

R3: Reduce, Reuse and Recycle

Student Name: Samit Anwer

IIIT-D-MTech-CS-MT-12-074

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 Mobile Computing

©2014 Samit Anwer
All rights reserved

Keywords: Android, Memory Optimization, Garbage Collection, Performance, Static Analysis, Code Instrumentation, Soot, Escape analysis

Certificate

This is to certify that the thesis titled “**R3: Reduce, Reuse and Recycle**” submitted by **Samit Anwer** 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 him under my guidance and supervision in the Mobile and Ubiquitous Computing 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

Every Android application runs in its own virtual machine, with its own Linux user account and corresponding permissions. Although this ensures that permissions are given as per each application's requirements, each permission itself is still broad enough to possible exploitation. The heap memory can be accessed by default by all apps and can be misutilized to unimaginable extents. Such exploitations may result in an over consumption of phone's resources, in terms of memory, battery, and communication bandwidth. In this work, we propose a tool called *R3*, for the app developers and users to control application's permissions at a fine granularity thereby reducing the exploitation of permissions. We provide the developers an opportunity to recycle the objects that are short lived and created in large numbers so that they can be reused instead of getting garbage collected. The framework is based on static code analysis and code instrumentation. It takes in compiled code and so does not require access to source code of the application. As a case study, we passed publicly available applications through *R3* to fine tune their performance. We compared energy, data and memory consumed by these applications before and after the code injection to corroborate our claims of improvement in performance. The data consumption reduced by a factor of 12.2 after removing advertisements, energy consumption reduced by a factor of 1.88 by optimizing the wake lock type and energy consumption reduced by a factor of 3.7 after optimizing GPS location update frequency. The pause times due to garbage collection reduced from 184 ms to 80 ms as the object pool size was increased from 0 to 1000.

Acknowledgments

I would like to thank Dr. Rahul Purandare for his unparalleled guidance and motivation. Without his guidance this work would not have been possible. I would like to take this opportunity to express gratitude to him for providing me with opportunities to attend and present at top-tier conferences and also for providing a strong platform to undertake research as a career. The amount of time that he has taken out from his busy schedule is quite commendable and worth mentioning. I thank him from the bottom of my heart.

My heartfelt thanks to Dr. Vinayak Naik for his valuable suggestions. He has always been there to support me and to share his expertise in the mobile platform domain.

I would like to thank Dr. Eric Bodden and Steven Arzt for sharing their expertise in Soot and insights throughout the course of this work. A special thanks to all my friend for critically reviewing and commenting on the work and helping to make it what it is. I would like to thank my parents for always supporting me and providing me an environment where I could work dedicatedly. Last but not the least, I would like to thank IIIT Delhi for making this happen. The infrastructure, services and environment provided to us as students are truly remarkable.

Samit Anwer

New Delhi

Contents

1	Introduction	1
1.1	Background	1
1.2	Our Contribution	3
2	Related Work	7
2.1	Related Work	7
3	R3 Architecture	10
3.1	R3 Architecture	10
3.2	R3 GUI	12
4	Chiromancer Framework	13
4.1	Chiromancer Framework	13
5	Our Approach	16
5.1	Reduce	16
5.1.1	Advertisements in applications	16
5.1.2	SMS to premium rate numbers	17
5.1.3	GPS location update frequency is too high	19
5.1.4	Inappropriate wake lock acquired	20

5.2	Reuse and Recycle	21
5.2.1	Inefficient Heap space utilization	21
5.2.2	Heuristics	23
5.2.3	Modules	24
6	Evaluation	40
6.1	Evaluation Overview	40
6.2	Testbed	40
6.3	Methodology of Collecting Results	41
7	Discussion	47
7.1	Strengths	47
7.2	Weaknesses	49
8	Conclusion and Future Work	50
8.1	Conclusion and Future Work	50

List of Figures

1.1	Android OS Stack	2
3.1	R3 Architecture	10
3.2	GUI of the app	12
4.1	Chiromancer Framework - Levels of abstraction	15
5.1	Advertisement application Jimple code after injection	17
5.2	Premium rate SMS application Jimple code after injection	18
5.3	GPS Location application Jimple code before injection	19
5.4	GPS Location application Jimple code after injection	19
5.5	Wake lock application Jimple code before injection	20
5.6	Wake lock application Jimple code after injection	20
5.7	Hill Climb Racing: Short-lived objects	21
5.8	Call graph overlaid on a CFG and Escape Analysis for SlideShow app	27
5.9	CFG of testPool() method and LUP analysis on SlideShow app	32
6.1	GPSShake Lite results OPT v/s U-OPT for both phones	43
6.2	Swing Ball results OPT v/s U-OPT for both phones	43
6.3	Accelerometer Monitor results OPT v/s U-OPT for both phones	44
6.4	Results from Memory Optimization	46

List of Tables

5.1	Wake lock levels and component On/Off information	20
6.1	Chiromancer Evaluation Results	42
6.2	Memory Optimization Evaluation Results	44

Chapter 1

Introduction

1.1 Background

Android has about 75% of the market share with a growth rate of 91.5% in the last year [22]. Undoubtedly, Android is the most pervasive mobile OS. Performance and security are the dominant factors considered when gauging the quality of Android applications. According to AppBrain [6] by the end of September 2014, there were a total of 1,376,862 applications (apps) in the Android market store and 219,250 of these have been declared as low quality apps. This is a large number considering the fact that Google removes low quality applications from the Android marketplace quarterly.

With increasing rate of cybercrime, people are more concerned about malwares and privacy issues. However, performance issues are equally important as they might impact the end user experience, data, memory and battery consumption of an application. Choosing Android applications from Google Play is similar to choosing apples on an apple tree. Before plucking, nobody knows whether there is a worm in them. Android applications could also have worms; worms which hamper their performance by either devouring a phone's battery or by profligately consuming network data or memory. This motivates us to target Android applications (apps) for proposing performance optimizations.

Figure 1.1 depicts the Android software stack. From bottom to top it consists of a Linux Kernel, a Middleware, Libraries, Android Runtime Environment, and Applications running on top of the Application Framework. The application layer consists of (a) in-built applications like Home, Contacts, Phone, and Browser and (b) user-installed applications. The Application Framework, on top of which these applications run, consists of various Managers like Activity manager, Telephony manager, Resource manager, Location manager, etc. Below the Application Framework layer, are the Libraries and the Android Runtime Environment, which contains the Dalvik Virtual Machine.

The Dalvik Virtual Machine is a process virtual machine that provides applications a platform independent environment to run. Android uses Dalvik Virtual Machine to run dex code, which is formed from Java bytecode. The bottom-most layer is where the Linux Kernel resides. The underlying Linux system provides features, like user-based permission model [10], process isolation and sandboxed environments for applications. Such features of Android make Google Play a favorite hub for downloading applications.

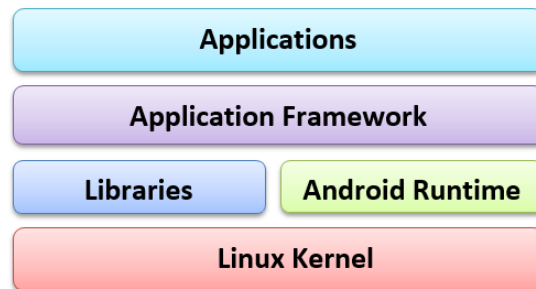


Figure 1.1: Android OS Stack

1.2 Our Contribution

Poor performance of an application directly impacts many aspects like data consumption, battery consumption, memory consumption, cost incurred by user, etc. These concerns of users/developers can become evident (even prominent) after long periods of application usage. With these concerns in mind, our work entails devising strategies to tackle different problems that hamper performance. Through R3, we want the application users and developers to be sure that the code of the application is going to efficiently use the resources on the device.

The work R3 can be logically segregated into two components:

1. Chiromancer (Reduce paradigm)

We initially present our work *Chiromancer* [13], a prototype that takes Application Package (APK) of apps and performs static analysis to uncover performance issues. The issues we consider are -

- (a) Cost incurred to users due to SMS to premium rate numbers.
- (b) Data consumption due to In-app advertisements
- (c) Energy consumption due to very frequent GPS location fixes
- (d) Energy consumption due to inappropriate wakelocks.

After detecting these issues, the tool transforms the APK based on the inputs provided by the user so that the app gets tuned according to the user's needs rather than the developer's whims. Such modifications may not always be meaningful since the user is completely code agnostic. However, the user is informed by our app about the implications of the modifications a priori. Our approach relies on the fact that the user understands her needs better than the developer. By allowing the users to tweak the input parameters, the tool offers them a better control over the applications.

We assume that the user must have an idea about the kind of application she is going to install, which is generally true or can be known by reading the app's features on the store. This way, the user can decide certain aspects of the application. For instance, let us assume that the app

is a game like Chess. Time is not a major factor in the game, so a user can choose to have a *PARTIAL_WAKE_LOCK* [7] instead of a *FULL_WAKE_LOCK* [7]. This would allow the screen to go off after a while thereby saving energy. In contrast, for a car racing game it would not be wise to keep a *PARTIAL_WAKE_LOCK*. The aforementioned four optimizations fall under the ‘*Reduce*’ category of our work since they reduce energy consumption, data consumption and cost incurred by users.

2. Extension of Chiromancer (Reuse and Recycle paradigm)

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, advocate creating objects only when it is inevitable.

Android’s Dalvik garbage collection does a good job of reclaiming memory. But the garbage collector introduces pauses during execution of the app to recover memory. Although the introduction of *Concurrent Garbage Collection* [12] has reduced the stoppage times to quite an extent, jitters are still prevalent among apps. Jitters are usually experienced when a large number of short lived objects are created. Garbage collection can happen at any time and as a result the execution time becomes uncertain.

Excessive object allocation causes fragmentation in memory and leaves holes which cannot be used by the app unless the memory gets compacted. The Random Access Memory (RAM) is a precious and scarce resource on handheld devices. Android has a Least Recently Used cache that stores the processes/apps that are moved to the background for quick retrieval. If an app in the foreground needs more memory, then one of the apps in the LRU cache might get picked for termination based on how much memory an app is holding and the LRU policy. Thus, to prevent termination of an app a developer must strive to reduce GC invocations as much as possible by allocating memory wisely.

Realizing the importance of memory management in the context of Android devices, we extend Chiromancer with a memory optimization that allows objects to be recycled to object pools so that they can be reused if they are needed again in the application. This optimization emphasizes

on object *‘Reuse’* and *‘Recycle’*. By means of this optimization, we assist the developers and the app users to transform the memory intensive code to a light-weight version by leveraging the *object pool* design pattern. This optimization could have been considered as a part of the *‘Reduce’* category as we would be reducing the number of objects that are allocated, however, to differentiate from Chiromancer we consider this optimization as a part of the *‘Reuse’* and *‘Recycle’* category.

We use an umbrella term for all these optimizations - “R3”, which stands for ‘Reduce, Reuse and Recycle’. Please note that the tool can be used by both, the general public and the app developers to detect and overcome performance issues. Since, both Chiromancer and R3 use the same architecture, we use the terms interchangeably in Section 3.1 .

In essence, the contributions of this work are as follows:

1. In Chiromancer (Reduce), we identify statements in the code that cause deterioration of performance and inform the user of the current configurations.
2. We perform transformation of code to alter configurations according to those specified by the user.
3. We evaluate Chiromancer on third party apps from Google Play.
4. As a part of the extension (Reuse and Recycle), we analyse apps for detecting objects that are created temporarily but multiple times in an app. These objects can then be recycled by leveraging a melange of Points-To analysis, Escape analysis and Last Usage Point analysis.
5. We present certain heuristics to prioritize objects for optimization and thus reduce the amount of instrumentation that needs to be done.
6. We transform the code to cache objects in an object pool and use these objects when required rather than allocating space for new objects everytime they are needed.
7. We show benefits of the reuse on a third party application.

We claim that a static analysis framework (like Soot [9]) is better than using AspectJ [23] because, firstly, AspectJ cannot detect the presence of method calls in the code statically. Hence, it cannot be used to skip some portion of the code based on whether a specific method call is present in the rest of the code, whereas, Soot can do this easily. Secondly, it is not possible to intercept assignment statements using AspectJ, hence it cannot be used to perform sophisticated analysis like data flow analysis. Thirdly, using AspectJ, one can only go “around” the code that sends an SMS and cannot remove it from the app’s binary. Moreover, AspectJ does not facilitate code instrumentation in the bytecode. These limitations of AspectJ diverted our attention to Soot.

The purpose of our work is neither to hack the app (as the modified APK would need to be re-signed) nor to infringe copyrights. The work is not targeted to gain monetary benefits, rather it is aimed to show that such performance oriented optimizations are possible by leveraging code injection, without requiring source code of apps. In this work, we also propose a framework which can be used by developers to customize the optimizations provided by Chiromancer and extend them in various ways.

As per our knowledge, there has not been much work done to detect and rectify performance issues in code extracted from APKs automatically. There has been loads of work based on the *object pool* design pattern [8] which requires developers to code in a way such that object reuse is prevalent. However, in this work, we automate the process by transforming the code such that the app recycles objects to an object pool and pulls objects from it whenever there is a need to create more objects of the same kind. In this context, our work elicits a novel approach to detect and deal with performance issues.

In Chapter 2, we discuss the related works. Chapter 3 explains R3’s architecture and our proposed framework. Chapter 5 explain the issues and our approach to tackle them. Chapter 6 deals with the evaluation of R3 on third party apps. We discuss the strengths and weaknesses of our tool in Chapter 7. Finally, in Chapter 8, we conclude and discuss the future work.

Chapter 2

Related Work

2.1 Related Work

Our implementation is based on Soot [9], which is a Java analysis tool which has now been extended to work for Android apps by addition of a component called Dexpler [18]. Dexpler is a modification of Soot and converts Dalvik bytecode directly to Jimple representation. This direct conversion obviates the need for Dalvik bytecode to Java bytecode conversion in order to use Soot. The Dalvik virtual machine is register-based, which means that most of the instructions specify a register that they manipulate. The challenge that the authors of [18] reported to face during Dalvik code to Jimple code conversion was that unlike Dalvik which is untyped, Jimple is typed [19]. Thus, in Dalvik the same register can hold values of different types as against Jimple which needs the type information. The authors use Soot’s fast typing Jimple component that implements a type inference algorithm to infer types of variables.

In [16] the authors present a tutorial on instrumenting Android apps using Soot and Aspect Bench Compiler. This paper takes a running example of an SMS application, that simply sends SMS by accepting a message and a receiver phone number. Using Soot and Tracematches the authors show how SMS spams and premium rate SMSs can be detected and blocked. In our work, similar to the example in [16], we also cater to the SMS to premium rate number problem. We extend it by performing a Flow Insensitive Analysis to see whether the SMS that was sent

to a premium number required an SMS in reply on the basis of which some future events are triggered. If there is no such response awaited, we can simply skip the call to *sendTextMessage()*, otherwise we choose not to skip it.

The work [17] presents a tool that instruments the app’s bytecode directly on the smartphone at runtime (*In-Vivo* approach). This might cause battery to drain much faster. To demonstrate the feasibility of their approach, the authors have provided implementation details for two use cases, namely, *AdRemover*, an advertisement remover and *FineGPolicy*, a fine-grained user centric permission policy system. AdRemover uses a heuristic-based static analysis approach. It checks every try/catch block for functionalities that belong to packages used by the advertisement libraries. If such functionalities are present then code for throwing a user defined exception is inserted at the beginning of the try block. AdRemover assumes that developers place dangerous code inside try/catch blocks; this might not always be true. The FineGPolicy causes invocation of method stubs which increases the overhead. In contrast, our approach relies on conditional skipping of method invocations.

The work [24], presents a tool that performs a very precise analysis for tracking data flows from *sources* to *sinks*. We leverage FlowDroid’s idea of implementing a dummy main method for Android apps so that they can have a starting point for the call graph to be built. Tracking data is essentially a security issue as against our work, which is more focussed on improving performance. In the work [15], researchers have argued over why static analysis techniques score over dynamic techniques for securing Android applications. Generally speaking, runtime analysis techniques incur undesirable time and power overhead.

In [30], the authors introduce a distributed object pool service called DOPS to efficiently manage lifecycles of objects in distributed systems. It takes requests from middlewares, and services these requests keeping in mind efficient utilization of memory. We in contrast use object pools in the context of Android apps. Smartphones are very constrained when it comes to resources and careless allocations of objects can deplete heap space quickly. Hence, the problem becomes even more important on Android devices.

In [11], the authors use object pools to reuse objects allocated to help real time applications achieve predictable running times. The primitives they propose have to be used by developers manually. We on the other hand instrument the applications so that they become self sustained. The aforementioned works provided us useful insights to make apt design choices and motivated us to take up this work.

Chapter 3

R3 Architecture

3.1 R3 Architecture

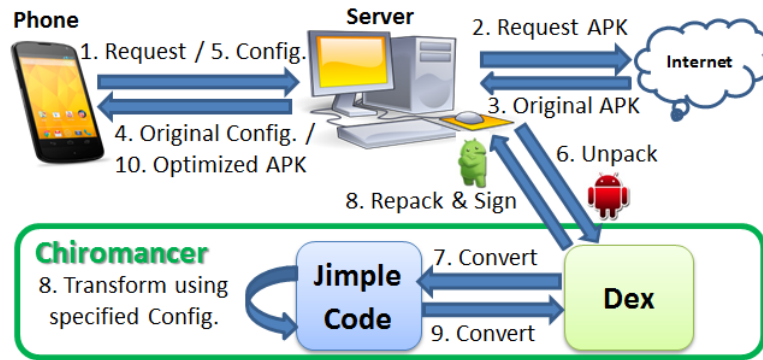


Figure 3.1: R3 Architecture

The architecture of Chiromancer is shown in Figure 3.1. The user places application request (step 1) to our server. The requested app’s APK is then fetched by our server (step 2 and 3) from the Internet or our local repository and statically analyzed to detect probable performance issues. The performance related configuration values originally present in the application are reported back to the user (step 4). The user can then specify the configuration changes she wishes to apply to the app using the GUI (step 5). Chiromancer, then modifies the requested APK based on these inputs and re-signs the modified APK (step 6, 7 and 8 and 9). This APK can be hosted on a server and the link to it can be sent to the person who requested the optimized version (step 10).

The work horse of Chiromancer is Soot’s [9] modification known as Dexpler [18], which takes an APK as input, unpacks it, finds the Dalvik bytecode (.dex) and converts it directly into Jimple, i.e. *Java sIMPLE*’ code (.jimple). Jimple is like Java but is much simpler to analyze (as compared to Dalvik bytecode) primarily because it uses typed variable, a three address internal representation and has only fifteen different types of statements to play with.

Soot’s phases run one by one. The *Transformation phase* executes first. In this phase the application’s Jimple code is analyzed to detect method signatures of interest and the relevant Jimple statements are inserted or parameters modified according to the input configuration. Next, the *Optimization phase* optimizes the code and finally the *Annotation phase* performs built-in analysis. The resultant Jimple code is now performance optimized.

Note that R3 uses the same strategy for optimizing apps with the difference that the analysis in the *Transformation phase* is based on a Summary-based Interprocedural Escape analysis, a Pointer analysis and an analysis that identifies last usage points of reference variables. Creating an object pool in an app, recycling objects to that object pool, and fetching objects from the pool for reuse requires significant amount of code to be instrumented. This makes the extension to Chiromancer an interesting and challenging task.

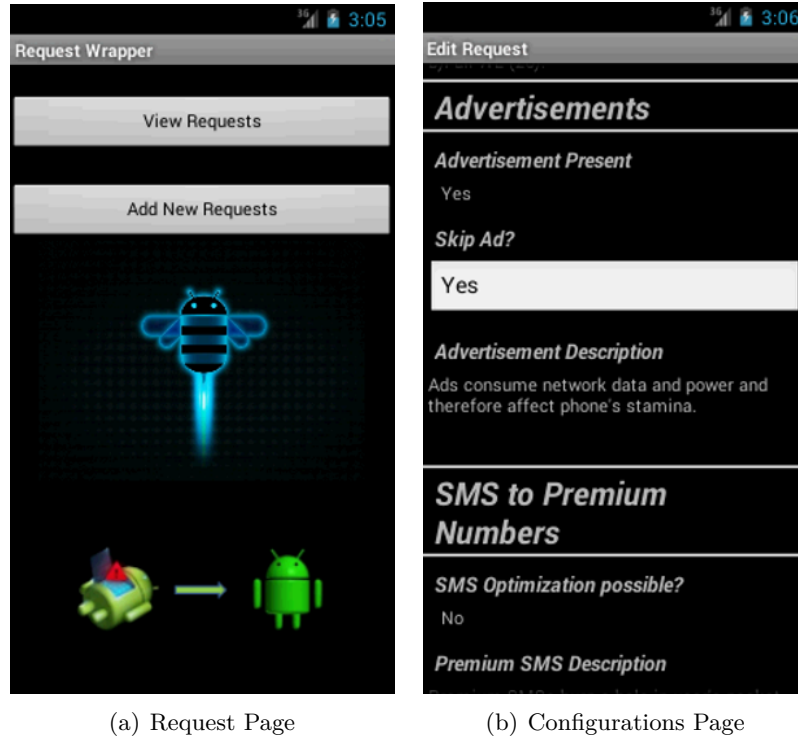


Figure 3.2: GUI of the app

3.2 R3 GUI

Figure 3.2 shows the GUI of our tool. Sub-figure 3.2(a) shows the main page. This page has two buttons - one for placing a new request for optimization and another for viewing the pending requests. The request is placed using the URI of the app on the Google Play store and the corresponding app APK is fetched.

Sub-figure 3.2(b) shows the page where initial configurations for an app are reported back by our tool. Using this page, the user or developer can change the configurations and then send a request for applying the changes. The request is forwarded to our server, where the instrumentation happens and the optimized APK is made available for download.

Chapter 4

Chiromancer Framework

4.1 Chiromancer Framework

We have built a custom framework on top of Soot that provides three levels of abstraction and thus provides an easy way to develop new performance optimizations or extend the existing ones. Figure 4.1 shows these levels of abstraction.

The outermost level provides an interactive GUI-based functionality which enables users to register themselves and get their desired APKs optimized without requiring any knowledge about the internal code. The next two levels modularize the framework and allow the developers to build custom optimizations on top of our API. The multiple abstraction levels relieve the developer from worrying about the complicated code structures that are required to be constructed while instrumenting using Soot.

The middle level of the framework captures high level functionalities that aim to improve domain specific performance. This level comprises of four domain specific modules - GPS Update Frequency Optimizer (GUFO), Wake Lock Optimizer (WLO), Advertisement Remover (AR), and SMS to Premium Number Eliminator (SPNE). These modules correspond to the performance issues which are discussed later in Section 5.1. Together these modules assist the app users and

developers to make effective changes to the apps. For instance, the GUFO module has one of the methods declared as -

void modifyTimeDist(long minTime, float minDist)

Here, *minTime* and *minDist* refer to the minimum time interval (in milli seconds) and the minimum distance interval (in meters) respectively at which the location updates are requested. One can set the desired location update frequency by just passing desired values as arguments to the function call. Likewise, other modules also have several other relevant method declarations. This middle layer would expand accommodating support for more optimizations in the future.

The innermost level of abstraction is typically meant for the developers who wish to develop new performance-oriented optimization modules on top of our API and Soot's API. This level contains the generic core functionalities which can be used across several optimization modules residing at the middle level of the framework. One such functionality used by the GUFO and WLO is -

void mod_Param(Unit unit, int param_idx, < T > param_val)

where the first argument *unit* of type *Unit* [9] is the method whose parameter value at index *param_idx* will be replaced by the value passed in *param_val* having type *T*.

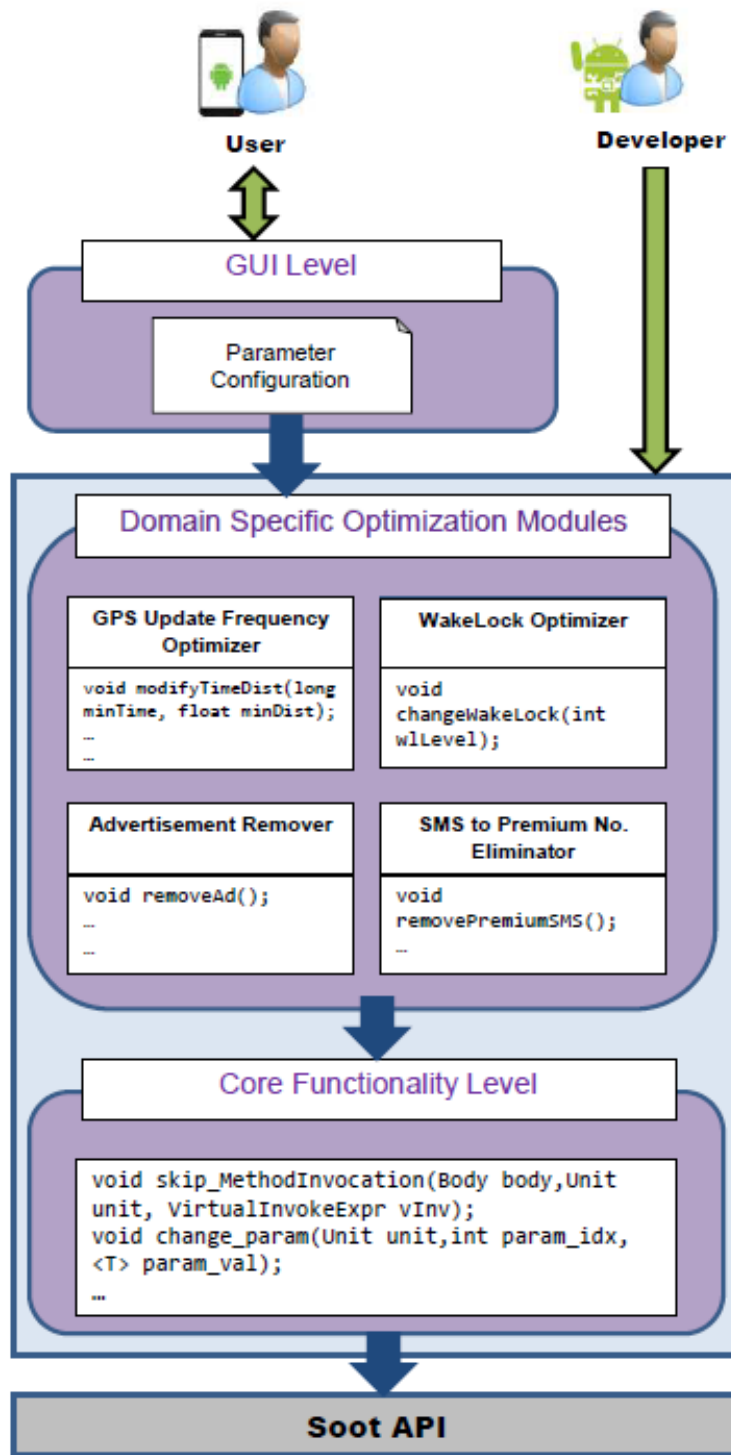


Figure 4.1: Chiromancer Framework - Levels of abstraction

Chapter 5

Our Approach

In this chapter, we describe various performance issues circumscribing energy consumption, data consumption, cost to user, and memory consumption and the techniques we propose to tackle them. The issues have been categorized under two divisions, namely ‘Reduce’ and ‘Reuse and Recycle’.

5.1 Reduce

5.1.1 Advertisements in applications

Most apps these days have advertisement (ad) banners. According to AppBrain [6], Admobs [1] is the most famous API for in-app advertising. These banners fetch ads from the Internet and consume a significant portion of the total network data consumed by the app. This way, the permission to access the Internet can be misused by the developers. Moreover, advertisement libraries also dramatically affect mobile phone’s battery backup. Indeed, third-party advertisements can be held responsible for up to 65%-75% of energy spent in free applications [28].

Solution: In order to avoid consumption of data by advertisements we instrument appropriate code in the app for skipping the call to the `loadAd(AdRequest new AdRequest())` method. Figure 5.1 shows the Jimple code after injection of code to skip an advertisement load call. The alpha-numeric identifiers that begin with a ‘\$’ symbol are local variables generated by Jimple for

```

    $r18 = $r0.<com.lul.accelerometer.AccelerometerMonitorActivity:
                                com.google.ads.AdView adView>;
    $z0 = 1;
    if $z0 == 1 goto label0;

    virtualinvoke $r18.<com.google.ads.AdView:
                                void loadAd(com.google.ads.AdRequest)>(r31);

label0:
    nop;
    r32 = new com.lul.accelerometer.SignalChart;

```

Figure 5.1: Advertisement application Jimple code after injection

adhering to its three address code representation. The users, prior to installation, can decide whether they wish to see any advertisements and can then proceed to install the application.

In contrast, the authors of [17] propose two use cases - *AdRemover* and *FineGPolicy*. As mentioned in section 2.1, *AdRemover* assumes that developers place potentially dangerous code in try/catch blocks, however, this might not always be true. *FineGPolicy* causes invocation of method stubs instead of actual methods for those methods for which permission has not been granted. This increases the overhead because the method’s local variables and other context related information has to be encapsulated in a stack frame and stored on the call stack.

Furthermore, in [17] the authors instrument the app’s bytecode directly on the smartphone. This could drain the battery very quickly and hence is not advisable. Our technique can be extended to work for other advertisement libraries also by adding their respective load advertisement method signatures.

5.1.2 SMS to premium rate numbers

A very common issue that incurs cost is sending SMSs to premium rate numbers. Many applications ask the permission for sending SMSs. Unfortunately, there is no restriction built in the security model of Android that checks whether the target number is a premium rate number or not.

Solution: To avoid exorbitant expenditure we inject application code to skip execution of the *sendTextMessage(String destinationAddress, String scAddress, String text, PendingIntent*

sentIntent, *PendingIntent deliveryIntent*) method if the recipient’s number is a premium rate number, just like the authors have shown in [16]. Figure 5.2 shows the code after injection of an if clause in the Jimple file to check whether the number is a premium rate number. The code is injected irrespective of whether the application will send SMS to a premium rate number or not. Then, during runtime depending on the recipient’s phone number entered by the user, the SMS is either sent or blocked, i.e., the control would skip over the code that sends the message, only if the target number begins with “0900”, and resume execution from a *nop* statement identified by *label 0*.

We further extend this implementation to take care of a situation where an SMS to a premium number is sent and a response is awaited as a reply. This might happen when some other task depends on the content that is sent back as a response. In such a case we cannot skip the execution of *sendTextMessage()* method because then it would break the app. This entails implementing a Flow Insensitive Analysis that checks whether the code has a *createFromPdu(byte[] pdu)* method which converts a Protocol Data Unit (PDU) to an *SmsMessage* [7] object. This confirms that the message contents will be used further.

We use Flow Insensitive Analysis because in Android apps, Activities and Services can be directly called from any other activity even if it belongs to a different app. This results in Android apps to have multiple entry points. Hence, it is difficult to accurately predict whether the *createFromPdu()* method comes before or after the *sendTextMessage()* method call. It is evident that such conditional skipping of method calls would not have been possible by using AspectJ because of its inability to discover a call to *createFromPdu()* when it detects a *sendSms()* *Join Point* [23].

```

$R1 = interfaceinvoke $R10.<android.text.Editable: java.lang.String toString()>();
$Z0 = virtualinvoke $R2.<java.lang.String: boolean startsWith(java.lang.String)>("0900");
if $Z0 == 1 goto label0;

virtualinvoke $R9.<android.telephony.SmsManager: void sendTextMessage(java.lang.String,
java.lang.String,java.lang.String,android.app.PendingIntent,android.app.PendingIntent)>
($R2, null,$R1, $R3, null);

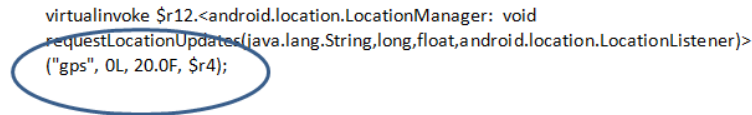
label0:
nop;
return;

```

Figure 5.2: Premium rate SMS application Jimple code after injection

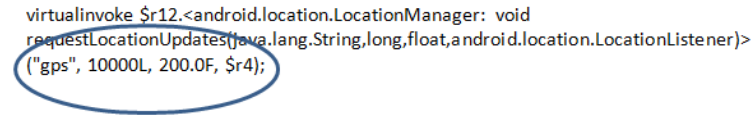
5.1.3 GPS location update frequency is too high

Global Positioning System, now common in smartphones, used to accurately locate a user, is an accomplice in draining battery. GPS requires communication with three to four satellites to accurately locate a device. There is no time division during the communication and therefore the antenna needs to be powered all the time during the communication period. This prevents the phone from going into a sleep state. Matters get even worse when the location updates are required very frequently with short time gap and short difference in distance. Short location update intervals are often unnecessary and cause significant battery drain.

A screenshot of Jimple code. The code consists of two lines: `virtualinvoke $r12.<android.location.LocationManager: void requestLocationUpdates(java.lang.String,long,float,android.location.LocationListener)>` and `("gps", 0L, 20.0F, $r4);`. The second line is circled in blue.

```
virtualinvoke $r12.<android.location.LocationManager: void  
requestLocationUpdates(java.lang.String,long,float,android.location.LocationListener)>  
("gps", 0L, 20.0F, $r4);
```

Figure 5.3: GPS Location application Jimple code before injection

A screenshot of Jimple code, similar to Figure 5.3 but with modified values. The second line, `("gps", 10000L, 200.0F, $r4);`, is circled in blue.

```
virtualinvoke $r12.<android.location.LocationManager: void  
requestLocationUpdates(java.lang.String,long,float,android.location.LocationListener)>  
("gps", 10000L, 200.0F, $r4);
```

Figure 5.4: GPS Location application Jimple code after injection

Solution: We let the users decide the parameters to be passed to the *requestLocationUpdates(String provider, long minTime, float minDistance, LocationListener listener)* method. Users know better whether they will be travelling on foot, by train, or by car. Therefore it becomes necessary to give the users the freedom to decide the location update intervals. The users can input the frequency at which they want the application to make location updates. This value is replaced in the parameter of the *requestLocationUpdate()*. Figure 5.4 shows the Jimple code after parameter replacements. The parameters were 0 milli second and 20 meters initially and were changed to 10,000 milli seconds and 200 meters respectively after optimization.

Table 5.1: Wake lock levels and component On/Off information

C.V.	Flag Value	CPU	Screen	K.B.
1	PARTIAL_WAKE_LOCK	On*	Off	Off
6	SCREEN_DIM_WAKE_LOCK	On	Dim	Off
10	SCREEN_BRIGHT_WAKE_LOCK	On	Bright	Off
26	FULL_WAKE_LOCK	On	Bright	Bright

Here:

C.V. - Constant value representing different wake locks

K.B. - Keyboard

*Note that if you hold a partial wake lock, the CPU will continue to run, regardless of any display timeouts or the state of the screen and even after the user presses the power button. In all other wake locks, the CPU will run, but the user can still put the device to sleep using the power button.

5.1.4 Inappropriate wake lock acquired

The wake lock mechanism informs an application about the duration for which the device should keep its screen, CPU or keyboard active. The screen consumes maximum battery in Android phones. Careless use of this API can drain the battery quickly. Various wake lock levels along with their corresponding constant values that are taken as the first parameter by the *PowerManager.WakeLock newWakeLock(int levelAndFlags, String tag)* method are shown in Table 5.1. The table also shows the corresponding components that remain on or are allowed to switch off when one acquires a particular wake lock.

```
$r13 = virtualinvoke $r12.<android.os.PowerManager>; android.os.PowerManager$WakeLock
newWakeLock(int,java.lang.String)>(6, "Stopwatch");
```

Figure 5.5: Wake lock application Jimple code before injection

```
$r13 = virtualinvoke $r12.<android.os.PowerManager>; android.os.PowerManager$WakeLock
newWakeLock(int,java.lang.String)>(1, "Stopwatch");
```

Figure 5.6: Wake lock application Jimple code after injection

Solution: We allow the users to decide which wake lock they want the application to acquire. Then we replace the parameter of the *PowerManager.WakeLock newWakeLock()* by the one supplied by the user. For instance, a user can replace a *FULL_WAKE_LOCK* by *PAR-*

TIAL_WAKE_LOCK [7] to save energy. Figure 5.5 and Figure 5.6 respectively show Jimple code before and after replacing the original wake lock level parameter by the one supplied by the user. Here, “1” denotes *PARTIAL_WAKE_LOCK* and “6” denotes *FULL_WAKE_LOCK*.

5.2 Reuse and Recycle

5.2.1 Inefficient Heap space utilization

When a large number of short-lived objects are created and left at the discretion of garbage collector to be collected, memory and as a result end users suffer. The allocated memory is held until garbage collection spins into action. When it does get invoked, the garbage collector introduces pauses which result in jitters. This commonly happens when generating animation graphics, sounds, or performing some repetitive tasks. Figure 5.7 shows a screenshot from a famous game called ‘Hill Climb Racing’, which shows multiple stone-like objects that get created whenever the tyres of the vehicle rub against the surface of the road. These are short-lived objects that get created and discarded quickly in large numbers and thus impose a heavy overhead on the Dalvik garbage collector.



Figure 5.7: Hill Climb Racing: Short-lived objects

Large number of object allocations also cause fragmentation and might require frequent compaction. If such objects encapsulate a resource like a database or network connection, which is expensive to acquire, there will be a greater performance hit. As a result the user experience suffers. We propose that these short-lived objects must be shared with other apps so that they

can be reused. We advocate recycling objects to fixed size object pools specific to every class. This is also referred to as the Object Pool design pattern [8] in literature.

The pool size should ideally be adjusted based on the app. However, it should remain constant for an app otherwise increasing/decreasing the pool data structure size requires data to be copied and as a result the idea of predictable execution time goes for a toss. We use fixed size arraylists as object pools and create separate arraylists for every class of objects except Java library classes¹.

If we do not use separate arraylists for every class and store different sized objects in the same pool then this would cause space to be wasted because each slot in the arraylist would have to be of the size of the largest sized object. Furthermore, if we use a common arraylist for all classes of objects then a large number of allocation of one kind of object will fill up the arraylist and leave no room for pooling other objects.

From the pool of recycled objects, the objects are picked according to the *last in first out* policy and an index variable helps to mark the current size of the pool. All objects in the arraylist pool from index 0 to the integer value of the index variable are free to be reused. However, before being picked from the pool for reuse the field of the objects would have to be re-initialized appropriately. The pool size should not be too large nor too small. Big pools will hold too many objects all of which might not be reused and small pools will hold too less and the app might have to fall back on the usual way of allocating objects using the *new* statement.

Evidently, there is a space-time tradeoff involved, i.e. to save time for re-allocation of these objects, we are preventing them from getting garbage collected by caching them in an object pool created in the RAM. Note that we cannot store the object pool in the storage (secondary memory) of the device because the access time required to a pool in the storage of the device would increase the jitters instead of reducing them.

In order to identify shareable objects we target loop constructs since loops can cause objects to be created in bulk. We leverage Escape analysis [3] to figure out if these objects are short lived. If these objects escape the scope of the method then there is a high possibility that they will be

¹Java library class objects are not pooled because the resetting of their fields requires prior knowledge of the default field values. If such default values are available we can add support for Java library class objects as well.

used ahead, in such a case we do not add them to a pool or create a pool for them if it does not exist. If they do not escape, then we can recycle them to a pool specific to that class of objects or create a fixed size pool if the pool does not exist. We use a Context Sensitive Interprocedural Summary based Escape analysis for the purpose of identifying whether a particular variable escapes or not.

After identification of the poolable objects (those that do not escape the scope of the loop), we perform an analysis, which we call ‘LUP Analysis’, to identify the ‘Last Usage Point’ of these objects. The last point of usage is precisely where we instrument the code to release the objects to the object pool.

Please note that there is also a need for performing a Points-to analysis [33] before the Escape and LUP analysis. This is because two or more different reference variables (forming an alias set) can refer to the same object. In the case of Escape analysis, even if one of the variables in the alias set escapes then the object would have escaped. In the case of LUP analysis, the last usage of all the variables in the alias set needs to be identified. The variable that is used last among them will indicate the target instrumentation site. Hence, the use of all reference variables in the set would have to be tracked. We use Soot’s built-in component called Soot Pointer Analysis Research Kit (SPARK) framework [14] for performing a subset based Points-to analysis based on the Anderson’s algorithm [29].

5.2.2 Heuristics

We also apply certain heuristics to prune the set of objects we would like to reuse. This feature is optional and can be used for large apps so that focus of the optimization remains on objects that are created in large numbers. The heuristics we use are as follows:

1. We determine whether objects of a specific type actually need to be stored a pool. If new objects of the same type are never created again in the app, then there is no need for storing them in a pool of their own. Infact, there is no need for creating a pool for them at all. ²

²In Android, due to the event driven nature of apps, certain callback methods like *onClick()*, *onLongClick()*, *onFocusChange()*, etc. can be called multiple times based on appropriate user events. Hence, even if object

2. Objects created directly within loops are given priority.
3. Objects created inside methods which are called from within a loop are given priority.
4. Method that has one or more allocation statements and has a large *betweenness centrality* [31]³ in the call graph is given priority.

5.2.3 Modules

The overall process for enabling object reuse comprises of four modules, namely *Interprocedural Escape Analysis Module*, *Poolable Sites Identifier Module*, *Last Usage Point Identifier Module* and *Bytecode Instrumentation Module*.

To illustrate the working of the first two modules we consider the following code snippet from our custom made app known as `SlideShow`. The app is available at [5] for reference. This app presents a slide show of 9 images by picking the images one at a time and displaying them in an *ImageView* [25] 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      }

```

allocation happens once in such callback methods, they might be called multiple times during the lifetime of the app. For optimizing such allocations one should not use this heuristic.

³Betweenness 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.

```

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 5.1: Code snippet from SlideShow app

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

5.2.3.1 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 [24]. 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 5.8 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*.

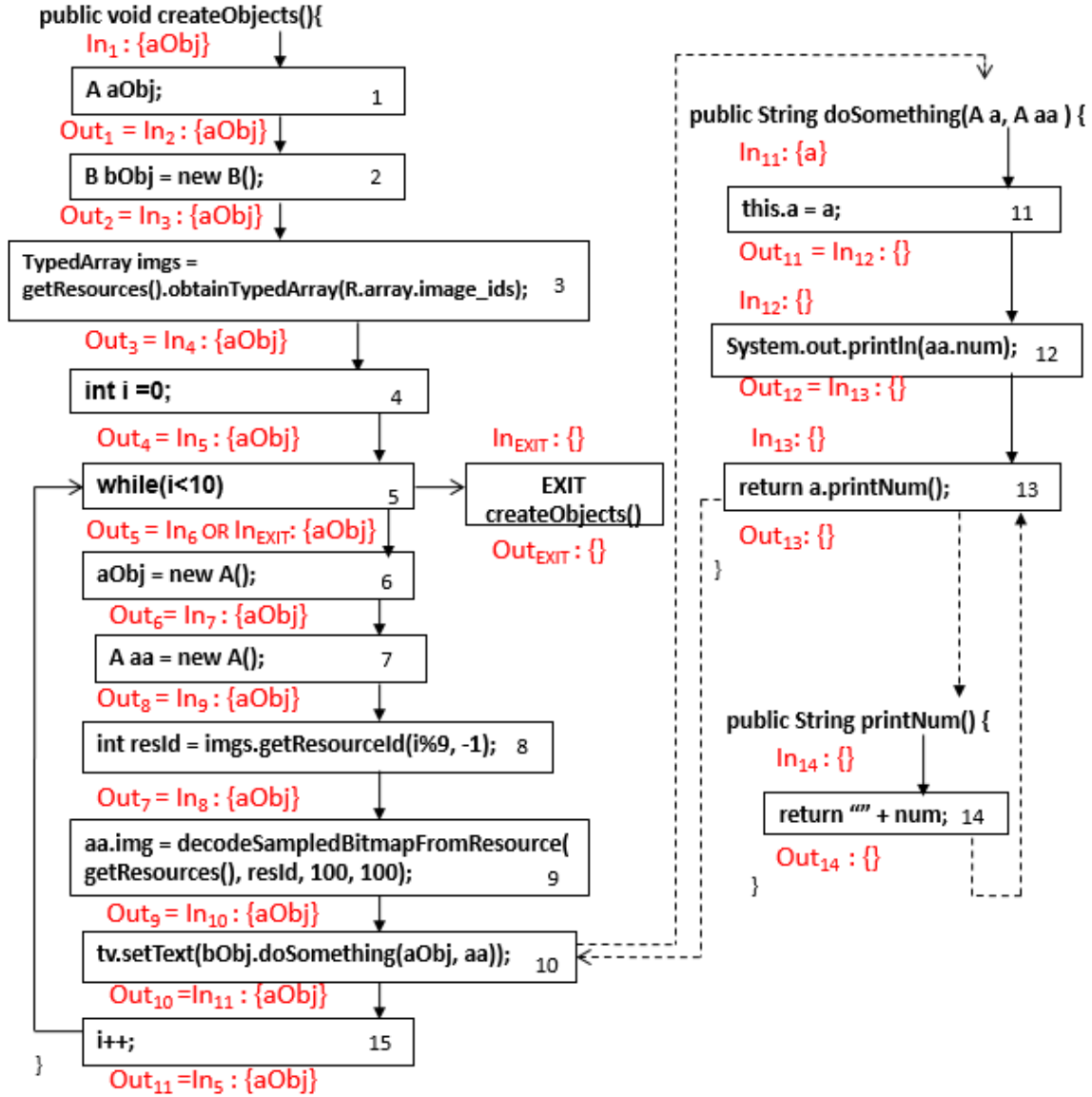


Figure 5.8: Call graph overlaid on a CFG and Escape Analysis for SlideShow app

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 [9]. 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 summary of a method can be computed using solely 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 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 [26] 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 *may* 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* [9] in a backward manner until a *fixed point* [26] is reached.

The flow functions for IEA are expressed as

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

$In_i = Out_i \cup Esc_i$; where

$$Esc_i = \begin{cases} \{y\} \mid S_i: x = y & \text{if } x \text{ is static/instance field} \\ \{y\} \mid S_i: x = y & \text{if } x \text{ has already escaped} \\ \{y\} \mid S_i: x.func(y) & \text{if } x \text{ is a (or alias of) field and } x \in \text{java.util.Collection} \\ \{p_i\} \mid S_i: x = f(p_1, p_i) & \text{if } p_i \in \text{summary}_f \\ \phi & \text{otherwise} \end{cases}$$

Here, Esc_i corresponds to the set of variables that have escaped due to the presence of code statement S_i in a method. The Esc_i acts as the *Gen* set and there is no *Kill* set required for our analysis. If f is a method, then “summary_f” denotes the summary of method f . Those variables which are added to a field that belongs to the *java.util.Collection* package (such as Set, List, Map, Vector etc.) using the *add()*, *addAll()*, *put()*, *putAll()*, etc. methods are also added in the Esc_i set. The method *func()* used in the definition of Esc_i set, thus may refer to any java library method which is responsible for adding an object to a collection object. Figure 5.8 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.

5.2.3.2 Poolable Sites Identifier Module (PSIM)

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 [9] 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 the method and hence the objects generated at such an allocation site can be safely pooled for reuse. We mark all such identified objects as “poolable objects” and refer to such allocation sites as “poolable sites”.

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.

The *createObjects()* method in Figure 5.8 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 poolable.

5.2.3.3 Last Usage Point Identifier Module (LUPIM)

After getting the poolable sites from the PSIM, we track the object reference variables that refer to the poolable objects using our LUP analysis. This analysis uses Points-to analysis to figure out which set of reference variables can possibly point to the same object. Leveraging this information it tracks program points where these variables are used and finds the last point of use for a particular poolable object. We refer to such a point in the program as “recyclable site”.

The strategy works by creating “alias sets” containing reference variables that point to the same poolable object. Thus, we get a cluster of such sets; one set for every poolable object. Now, we apply a modification of *Live Variable analysis* [32], which we call LUP analysis. For each program point inside a method, LUP analysis determines the variables that *must* have a subsequent use on *some* path from that point.

The analysis uses the following flow functions.

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

$In_i = Out_i \cup Gen_i$; where

$$Gen_i = \begin{cases} \{var(y)\} & \text{if } S_i: x=y \\ \{var(y)\} & \text{if } S_i: x=\text{if}(y) \\ \{var(y)\} & \text{if } S_i: x=\text{while}(y) \\ \{p\} & \text{if } S_i: x=f(p) \\ \phi & \text{otherwise} \end{cases}$$

Here, y is an expression and $var(y)$ means variables evaluated in the expression y . Using the aforementioned flow functions we find the first program point in the control flow where all variables in a particular alias set of an object become *dead*. This will be the point where we can insert code to recycle that object (recyclable site). The technique to do this is as follows.

In the control flow graph, we search for statements which have none of the variables belonging to a particular alias set in their *In* sets and *Out* sets. We collect all such statements in a set A.

Now we find successors of each of these statements and store them in another set B. Now the set difference between set A and set B gives the first such statement whose *In* and *Out* sets are devoid of any variables present in a particular object’s alias set. We instrument code before this statement and call this program point as the “recyclable site”.

Note that unlike *Live Variable analysis*, the *Kill* set is not present in the flow functions. This is because the Points-To analysis performed by Spark is a flow insensitive analysis and if a reference variable is re-defined in the code to refer to a new object then the points-to set of the reference variable will have both the objects. Hence, even if one of them escapes then the variable will be present in the summary for that method and as a result both the objects will not be recycled back to the pool. Thus, we will miss an opportunity to optimize, which is due to a weakness (flow-insensitivity) of Spark.

If present, the *Kill* set for a particular statement would have contained the variables that are re-defined to refer to some newly allocated object on that statement. If *Kill* set is added to the flow sets with this weakness unresolved, then the instrumentation point detected by LUP could be wrong and the reason is the flow-insensitivity of Spark. Hence, we hope that Spark developers would make it flow-sensitive in the future and then we would be able to catch more cases for optimization and the correct instrumentation points in case the same reference variable is used to refer to multiple newly allocated objects.

In order to show the working of LUP analysis in conjunction with a Points-to analysis to identify “recyclable sites” we consider a third party app available at [21]. Figure 5.9 illustrates the process of identification of recyclable sites. The *In* and *Out* sets, in accordance to the flow functions mentioned above, are shown for every statement of the *testPool()* method. The Points-to analysis will create the alias set {point, points[i]}. Now both the elements of this alias set are not present in the *In* and *Out* sets of statements 8, 9, 10, 11, and 12. The set of these statements is the set A. The set formed by taking successors of each of these statements is {9, 10, 11, 12}. Taking the set difference (A - B), we get 8 which is the first statement which has no element from a particular alias set in its *In* and *Out* sets. This will be our “recycle site” and we instrument statements for recycling the object just before this statement.

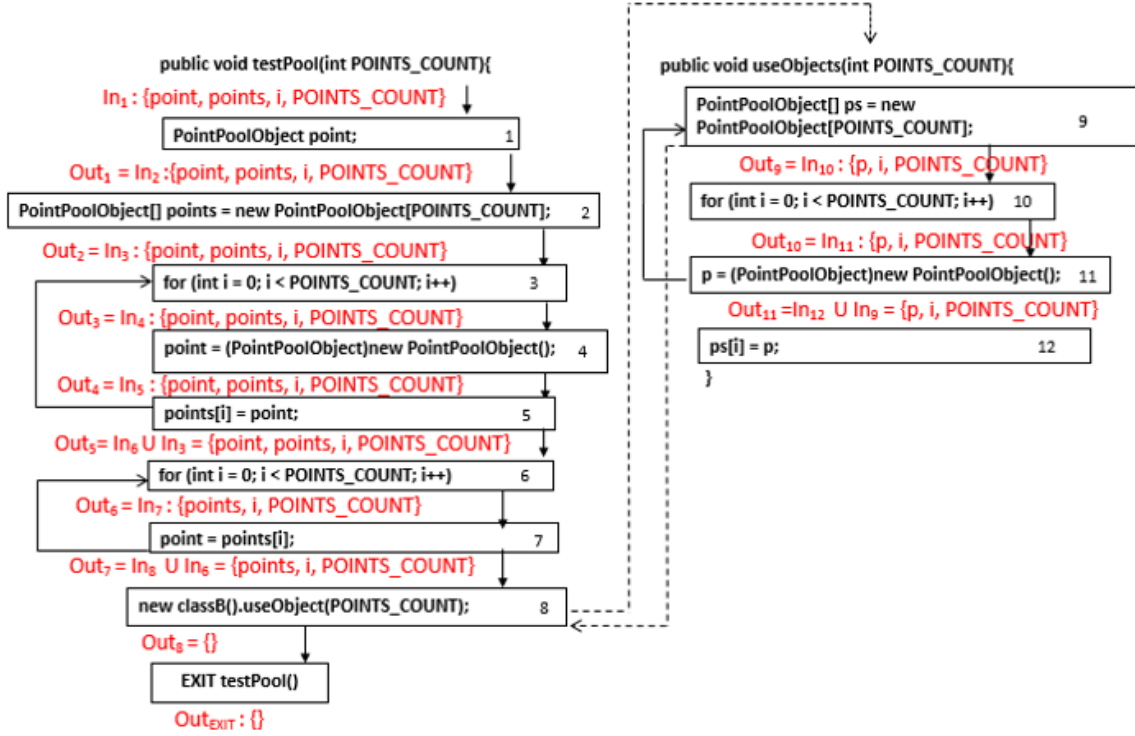


Figure 5.9: CFG of testPool() method and LUP analysis on SlideShow app

5.2.3.4 Bytecode Instrumentation Module (BIM)

Once the recyclable sites have been identified, we move into the instrumentation phase. We leverage the template, available at [21], suggested by Andrea Bresolin for implementing the Object Pool design pattern. However, we have made significant changes to this template for making it fit for our purpose. The app at [21] was originally developed for manifesting object reuse by leaving the ownness of detecting opportunities for object reuse on the developer and the developer would have to manipulate code to reuse objects. However, in our work we automate the entire process and relieve the developer from this arduous task. This automation necessitated changes to the template suggested at [21]. The instrumentation is done according to the following template.

We instrument the original class's definition to add initialization functionality for initializing the objects that will be reused. The *initializePoolObject()* method is invoked when an object is retrieved from the pool. This method resets the field of the object using values from the object's constructor. Multiple *initializePoolObject()* methods get created if there are multiple constructors

defined in the class. We simulate the code of the constructor by copying statements from it. However, we do not copy the *java.lang.Object*'s *init()* method call because this is responsible for allocating space to the object. Some additional features of the reset functionality are as follows:

- (a) Those fields that are of primitive data type and are not initialized in the constructor are assigned default values as per the Java compiler.
- (b) *StringBuilder* fields are emptied so that they become empty strings.
- (c) Arrays of primitive types are filled with their default values.
- (d) Collections are cleared.
- (e) Those fields which are not initialized in the constructor and are of a user defined type are initialized with *null* if and only if they are not allocated space in the constructor using the *new* statement. However, if they are allocated space, then we call their respective *initializePoolObject()* methods.
- (f) We do not reset values of static and final instance fields because static fields retain their values as their scope is the scope of the entire class and final fields cannot be re-initialized.
- (g) We initialize the variables that are inherited from a parent class which is one level higher in the inheritance hierarchy using the constructor of the parent class. Those variables that are inherited from Android library classes are not initialized because of the unavailability of the constructor's implementation. The implementation of Android library class constructors is not present in the android.jar, rather the methods/constructors are only defined as stubs. Hence, we try to synthesize *initializePoolObject()* method on a best effort basis since a precise resetting of the fields of an object would require the source code of the Android API, which currently resides on the devices only and is not a part of the android.jar.

We also instrument an *ObjectPool* class which has one parameterized constructor - *ObjectPool(int maxSize)* this constructor takes as parameter the maximum size of an object pool for a particular class that one wants to use. Note that the size will remain fixed throughout the life of the app. The *ObjectPool* class has three other methods, namely:

1. *hasObject()* - which checks whether there is any object left in the pool so that it can be reused
2. *getObject()* - which fetches an object from the object pool
3. *freeObject(Object obj)* - which stores the object *obj* in a pool if the maximum limit of the pool is not exceeded

In case the object pool is empty, then objects are created using the *new* statement. Hence, our approach falls back on the regular way of allocating objects in case none is available in the pool.

We instrument a *BufferPool* class which has a constructor that instantiates a `HashMap<String, ObjectPool>`. There will be one object pool for every class. Hence, we create a hashmap for storing a pool corresponding to every class. The *BufferPool* class has three other methods, namely

1. *isObjInPool(String c)* - which returns true if a pool exists for a particular class and it has atleast one object. It calls *hasObject()* on the pool object internally to accomplish this.
2. *getObject(String c)* - which fetches the pool corresponding to class 'c' and calls *getObject()* on it
3. *saveObject(String c, Object o)* - which checks whether a pool exists corresponding to class 'c' and creates a pool if it does not exist and adds the object 'o' to it by calling *freeObject(o)*, otherwise it only adds an object to the pool corresponding to class 'c' by calling *freeObject(o)*

Thus, if the object pool already exists it is not created again and the objects are released to the pool straight away after the recyclable site. Moreover, both the *getObject()* and *freeObject(Object obj)* methods are *synchronized* so that consistency is maintained when multiple threads try to access the pool.

The app at [20] is a minor modification of the app available at [21]. The difference is that the *PointPoolObject* has been made heavier by the addition of an array field and the third loop in

testPool method has been moved to a method in a separate class to show the benefits of R3 across classes. Rest of the app is the same.

The app [20] creates multiple instances of the *PointPoolObject* class, which extends the *Point* class, in a loop. The *testPool()* method of the *MainActivity* class and *useObjects()* method of the *classB* class from the unoptimized version of the app are shown below. In the first for loop of *testPool()* method, *PointPoolObject* class objects are created and stored in a loop. In the second for loop, the app iterates over the array to fetch the *PointPoolObject* objects from it and uses them. Finally, there is a call to the *useObjects()* method of *classB* class in which the re-allocation of *PointPoolObject* class objects takes place.

```
1
2  public class MainActivity extends Activity implements OnClickListener
3  {
4      //Appropriate field declarations
5      //onCreate() method definition
6
7      @Override
8      public void onClick(View v)
9      {
10         if (v == testExceedPoolBtn)
11         {
12             testPool( 1000, testExceedPoolText);
13         }
14     }
15
16     private void testPool( final int POINTS_COUNT, TextView textView)
17     {
18         PointPoolObject point;
19         PointPoolObject[] points = new PointPoolObject[POINTS_COUNT];
20         Debug.startAllocCounting();
21
22         for (int i = 0; i < POINTS_COUNT; i++)
23         {
```

```

24     point = new PointPoolObject();
25     points[i] = point;
26 }
27
28 for (int i = 0; i < POINTS_COUNT; i++)
29 {
30     point = points[i];
31     //no pool storing happens here
32 }
33 new classB().useObjects(POINTS_COUNT);
34
35 Debug.stopAllocCounting();
36 int ac = Debug.getThreadAllocCount();
37 textView.setText(ac + " ac");
38 }
39 }
40
41 public class classB {
42     public void useObjects(int POINTS_COUNT)
43     {
44         PointPoolObject[] ps = new PointPoolObject[POINTS_COUNT];
45         PointPoolObject p;
46         for (int i = 0; i < POINTS_COUNT; i++)
47         {
48             p = new PointPoolObject();
49             ps[i] = p;
50         }
51     }
52 }

```

Listing 5.2: Code before optimization

We manifest object reuse by recycling to the pool object instances (created in the first loop) after their last usage (in the second loop) and reusing them later when required in the third loop. The optimized version of the *testPool()* method and *useObjects()* method is shown below.

```
1  public class MainActivity extends Activity implements OnClickListener
2  {
3      //Appropriate field declarations
4      //onCreate() method definition
5
6      @Override
7      public void onClick(View v)
8      {
9          if (v == testExceedPoolBtn)
10         {
11             testPool( 1000, testExceedPoolText);
12         }
13     }
14
15     private void testPool(final int POINTS_COUNT, TextView textView)
16     {
17         PointPoolObject point;
18         PointPoolObject[] points = new PointPoolObject[POINTS_COUNT];
19         Debug.startAllocCounting();
20         BufferPool buffer = MainActivity.buffer;
21         for (int i = 0; i < POINTS_COUNT; i++)
22         {
23             String c = "PointPoolObject";
24
25             if(buffer.isObjInPool(c))
26             {
27                 point = (PointPoolObject) buffer.getObject(c);
28                 point.initializePoolObject();
29             }
30             else
```

```

31     {
32         point = new PointPoolObject();
33     }
34
35     points[i] = point;
36 }
37 for (int i = 0; i < POINTS_COUNT; i++)
38 {
39     point = points[i];
40     String c = "PointPoolObject";
41     buffer.saveObject(c, point);
42 }
43 new classB().useObjects(POINTS_COUNT);
44
45 Debug.stopAllocCounting();
46 int ac = Debug.getThreadAllocCount();
47 textView.setText(ac + " ac");
48 }
49 }
50
51 public class classB {
52     public void useObjects(int POINTS_COUNT)
53     {
54         PointPoolObject[] ps = new PointPoolObject[POINTS_COUNT];
55         PointPoolObject p;
56         for (int i = 0; i < POINTS_COUNT; i++)
57         {
58             String c = "PointPoolObject";
59             if(buffer.isObjInPool(c))
60             {
61                 p = (PointPoolObject) buffer.getObject(c);
62                 p.initializePoolObject();
63             }

```

```
64     else
65     {
66         p = new PointPoolObject();
67     }
68     ps[i] = p;
69 }
70 }
```

Listing 5.3: Code after optimization

As can be observed from the aforementioned code, on line 41 the optimized app now recycles objects to the object pool after their last usage. On lines 25-29 and 59-63, before creating new objects of a specific type the pool is checked for pre-existing objects of that type. If the same type of objects are present in the pool they are reused after appropriate re-initialization using the *initializePoolObject()* method. In case there are no objects of the same type available for reuse in the pool then the app falls back on the *new* allocation statement as is evident from line 32 and 66.

The *initializePoolObject()* method is instrumented in the class of the object. In the sample app, the code that was instrumented for the *initializePoolObject()* method is as follows. It initializes the primitive integer fields inherited from *Point* class to 0.

```
1  public void initializePoolObject()
2  {
3      x = 0;
4      y = 0;
5  }
```

Listing 5.4: Code after optimization

Chapter 6

Evaluation

6.1 Evaluation Overview

In this section first we elicit the improvements that Chiromancer yields using three third party applications. Then we elicit the benefits of the extension (memory optimization) to Chiromancer by showing memory utilization results from another third party application.

We discuss the testbed for experimenting with Chiromancer and its extension. Then we discuss the methodology for collecting data and manifest the results from the experiments conducted on the third party applications.

6.2 Testbed

The applications - `Accelerometer Monitor`, `Swing Ball`, and `GPShake Lite` were chosen because they had advertisements, dim wake lock [7], and very frequent GPS location updates respectively. We performed our experiments with these three applications on two Android phones-

- (a) Sony Xperia P LT22i (dual-core, 1 GHz Cortex-A9 processor, 1 GB RAM, Android version 4.1.2)
- (b) HTC Explorer (single-core, 600 MHz Cortex A5 processor, 512 MB RAM, Android version 2.3)

For **Accelerometer Monitor** application, which has an advertisement, we are concerned only with the network data consumption, whereas for **Swing Ball** and **GPSHake Lite**, only energy consumption is considered.

Xperia P had screen brightness set to “Adapt to lighting conditions” and the display timeout set to 1 minute. On Explorer, the brightness was set to 50% and the timeout was set to 10 minutes. All the readings for **Swing Ball** application were taken in the same light conditions.

For the memory optimization we considered the app available at [21] and performed the experiments on Moto G. As mentioned earlier in Section 5.2, we illustrate the reuse of *PointPoolObject* objects created by the app in a loop. The app, before instrumentation, can be downloaded from [20] and after transformation the app looks like the one available at [27]. The code that is instrumented during the transformation is as mentioned in Section 5.2.

6.3 Methodology of Collecting Results

We ran two versions (unoptimized and optimized) of the same application on the phones and collected energy and data consumption readings (whichever appropriate). The optimized (OPT) and unoptimized (U-OPT) versions of the application were both run ten times for a duration $T = 180$ seconds each. The energy and data consumption were measured by using **PowerTutor** [4] and **Traffic Monitor** [4] respectively. Table 6.1 tabulates the data and energy consumption statistics obtained from both the phones for the three applications.

Accelerometer Monitor [2] application has an ad from Google’s AdMob [1], which forms a significant portion of the total network data consumption. Our tool generates an optimized version of this application by skipping the *loadAd()* method calls. The statistics in Table 6.1 show that data consumption for this application reduced by a factor of 12.2 for Xperia P and by a factor of 33.94 for Explorer.

Similarly, **Swing Ball** application [2] was used to show improvements achieved by wakelock manipulation. This application originally had *SCREEN_DIM_WAKE_LOCK* [7] which was

Table 6.1: Chiromancer Evaluation Results

Application	Version	Metric	Sony	HTC
Accelerometer Monitor	U-OPT	D_{avg}	20.4	17.99
		R	10	10
		S.D.	2.57	0.43
	OPT	D_{avg}	1.67	0.53
		R	10	10
		S.D.	0.97	0
Swing Ball	U-OPT	E_{avg}	114.84	176.02
		R	10	10
		S.D.	3.31	4.01
	OPT	E_{avg}	60.88	73.5
		R	10	10
		S.D.	3.13	1.6
GPSHake Lite	U-OPT	E_{avg}	1.98	1.42
		R	9	9
		S.D.	0.28	0.14
	OPT	E_{avg}	0.53	1.02
		R	9	9
		S.D.	0.15	0.05

Here:

U-OPT : Un-optimized and OPT : Optimized

E_{avg} : Average energy consumed by the application in Joule where the average is over R readings of E

E : Total energy consumed by application in Joule in time T

D_{avg} : Average network data consumed by the application in KB where the average is over R readings of D

D : Total network data consumed by application in KB in time T

R : Number of readings over which E_{avg} and D_{avg} are calculated

S.D. : Standard Deviation of the R readings

changed to *PARTIAL_WAKE_LOCK* [7] after optimization. The total energy consumed, E is recorded as

$$E = E_{CPU} + E_{LCD} + E_{3G}$$

where E_{CPU} , E_{3G} and E_{LCD} denote the energy consumed by CPU, 3G service usage and device screen respectively in time T. Statistics from Table 6.1 clearly show that energy consumption for **Swing Ball** application reduced by a factor of 1.88 in Xperia P and by a factor of 2.39 in Explorer.

Finally, **GPSHake Lite** application [2] was used to show the improvement achieved after reducing the location update frequency to 200 meters or 10 sec from 20 meters or 0 sec respectively. The total energy consumed, E is given by

$$E = E_{CPU} + E_{3G}$$

where E_{CPU} and E_{3G} represent the energy consumed by CPU and 3G service respectively for time T. The results in Table 6.1 show a decrease in energy consumption by a factor of 3.7 on Xperia P and by a factor of 1.39 on Explorer. GPS consumes considerably more time and energy for the first location fix. Hence, the first reading, being an outlier, has been ignored and the average has been taken over the remaining nine readings. The improvements offered by Chiromancer are evident from these experiments.

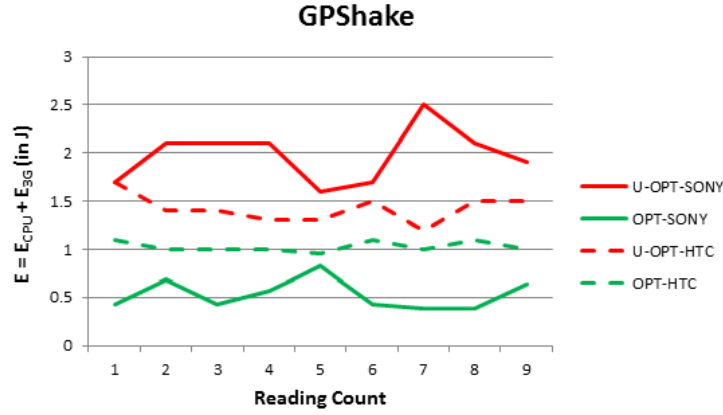


Figure 6.1: GPShake Lite results OPT v/s U-OPT for both phones

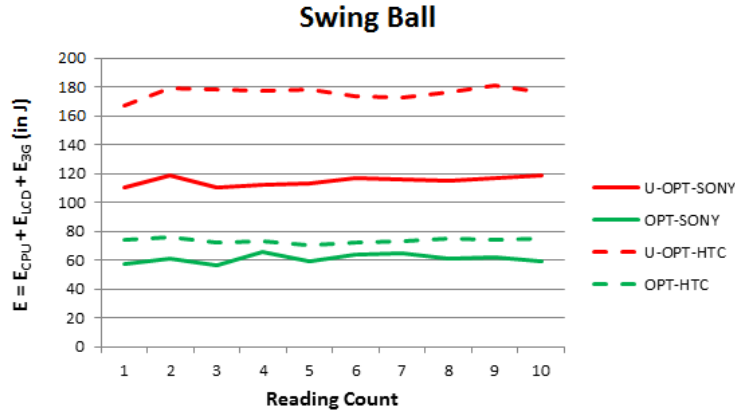


Figure 6.2: Swing Ball results OPT v/s U-OPT for both phones

The results from our experiments on the three applications are also depicted as line charts ¹ in Figures 6.1, 6.2 and 6.3. The solid lines corresponds to Sony Xperia P readings and the dashed lines correspond to HTC Explorer. Lines in red color are for Un-optimized readings

¹Note that the readings are not continuous and the lines are drawn in order to show the trends that were observed.

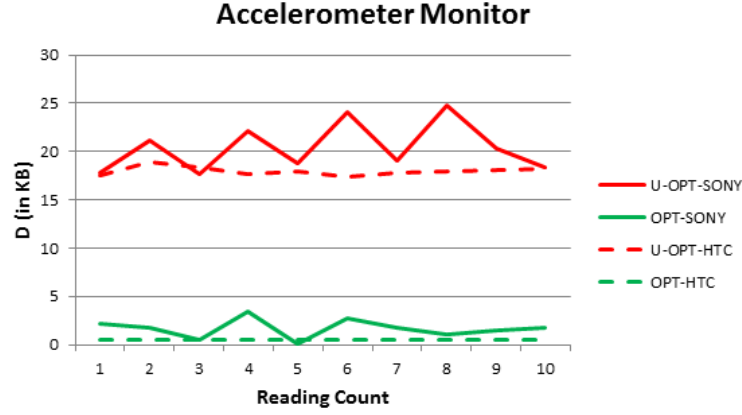


Figure 6.3: Accelerometer Monitor results OPT v/s U-OPT for both phones

and lines in green are for Optimized readings. The X axis represents the reading count and the Y axis represents the energy or data consumed by the app. These charts clearly highlight the improvements that Chiromancer has to offer.

Next we show the initial results from our memory optimization in Table 6.2. The application used for evaluation is available at [20] and [27], before and after optimization respectively, and the phone on which the experiments have been performed is Moto G. Figures 6.4(a), 6.4(b) and 6.4(c) depict the results as a graph. In Table 6.2, the record where the pool size is 0 indicates the readings for the unoptimized case. Rest of the readings are recorded for the optimized case by varying the pool size from 100 to 1000 at intervals of 100.

Table 6.2: Memory Optimization Evaluation Results

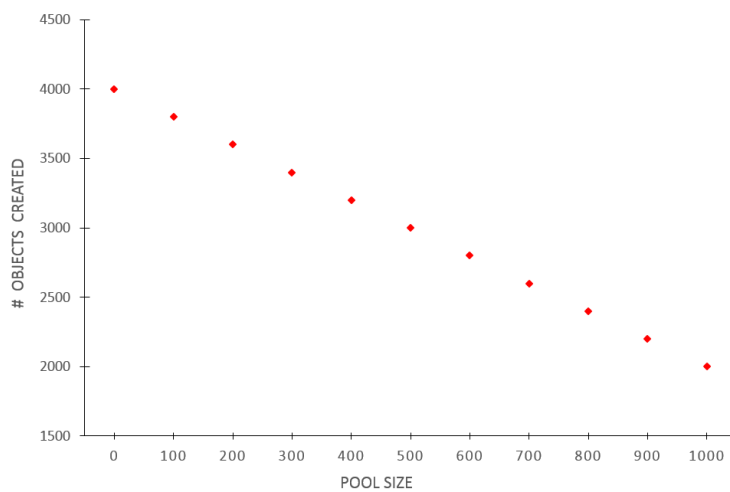
Pool Size	# Objects Created	$T_{avg}(ms)$	$S_{avg}(KB)$	# GC invocations
0	4000	184	34023	11
100	3800	153	34016	11
200	3600	145	25953	10
300	3400	144	25318	10
400	3200	145	17945	9
500	3000	137	17930	9
600	2800	178	9891	8
700	2600	114	9874	8
800	2400	115	1842	7
900	2200	114	1836	7
1000	2000	80	59	6

Here:

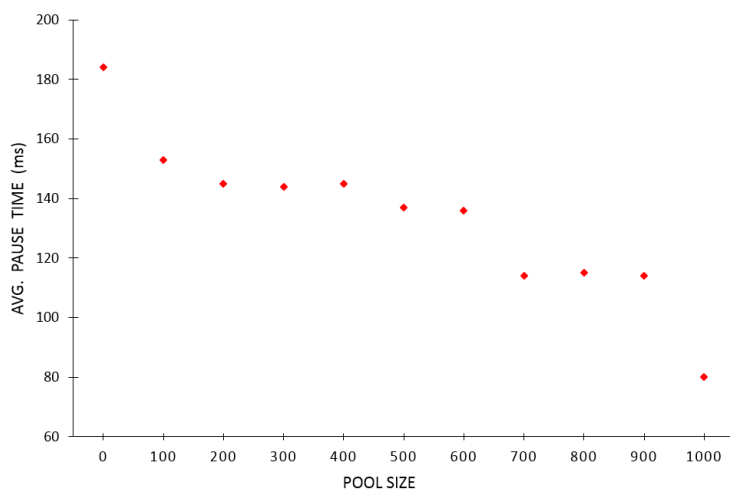
S_{avg} : Average heap space freed due to GC in KB where the average is over 10 readings

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

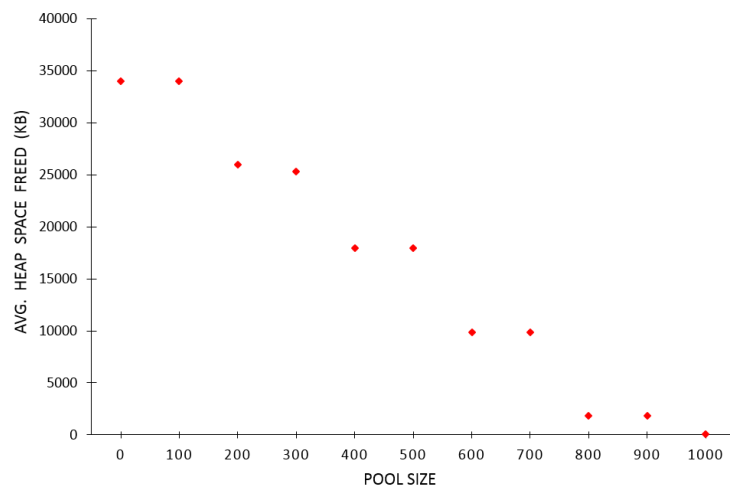
As can be observed from Table 6.2, the number of objects that get created, time spent by Dalvik garbage collector collecting garbage, space freed due to GC, and the number of GC invocations all are inversely proportional to the pool size. These results indicate that the object pool design pattern when used for Android apps can reduce the overhead on the Dalvik garbage collector significantly.



(a) # of Objects v/s Pool Size



(b) Average Pause Times v/s Pool Size



(c) Heap Space Freed v/s Pool Size

Figure 6.4: Results from Memory Optimization

Chapter 7

Discussion

7.1 Strengths

The approach we have used builds on tools like Dexpler and Soot therefore it takes advantage of their strengths. The memory based optimization presented as an extension to Chiromancer creates an Object Pool and shares it with other apps to promote object reuse. The properties of our object pool are as follows:

- (a) The Object Pool falls back on the new statement if it cannot cater to a request for an object.
- (b) We prefer fixed size arraylists. This is because allowing expansion and contraction of the data structure would introduce irregular store times.
- (c) The object pool facilitates initialization of objects which prevents stale instance values to be reused.
- (d) The object pool can handle any type of object, however, we do not allow different kinds of objects to get stored in the same pool due to reasons mentioned in Section 5.2
- (e) The object pool can be shared between multiple threads as the *newObject()* and *freeObject()* methods are synchronized.

Some advantages of Chiromancer and its extension are as follows:

- (a) Chiromancer provides fine grained control over the solution for the issues detected in the application. This means that one is not bounded by the framework to handle problems in a specific way, rather you can devise a number of techniques specific to different method signatures. For instance, *sendTextMessage()* method has a number of different parameter combinations for performing the same task of sending an SMS. But, since we detect these methods by an exact match of their parameters we can make signature specific optimizations. For the same reason Chiromancer's analysis can adapt to new Android versions with changed APIs.
- (b) The use of Dexpler avoids an intermediate conversion to Java Bytecode and hence, speeds up the overall process.
- (c) The technique used in Chiromancer is bound to be accurate; accurate in terms of number of false positives and false negatives because it relies on exact matches of method signatures.
- (d) We completely rely on static analysis and avoid the extra overhead imposed by dynamic analysis. Moreover, since the code injection happens off the shelf before the application is installed, the resources of the phone are not used thereby saving battery.
- (e) Our approach is intuitively scalable to large code bases, since it relies on Soot for most of its functionality.
- (f) The memory based optimization strikes a balance between accuracy and conservativeness. The summary based and inter-procedural nature of the analysis makes it scalable and expands the horizon of the analysis beyond a method.
- (g) This optimization also provides a framework that lays a foundation for other optimizations to be added. For instance, one can leverage our analysis (with some modifications) to reuse and recycle objects between apps.
- (h) The tool is designed so that all the relevant information is collected in a single pass over the code. Moreover, the instrumentation is done carefully so that no side effects are introduced.

7.2 Weaknesses

Despite providing the end-users with the paraphernalia to improve the performance of the application, the tool has its drawbacks. Some of the limitations it suffers from are as mentioned below:

- (a) The current version of our tool does not allow application to interact to get parameters during runtime. All user inputs are given prior to installation. So if the user wishes to change a few parameters, he would have to request the server for another copy of the APK instrumented with appropriate new parameters and would have to reinstall the application on his phone.
- (b) Android requires all applications to bear cryptographic signature. Hence, the modified APKs need to be re-signed before porting them back on the smartphone for use. So the user would have to trust the APKs we provide. Furthermore, if we re-sign the apps using our keys, the updates pushed to the apps by the developer will stop coming. We can avoid the credibility issue altogether if we provide our tool to the developers directly or to Google.
- (c) Soot and FlowDroid can sometimes be slow and could need a lot of memory [17], which is another reason we should avoid Soot based analysis to run on phone.
- (d) The recycle and reuse optimization currently does not support determining size of object pools automatically based on the app.
- (e) Due to the flow-insensitive nature of Spark we are unable to correctly deal with cases where the same reference variable (already referring to an object) is re-defined to refer to newly allocated objects. However, this issue can be resolved once Spark developers make it more flow-sensitive.
- (f) Currently, we are creating object pools for objects allocated inside loops. However, we can improve the efficiency of the app further by leveraging results from runtime traces which would indicate paths that are memory intensive and should be given a higher priority.

Chapter 8

Conclusion and Future Work

8.1 Conclusion and Future Work

We presented a tool that performs performance optimizations on Android applications using static code analysis targetting *reduction* in cost to user, energy, data and memory consumption. It offers a convenient interface that allows a user to set application parameters according to her need. The tool provides an extensible framework that allows the developers to tune parameters and build new optimizations. We are working on the API to add more functionality and make applications interactive so that they can accept parameters from the user at runtime. Few more performance issues that we wish to overcome are as follows.

- (a) Allow apps to download content only in the vicinity of a Wifi zone.
- (b) Prompt user to suspend the apps that make use of Internet, if the battery is low.
- (c) Provide support to modify application synchronization frequency with servers that hold data.

Furthermore, we *reuse* heap space allocated to objects by *recycling* objects into a pool and reusing them whenever required in the app. We leverage certain heuristics to prune the set of optimizable objects. In future we would like to add support for the following.

- (a) Automatically deciding size of object pool based on certain heuristics of the app.

- (b) Selectively creating object pools based on the number of objects of a particular class. Some objects will not be created in bulk and thus one can avoid maintaining a pool for them. We can leverage runtime traces of the app to find this out.
- (c) We would like to corroborate our claim with further evaluation on more apps.
- (d) In case the object to be optimized inherits fields from an Android library class, we leave such fields as they are. In future we would like to extend the reset functionality by incorporating initialization of such fields using the constructors of the parent Android library class. This would require access to the precise implementation of the Android API. Currently, ‘android.jar’ has only stub implementations of the corresponding constructors for Android library classes.

With the existing features and those that will be added in future, we hope that R3 helps users and developers to improve their app’s performance.

Bibliography

- [1] Admob - monetize and promote your mobile apps with ads - google ads. <http://www.google.co.in/ads/admob/>.
- [2] Android Apps, Download APK. <http://www.appsapk.com/>.
- [3] Escape Analysis. http://en.wikipedia.org/wiki/Escape_analysis.
- [4] Google Play. <https://play.google.com/store>.
- [5] Link to SlideShow App. <https://www.dropbox.com/s/8jtsjzhgfh4h1vy/ToyExample.rar?dl=0>.
- [6] Number of available Android applications - AppBrain. <http://www.appbrain.com/stats/number-of-android-apps>.
- [7] Package Index - Android Developers. <http://developer.android.com/reference/packages.html>.
- [8] Software design pattern. http://en.wikipedia.org/wiki/Software_design_pattern.
- [9] Soot: a Java Optimization Framework. <http://www.sable.mcgill.ca/soot/>.
- [10] System Permissions - Android Developers. <http://developer.android.com/guide/topics/security/permissions.html>.
- [11] AL-JAROODI, J., AND MOHAMED, N. Object-reuse for more predictable real-time java behavior. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (Washington, DC, USA, 2005), ISORC '05, IEEE Computer Society, pp. 398–401.
- [12] ALOIS REITBAUER, E. A. Reducing Garbage-Collection Pause Time. <http://javabook.compuware.com/content/memory/reduce-garbage-collection-pause-time.aspx>.
- [13] 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.

- [14] ARNI EINARSSON, J. D. N. A Survivor’s Guide to Java Program Analysis with Soot. www.brics.dk/SootGuide/sootsurvivorsguide.pdf.
- [15] ARZT, S., FALZON, K., FOLLNER, A., RASTHOFER, S., BODDEN, E., AND STOLZ, V. How Useful Are Existing Monitoring languages for securing android apps? In *Software Engineering (Workshops)* (2013), pp. 107–122.
- [16] ARZT, S., RASTHOFER, S., AND BODDEN, E. Instrumenting Android and Java Applications as Easy as abc. In *RV* (2013), pp. 364–381.
- [17] BARTEL, A., KLEIN, J., MONPERRUS, M., ALLIX, K., AND TRAON, Y. L. Improving Privacy on Android Smartphones Through In-Vivo bytecode instrumentation. *CoRR abs/1208.4536* (2012).
- [18] 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).
- [19] BODDEN, E. Instrumenting Android Apps with Soot. <http://www.bodden.de/2013/01/08/soot-android-instrumentation/>.
- [20] BRESOLIN, A. App Before Optimization. <https://dl.dropboxusercontent.com/u/105124403/Thesis%20apps%20before%20and%20after%20opt/CallWithinLoopBefore.rar>.
- [21] BRESOLIN, A. Recycling objects in Android with an Object Pool to avoid garbage collection. <http://www.devahead.com/blog/2011/12/recycling-objects-in-android-with-an-object-pool-to-avoid-garbage-collection/>.
- [22] CORPORATION, I. D. Worldwide Quarterly Mobile Phone Tracker. http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37.
- [23] ECLIPSE. The AspectJ Project. <https://www.eclipse.org/aspectj/>.
- [24] FRITZ, C. Flowdroid: A Precise and Scalable Data Flow Analysis for Android. In *EC SPRIDE* (2013).
- [25] GOOGLE. ImageView. <http://developer.android.com/reference/android/widget/ImageView.html>.

- [26] 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>.
- [27] (MODIFIED BY SAMIT ANWER), A. B. App After Optimization. <https://dl.dropboxusercontent.com/u/105124403/Thesis%20apps%20before%20and%20after%20opt/CallWithinLoopAfter.rar>.
- [28] PATHAK, A., HU, Y. C., AND ZHANG, M. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *EuroSys* (2012), pp. 29–42.
- [29] RAYSIDE, D. Points-To Analysis. www.cs.utexas.edu/~pingali/CS395T/2012sp/lectures/points-to.pdf.
- [30] SADAoui, S., AND SHARIFIMEHR, N. A novel object pool service for distributed systems. In *Proceedings of the 2006 Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part II* (Berlin, Heidelberg, 2006), ODBASE'06/OTM'06, Springer-Verlag, pp. 1757–1771.
- [31] WIKIPEDIA. Centrality. http://en.wikipedia.org/wiki/Centrality#Betweenness_centrality.
- [32] WIKIPEDIA. Live Variable Analysis. http://en.wikipedia.org/wiki/Live_variable_analysis.
- [33] WIKIPEDIA. Pointer Analysis. http://en.wikipedia.org/wiki/Pointer_analysis.