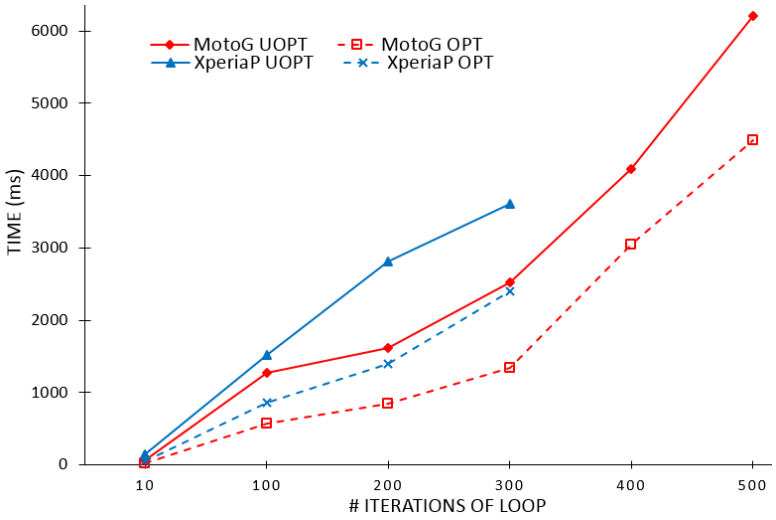

Figure 5.1: Pause Time v/s Iterations of Loop

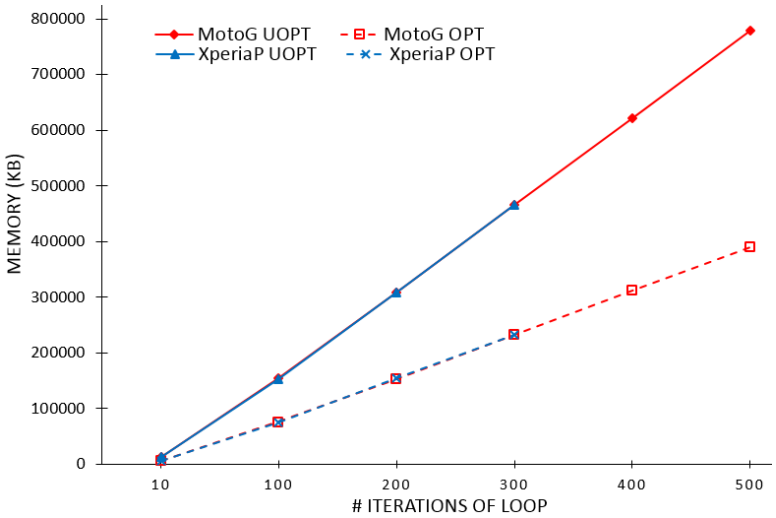

Figure 5.2: Freed Memory v/s Iterations of Loop

We also monitored the trends in *a) Pause times due to GC, b) Memory freed due to GC* and *c) Number of GC invocations*, for the `SS` app as the number of iterations of the loops are increased. Figures 5.1, 5.2, and 5.3 show these trends. The X axis denotes the iterations. The Y axis denotes *Pause Times, Memory freed* and *Number of Invocations* respectively. The solid lines represent readings from the un-optimized version of the app and the dashed lines denote readings from the optimized versions. The red lines depict readings from Moto G and the blue ones depict readings from Xperia P.

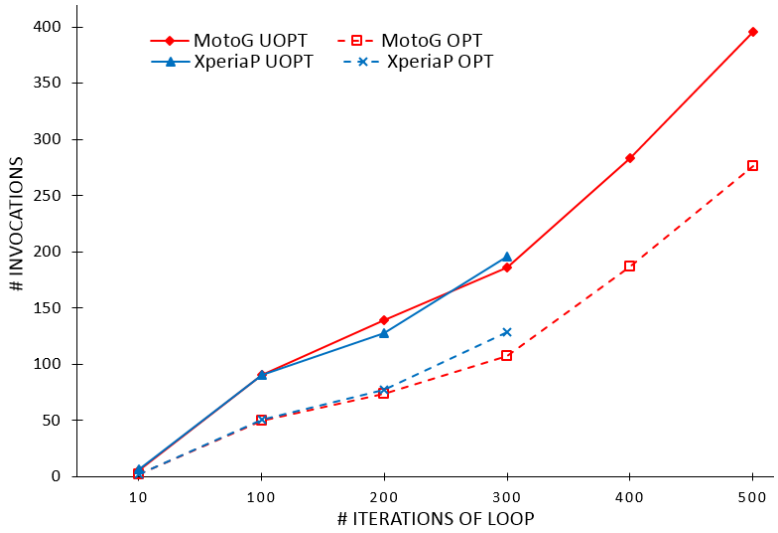

Figure 5.3: GC Invocations v/s Iterations of Loop

It can be observed from Figure 5.1 that the pause times for Moto G were relatively more than those for Xperia P; which means that the likelihood of jitters is more in Moto G. Not explicitly shown is the point at which the apps run out of memory and display an *OutOfMemory* error. This happens when the apps spend 98% of their time in collecting garbage and less than 2% of heap is reclaimed [20]. In Xperia P, the heap available for the apps was 64 MB and as a result the `SS` app went out of memory earlier as compared to Moto G, which had a heap size of 96 MB available. The point at which the lines discontinue is the point after which the app started showing an *OutOfMemory* error. This is beyond 300 iterations for Xperia P and beyond 500 iterations for Moto G. All the three line graphs show a linear characteristic emphasizing that the GC overhead (of pause times, invocations and freed memory) increases linearly as the objects increase.

Chapter 6

Discussion

6.1 Strengths

The strengths of our approach are as follows:

(a) We have been careful in making SOS conservative but at the same time not making it overly conservative.

(b) SOS expands the horizon of analysis beyond a method due to its summary based approach.

(c) It provides a generic framework to optimize memory and several other optimizations can be simply incorporated without the need for another pass over the code.

(d) SOS completely relies on static analysis and avoids the overhead of dynamic analysis.

(e) The use of Dexpler avoids an intermediate conversion to Java Bytecode thereby speeding up the overall process and the use of Soot makes our approach scalable to large code bases.

(f) SOS has been designed so that all the required information for the analysis can be collected in a single pass over the code. This helps to speed up the analysis. Moreover, the instrumentation is done carefully so that no side effects are introduced.

6.2 Weaknesses

Despite of its benefits, SOS suffers from a few limitations which are as follows:

(a) We do not claim that the tool covers each and every case of optimization as it is still in a prototype phase and there is plenty of scope for improvement. For instance, SOS cannot reuse objects created in a callee method that has its call site in a loop inside a caller method. In such a case the instrumentation would happen in the caller method because of a site in the callee method. In this work we have proposed and implemented a technique to detect such cases, however, we leave the detection of correct instrumentation point as future work.

(b) The modified APKs need to be re-signed before they can be deployed on smartphone. Hence, preferably the developer would have to resign the APK. In case they are signed by us, the app updates will stop coming. Moreover, the users would have to trust the changes made by us.

(c) Soot and FlowDroid can be slow sometimes, and could need a lot of memory; which is why we offload the analysis to a server.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

From the inception of *Object Oriented Programming*, programmers have been creating objects for even the smallest and simplest of tasks without paying attention to the cost incurred. They take for granted that the system will optimize all inefficiencies. However, neither the Just-In-Time compiler nor the Dalvik garbage collector can by itself completely optimize object allocations. Creating such short lived objects profligately could cause the performance and the end user experience to degrade severely.

In this work we presented a tool that optimizes allocation of objects by reusing heap space allocated to objects inside loops. If these objects do not escape the scope of the loop, then we hoist their respective allocation statements outside the loop so that the heap memory is allocated only once and reused in the subsequent iterations. For the subsequent iterations, the class fields have to be re-initialised appropriately using the body of the constructors already present in the app or using our own constructors for providing default values.

We also showed results from our experiments conducted over two Android apps and highlighted the benefits that SOS can provide.

7.2 Future Work

A few memory issues that we are working on and would like to take up next are as follows:

1. We would like to extend our analysis to reuse objects between multiple apps. This would necessitate a provision of an object pool to keep certain objects in memory for reuse later. Though, there is a tradeoff involved - at one end we are keeping valuable memory reserved and at the other we are reusing these objects in the pool to avoid unncessary allocations in the hope that it will reduce jitters and the number of times GC gets invoked.

2. Detecting memory leaks statically so that the developer is warned before actually executing the app about their presence.

3. Reducing creation of intermediate temporary String objects which could reduce the burden on GC.

4. Loops can also be present in a separate caller method and the allocation might happen in the callee method called from within the loop. Currently, we are in the process of adding support for such cases.

5. While resetting the fields of a class, the non-primitive fields inherited from a parent Android library classes are left alone because their constructors are defined as stubs in the android.jar. We would appreciate if their implementation is shared by Google as then we will be able to initialize such fields also.

With the existing features and those that will be added in future, we hope that SOS helps users and developers to utilize heap space efficiently.

Bibliography

[1] Escape Analysis. http://en.wikipedia.org/wiki/Escape_analysis.

[2] Link to SlideShow App. https://www.dropbox.com/s/8jtsjzhgfh4h1vy/ToyExample.rar?dl=0.

[3] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[4] Alois Reitbauer, e. a. Reducing Garbage-Collection Pause Time. http://javabook.compuware.com/content/memory/reduce-garbage-collection-pause-time.aspx.

[5] Anwer, S., Aggarwal, A., Purandare, R., and Naik, V. Chiromancer: A tool for boosting android application performance. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems* (New York, NY, USA, 2014), MOBILESoft 2014, ACM, pp. 62–65.

[6] AppBrain. Number of available Android applications - AppBrain. http://www.appbrain.com/stats/number-of-android-apps.

[7] Bartel, A., Klein, J., Monperrus, M., and Le Traon, Y. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *Proceedings of the International Workshop on the State Of the Art in Java Program Analysis (SOAP'2012)* (2012).

[8] Bhattacharya, S., Nanda, M. G., Gopinath, K., and Gupta, M. Reuse, recycle to de-bloat software. In *Proceedings of the 25th European Conference on Object-oriented Programming* (Berlin, Heidelberg, 2011), ECOOP'11, Springer-Verlag, pp. 408–432.

[9] Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C., and Midkiff, S. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 1999), OOPSLA '99, ACM, pp. 1–19.

[10] Example, A. How To Create A Custom Listview - Android Example. http://androidexample.com/How_To_Create_A_Custom_Listview_-_Android_Example/index.php?view=article_discription&aid=67&aaid=9.

[11] Fritz, C. Flowdroid: A Precise and Scalable Data Flow Analysis for Android. In *EC SPRIDE* (2013).

[12] Google. Investigating Your RAM Usage. http://developer.android.com/tools/debugging/debugging-memory.html#ViewingAllocations.

[13] Google. Location Performance Tips. http://developer.android.com/training/articles/perf-tips.html.

[14] Google. Managing Your App's Memory. https://developer.android.com/training/articles/memory.html.

[15] Google. Out Of Memory Error. http://developer.android.com/reference/java/lang/OutOfMemoryError.html.

[16] Google. Package Index. http://developer.android.com/reference/packages.html.

[17] Google. Using DDMS. http://developer.android.com/tools/debugging/ddms.html.

[18] Google. zipalign. https://developer.android.com/tool/help/zipalign.html.

[19] Johspaeth. Implementing an intra procedural data flow analysis in Soot. https://github.com/Sable/soot/wiki/Implementing-an-intra-procedural-data-flow-analysis-in-Soot.

[20] Oracle. Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning. http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#available_collectors.selecting.

[21] Oracle. Lesson: Annotations. http://docs.oracle.com/javase/tutorial/java/annotations/.

[22] Sălcianu, A., and Rinard, M. Purity and Side Effect Analysis for Java Programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2005), VMCAI'05, Springer-Verlag, pp. 199–215.

[23] Wikipedia. Pointer Analysis. http://en.wikipedia.org/wiki/Pointer_analysis.

[24] Xu, G., Yan, D., and Rountev, A. Static detection of loop-invariant data structures. In *ECOOP 2012 – Object-Oriented Programming*, J. Noble, Ed., vol. 7313 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 738–763.