A Thesis

entitled

A Context-Aware Approach to Android Memory Management

by

Srinivas Muthu

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the Masters of Science Degree in Engineering

Dr. Jackson Carvalho, Committee Chair

Dr. Mansoor Alam, Committee Member

Dr. Henry Ledgard, Committee Member

Dr. Patricia R. Komuniecki, Dean
College of Graduate Studies

The University of Toledo
December 2015

An Abstract of

A Context-Aware Approach
to Android Memory Management

by

Srinivas Muthu

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the Masters of Science Degree in Engineering

The University of Toledo
December 2015

Smart-phones are ubiquitous and operate in diverse environments. Hence, they are well suited to utilize contextual information in enhancing the behavior of their applications. However, the potential of context analysis is not fully utilized by Android devices. The Android operating system caches the processes of recently-used applications in memory, so that they can be loaded quickly should the user request them again. We postulate that incorporating context information can improve this caching scheme. As a proof of concept, we setup an experiment comprising of twenty volunteers (Android smart-phone users) and compared the cache hit ratios of the default scheme with a default-context hybrid scheme. This hybrid scheme incorporated information from the user's calendar in determining which processes were cached. To accomplish this task, each volunteer installed an Android application that parses their Calendar for contextual clues and generates a list of applications that the user is likely to use. This list is then combined with the default list (that Android generates through recency of usage) to determine the cache hit ratios of the hybrid scheme. Finally, we use the data collected from the experiment to show why the hybrid scheme is more efficient and on a broader level, that context analysis is beneficial in enhancing the user experience.

To my parents who've always put my needs ahead of theirs.

Acknowledgments

First and foremost, I'd like to thank Dr. Jackson Carvalho for mentoring me throughout my time here at UT. Without his guidance, I would'nt be half the student I am today. I'd like to thank Dr. Mansoor Alam for believing in me and offering me tuition scholarship to pursue graduate studies, here at UT. I'd like to thank Dr. Lawrence Thomas for being an exemplary professor and his words of wisdom have always guided me in tough times.

I am grateful to Dr. Henry Ledgard for taking the time to be on my committee and mentoring me during my freshman year. I'd like to thank Dr. Donald White and his students for helping me with data collection, design and representation. It was nothing short of a privelege to work with Dr. White and his students. I'd like to thank Dean Nagi G. Naganathan for employing me and guiding me over the course of my time in graduate school. Without the help of research volunteers, this work would'nt have been possible and I'd like to convey my gratitude to every volunteer that signed up for the experiment.

I'd like to thank all my friends and family for supporting me throughout my time here at UT but I'd like to especially mention Sandy Stewart for everything she's done for me. This page is'nt enough to expound on the details but it suffices to say I would'nt be here without her and she is like a second mother to me.

Contents

List of Tables

List of Figures

List of Abbreviations

RAM	Random Access Memory
LRU	Least Recently Used
CHR	Cache Hit Ratio
OS	Operating System
AOSP	Android Open Source Project
APK	Android Package
IPC	Inter Process Communication
IC	Integrated Circuits
MB	Mega-Byte
GB	Giga-Byte
APMD	Android Powered Mobile Device
CBP	Cached Background Processes
OEM	Original Equipment Manufacturer
ADB	Android Debug Bridge
UID	unique Identifier
OOM	Out Of Memory
API	Application Programming Interface
USA	United States of America
UI	User Interface

List of Symbols

✓ ..... Represents a check mark indicating that a particular item has been checked or in a different context, whether the checked item is correct

Preface

This thesis is original, unpublished, independent work by the author, Srinivas Muthu under the tutelage of Dr. Jackson Carvalho.

Chapter 1

Introduction

1.1 Problem

With the advent of increase in the amount of RAM (Random Access memory) available in Android smart-phones, there is a case to be made that this additional memory availability can be put to better use. By default, every Android application runs in its own Linux process [50]. When a smart-phone that runs on Android is active, the RAM contains all the active processes [15] and services [8]. In addition, it also contains cached background processes. These processes are kept in memory so that in the event the user clicks any of the corresponding applications, they can be loaded onto the screen quickly, thereby enhancing the user experience. Consequently, it's in Android's best interests to maximize the chances of the user requesting one of these applications. Currently, Android uses recency of usage to determine which applications get cached. To elaborate, the most recently used applications are cached as cached background processes. Therefore, the problem at hand is determining whether there is a better caching scheme and whether contextual information can be influential in such a scheme.

1.2 Proposed Solution

We postulate that incorporating context information can lead to a better caching scheme. In our case we utilize context by looking into the user's calendar to try and predict which applications the user is likely to use in the near future. This leads to three possible caching schemes:

- Default Scheme (based on caching recently used applications)
- Context Scheme (based on caching applications predicted by reading the user's calendar)
- Default-Context Hybrid Scheme (based on caching applications from both groups)

We need to measure the efficiency of each scheme in order to determine the best one. To accomplish this task, we setup an experiment that comprised of twenty android smart-phone users, to measure the cache hit ratios of each scheme. We developed an Android application that parsed the user's calendar and generated a list of applications likely to be used by the user. Additionally, it collected the basic cache metrics (cache hit, cache miss, cache hit ratio) for each scheme. We thoroughly analyze the data collected and demonstrate the superiority of the default-context hybrid scheme.

1.2.1 Assumptions

We've made the following assumptions:

- For the purpose of this thesis, the user's context is obtained through reading the user's calendar.
- All the research volunteers own an Android smart-phone that operates at API (Application Programming Interface) Level 21 and above [57].

- Each volunteer authorized the application to access their calendar.

There are plenty of ways to obtain contextual information but reading the calendar is beneficial for two reasons:

- Android provides a public interface to read the user's calendar with their permission [46]. Therefore it's quite simple to obtain this information.
- The probability that the event listed in the calendar actually occurring is quite high. This is due to the fact that the user personally entered the event which beats models that predict user behavior.

Chapter 4 elaborates on why volunteers require API level 21 and above, and Android mandates that the user authorize all permissions for third-party applications to access their data, during installation.

1.3 Android Fundamentals

1.3.1 Overview of the Android OS

Android is a mobile OS (Operating System) based on the Linux kernel and designed primarily for touchscreen devices such as smart-phones and tablets [1]. In addition to touchscreen devices, Android TV, Android Auto and Android Wear are emerging technologies with specialized user interfaces. Globally, it is the most popular mobile OS [2]. Android has an active community of developers and enthusiasts who use the AOSP (Android Open Source Project) source code to develop and distribute their own modified versions of the operating system [4]. Android homescreens are typically made up of application icons and widgets. Application icons launch the associated application, whereas widgets display live, auto-updating content such as the

weather forecast, the user's email inbox, or a news ticker directly on the homescreen [5].

Internally, Android OS is built on top of a Linux kernel. On top of the Linux kernel, there are the middleware, libraries and APIs written in C and application software running on an application framework. Development of the Linux kernel continues independently of other Android's source code bases.

Figure 1-1: [3] Android System Architecture

1.3.2 Applications, Activities and Services

1.3.2.1 Applications

Android applications are written in the Java programming language. The Android SDK tools compiles the code, along with any data and resource files into an APK (Android Package), which is an archive file with an .apk suffix. One APK file contains all the contents of an Android application and is the file that Android-powered devices

use to install the application [6]. The Android operating system is a multi-user Linux system in which each application is a different user. Each process has its own virtual machine (VM), so an application's code runs in isolation from other applications. By default, every application runs in its own Linux process.

1.3.2.2 Activities

An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map [7]. An application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the main activity, which is presented to the user when launching the application for the first time. Each activity can then start another activity in order to perform different actions.

1.3.2.3 Services

A Service is an application component that can perform long-running operations in the background and does not provide a user interface [8]. Another application component can start a service and it will continue to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform IPC (Inter Process Communication). For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

1.3.3 Android Process Lifecycle

A Android process can be in one of five different states at any given time, from most important to least important [15]:

- Foreground process: A foreground process is one that is required for what the user is currently doing.
- Visible process: A visible process is one holding an Activity that is visible to the user on-screen but not in the foreground.
- Service Process: Service processes are not directly visible to the user, they are generally doing things that the user cares about (such as playing music in the background).
- Background process: A background process is one holding an Activity that is not currently visible to the user. They are kept in an LRU (Least Recently Used) list to ensure the process that was most recently seen by the user is the last to be killed when running low on memory.
- Empty process: An empty process is one that doesn't hold any active application components. The only reason to keep such a process around is as a cache to improve startup time the next time a component of its application needs to run.

Every Android smart-phone user is capable of checking the processes that currently reside in physical memory. The Application Manager which is part of the Settings application shows a list of running processes and cached background processes. Additionally, it breaks down the composition of RAM usage by the System applications and user installed applications.

1.4 Growth of available RAM over the years

1.4.1 Advances in Technology

RAM is a form of computer data storage. A RAM device allows data items to be accessed (read or written) in almost the same amount of time irrespective of the

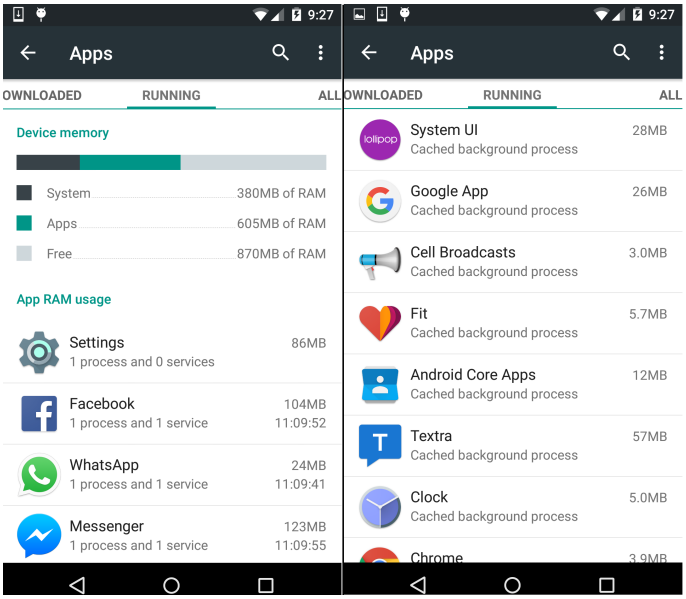

Figure 1-2: Left - List of Running Applications (Active Processes and Services)
Right - List of Cached Background Processes

physical location of data inside the memory. The overall goal of using a RAM device is to obtain the highest possible average access performance while minimizing the total cost of the entire memory system. Today, random-access memory takes the form of IC (Integrated Circuits)(s).

1.4.1.1 Moore's Law

Moore's law is the observation that the number of transistors in a dense IC doubles approximately every two years.

Advancements in digital electronics are strongly linked to Moore's law, especially in the context of memory capacity. T-Mobile G1, the very first Android smart-phone to be released back in 2008 [10] has a 256 MB (Mega-Byte) RAM [11]. In comparison, the Nexus 6P that was released in September 2015 has a 3 GB (Giga-Byte) RAM and some devices like the LG T585 have a 16 GB RAM [12][13], which amounts to 64 times the memory capacity of the T-Mobile G1. The rapid growth in the amount

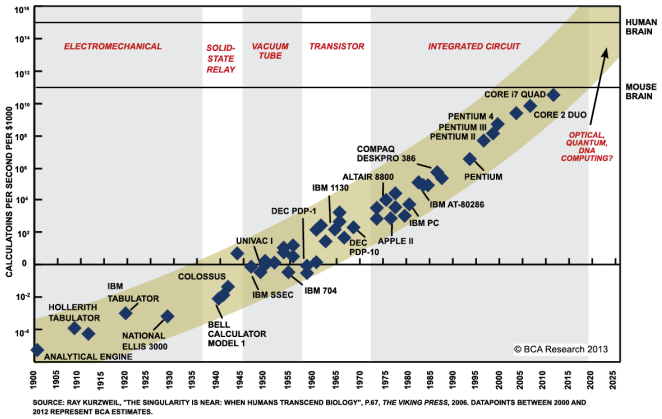

Figure 1-3: [9] Moore's Law

of RAM available to Android devices has in part led to newer possibilities and in our case, a better caching scheme for determining which application's components are cached in memory.

1.4.2 Demise of Task Killers

One of the main benefits of the Android OS is the fact that unlike certain other OS(s), it can run applications in the background. This enables us to have multiple applications open at the same time which results in true multitasking. Thus, RAM availability is highly desirable [14]. A popular misconception is that forcibly removing applications from memory in order to 'free up RAM' will result in increased performance. In fact, several task-killer applications promise to do precisely that. With the advancement in the amount of RAM available, the debate should be about how to better use all this extra space, not killing applications to 'free up' more space. In fact, the Android OS can go one step further and pro-actively cache applications that the user might use in the near future. We'll analyze this prospect in more detail in the upcoming chapters.

1.5 Organization of Thesis

Several works in the past have focused on context-aware applications and ways to observe user behavior patterns. We'll explore some these works and how this contextual data was put to use in Chapter 2. In Chapter 3, we'll take a look at some of the challenges in collecting cache metrics like detecting the applications in memory, detecting when the user clicks a new application, whether to approach the problem at the user level or the system level, to list a few. Chapter 4 elaborates on the setup of the experiment and analyzes the design of the application used to collect these metrics. We dive into the data in Chapter 5 and summarize the results. We also investigate how certain factors could explain the variation in data. We conclude this thesis with Chapter 6 and Chapter 7 talks about scope for future work and explores some of the possibilities of directly adding to this thesis.

Chapter 2

Related Work

2.1 Early Studies in Context-Aware Computing

Dey, Abowd and Salber, in their 2001 paper titled *A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications* laid out the foundation and in many ways a standardized framework for building context-aware applications. When this paper was published, mobile phones were becoming mainstream devices and were no longer confined to the hands of the wealthy. By definition, more people were mobile and the devices they carried with them had the potential to utilize contextual information gathered from the surroundings, in ways that desktops could'nt do due to their stationary nature. They defined context as any information that characterizes a situation related to the interaction between humans, applications and the surrounding environment. They opined that the state of research in context-awareness was inadequate for three main reasons:

- The notion of context was ill-defined.
- There was a lack of conceptual models and methods to help drive the design of context-aware applications.
- No tools were available to jump-start the development of context-aware appli-

cations.

They focused their efforts on the pieces of context that could be inferred automatically from sensors in a physical environment and produced Context Toolkit [20], a conceptual framework that supported the rapid development of context-aware applications. It is important to keep in mind that at the time this paper was published, the notion of what we refer to as a smart-phone was non-existent. This is evident from the fact that most APMD(s) have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful in monitoring three-dimensional device movement or positioning, or changes in the ambient environment near a device [21]. These conditions make APMD(s) very suitable for utilizing contextual information and translating that potential into a better user experience. We utilize this feature of APMD(s) to propose ways to improve user experience by incorporating contextual information in managing the physical memory of the APMD.

2.2 Context Patterns in Application Usage

Several works have attempted to collect user behavior data and they've accomplished this in unique ways. Hannu Verkasalo, in his 2007 paper titled *Contextual Patterns in Mobile Service Usage* analyzes how mobile services are used in different contexts [18]. Usage contexts were divided into *home, office* and *on the move*. A specialized algorithm tracked user's location patterns over a period of time to determine whether the user was at home or at work. Furthermore, changes in location data revealed whether the user was in transit between home and work. There was a correlation between the mobile services requested by the user and the user's physical location, which enabled the algorithm to understand the user's context quite well (refer Figure 2-1). It was also observed that certain services were requested more

during the weekends as opposed to certain others being requested more during the weekdays i.e. days with office hours.

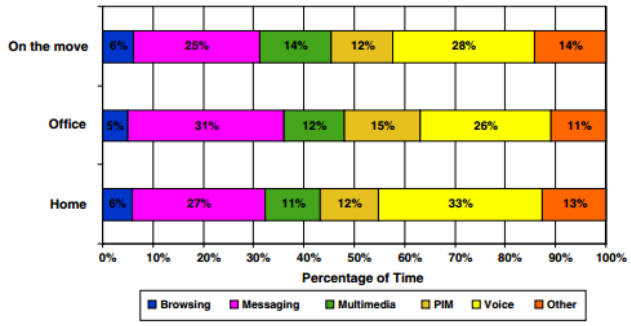

Figure 2-1: By classifying user context into three distinct entities namely *home, work* and *on the move*, the author managed to infer which services were requested more in a given context. For e.g. Voice services i.e. calling was predominantly requested while the user was at home. This type of information can be very useful.

2.3 Context Phone

Mika Raento, Antti Oulasvirta, Renaud Petit, and Hannu Toivonen, in their paper titled *ContextPhone: A Prototyping Platform for Context-Aware Mobile Applications* proposed that mobile phones are well suited for context aware computing due to an intimate relationship between the user and the phone [19]. Their primary goal was to provide context as a resource and in order to accomplish that, developed a *ContextPhone* which comprised of four components (Refer Figure 2-2):

- Sensors that acquire contextual data (such as location data from GPS)
- Communication services (such as SMS, MMS to list a few)
- Customizable Applications (that can replace existing applications)
- System Services (such as background services for error logging)

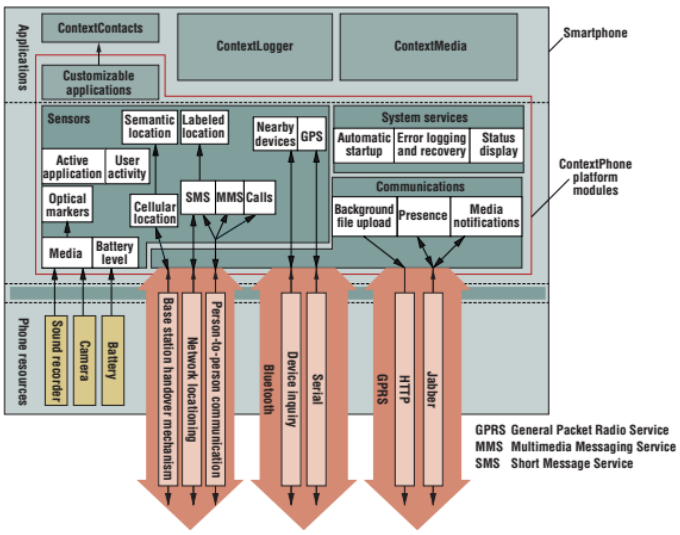

Figure 2-2: The ContextPhone platform; Four interconnected components sensors, system services, communication services, and customizable applications facilitate communication with the outside world.

It is important to observe that this paper came out in 2005, two years before the first version of Android was even launched. Yet, their fundamental point of focus is still relevant in how we can use context as a powerful resource. Their point about the necessity of customizable applications that are needed for context information to be relevant perfectly applies to this thesis. The Android OS which is open sourced as AOSP is as customizable as it can get when it comes to modifying mobile components and by extension the user experience with APMD(Android Powered Mobile Device)(s). To be specific, it facilitates ways to gather cache related information such as the list of processes currently residing in memory, the current foreground application and whether the caching mechanism per se is up for modification to list a few. It would be impossible to collect and/or modify such information on any other mobile OS (such as Apple's iOS).

2.4 Using Context Information for Authentication

Dey et al. established a framework for collecting contextual data and Verkasalo and Raento et al. have shown how user behavior can be analyzed through sensors and other modules that gather contextual information. Shi et al.'s paper titled *Implicit Authentication through Learning User Behavior* demonstrates a similar approach in collecting behavioral data of the user but differ in how they put that information to use. They have devised a model to implicitly authenticate smart-phone users based on their behavioral patterns. The user's usage patterns are collected (for e.g. how many calls a user makes a day, how many times user's location changes (Refer Figure 2-3) etc.) and this information is used in building a user model [22].

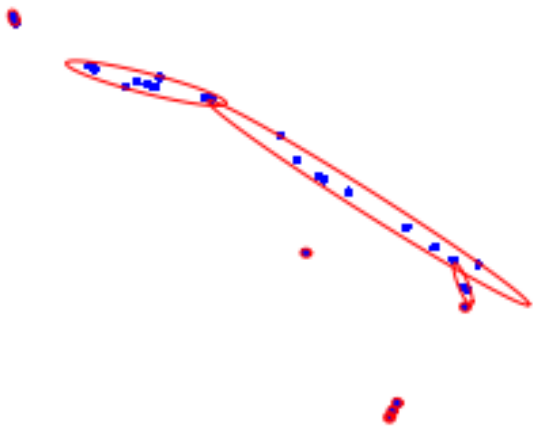

Figure 2-3: The blue dots represent the users traces in a two-hour epoch over multiple days, and the red ellipses represent the clusters fitted. The major two directional clusters correspond to the users trajectory on a highway

Once the user model is built and the profile completed, the algorithm can determine the likelihood of a random user of the phone not being the owner of the phone (the one whose user model was profiled).

As demonstrated by Shi et al., there are numerous ways contextual information can come in handy. In our case, we propose a user-centric caching scheme for caching

application components in memory, demonstrate its superior efficiency and user experience, by looking into the user's calendar for contextual information.

Chapter 3

Challenges in Collecting the Metrics

3.1 Introduction

Now that we've laid the foundation for using contextual information to influence which processes Android caches in memory, we need to establish the metrics that need to be collected in order to measure the efficiency of the existing caching scheme and the proposed caching scheme. The following section addresses the metrics we need.

3.2 What are the desired metrics?

3.2.1 Hits, Misses, Hit Ratio & Latency

Alan Jay Smith defines Cache as a high speed buffer that holds items in current use [23]. In our context, Android holds application components (such as processes) as CBP (Cached Background Process)(s) in memory. As previously discussed, CBP(s) do not take up processor time, they are merely cached for quick application startup should the user request that particular application. A cache hit is a state in which

data requested for processing by a component or application is found in the cache memory. It is a faster means of delivering data to the processor, as the cache already contains the requested data [24]. When applied to our scenario, a cache hit would represent a situation wherein the user requests (clicks) an application that currently resides in memory, either as an active process (including services) or as a CBP (Refer Figure 3-1).

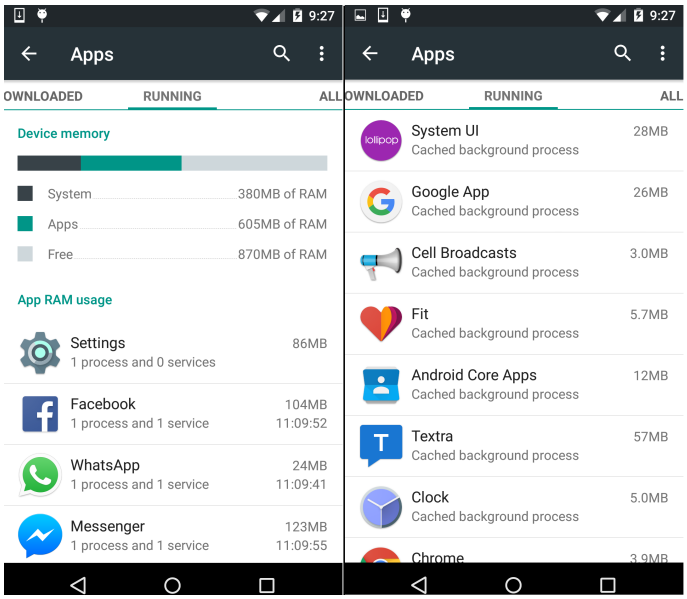

Figure 3-1: Given this snapshot of the RAM, if the user clicks on the Messenger application (active process) or Clock application (CBP), it would result in a Cache Hit

A cache miss is a state where the data requested for processing, by a component or application is not found in the cache memory [25]. When applied to our scenario, a cache miss would represent a situation wherein the user requests (clicks) an application that currently does not reside in memory, either as an active process (including services) or as a CBP (Refer Figure 3-2).

CHR is defined as the percentage of cache hits compared to the overall number of requests (total of cache hits and cache misses).

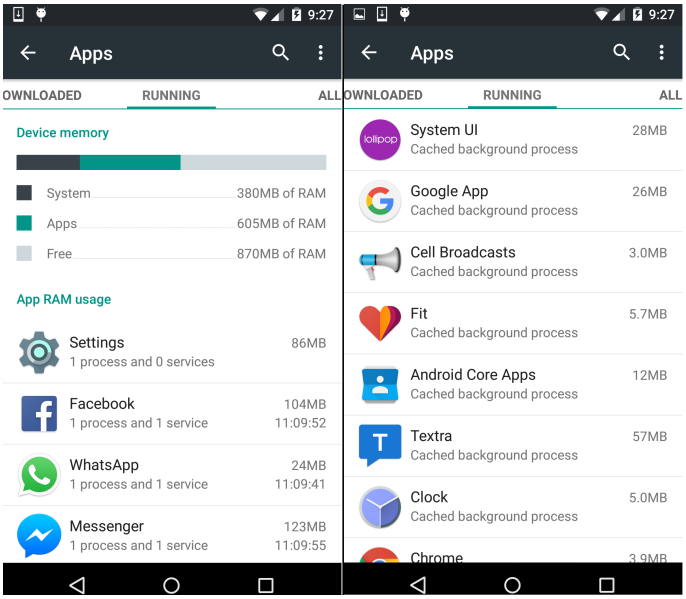

Figure 3-2: Given this snapshot of the RAM, if the user clicks on say, the Google Drive application (process not in memory), it would result in a Cache Miss

$$OverallNumberOfRequests = CacheHits + CacheMisses \tag{3.1}$$

$$CHR = \frac{CacheHits}{OverallNumberOfRequests} * 100 \tag{3.2}$$

There are two primary figures of merit of a cache [23]:

- Latency
- CHR

The latency of a cache describes how long after requesting a desired item, the cache can return that item (in the case of a cache hit). In our scenario, processes are cached in RAM. Therefore, even if the number of processes cached in memory increases, there will hardly be any difference in the time it takes to fetch a reque-sted item (application component) as they all reside in RAM which by definition supports

random access. Latency is more of a factor in caches that are located further away from physical memory (such as L1 and L2 caches).

The CHR of a cache describes how often a searched-for item is actually found in the cache. The higher the CHR, the better the cache is. When applied to our context, a high CHR results is more application components being fetched from memory, rather than disk which in turn results in quicker application startup times. This improvent in startup speed results in an enhanced user experience.

3.2.2 Caching Schemes Under Consideration

Now that we've determined the fundamental metrics we need, let's analyze the various caching schemes under consideration.

3.2.2.1 Optimal Caching Scheme

The most efficient caching scheme would always have the requested item in cache. In our case, the ideal scheme would be one that ensures that every application the user requests, resides in memory. This optimal result is referred to as Belady's optimal algorithm [26] or the clairvoyant algorithm. Since it is generally impossible to predict the exact behavior of the user, this is not implementable in practice. The practical minimum can be calculated only after experimentation, and one can compare the effectiveness of the actually chosen caching scheme against Belady's as a benchmark.

3.2.2.2 Recency of Usage

This caching scheme primarily relies on recency of application usage and this is the default scheme Android currently uses. Android caches application components in memory as CBP(s) and associates each with a time-stamp. If the memory gets too full, it starts removing the least recently used CBP(s) first and depending on how direly low the memory is, it may not stop killing processes until even the active

foreground application (the one user is interacting with) is removed [15], although this situation is extremely rare in practice and only results in the case of malfunctioning applications that leak memory. The Settings application can give the user clues about which CBP(s) were recently used. Clicking on an application that is currently not in memory results in that application being cached as a CBP near the top of the list. This suggests that the CBP(s) displayed by the Settings application are roughly sorted by their time-stamp from most recently used to least recently used (Refer Figure 3-3).

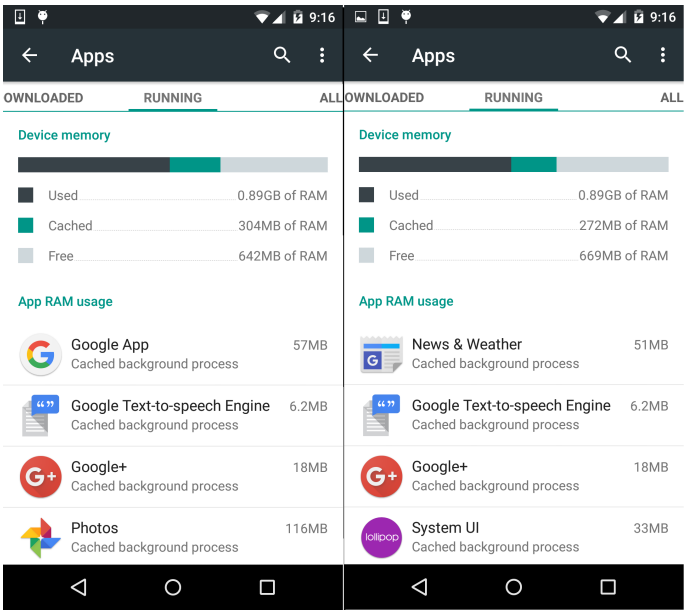

Figure 3-3: The image to the left describes the snapshot of the CBP(s) as displayed to the end-user by the Settings application. The image to the right displays the snapshot of the CBP(s) after the end-user requests for the News and Weather application. Note that the News and Weather application will not be cached as a CBP until the user is done interacting with it as up until that point, it will reside in memory as an active process.

It is worth noting that a perfect implementation of this caching scheme requires a time-stamp on each reference, and the OS needs to keep a list of items ordered by the time-stamp. This implementation may be deemed too expensive in certain cases

especially if the frequency of memory references is high. The common practice is to approximate the behavior of this scheme [27].

3.2.2.3 Frequency of Usage

This caching scheme relies on the frequency of application usage as opposed to the recency of application usage. To elaborate, the most frequently used applications would be cached as CBP(s) in memory. This could be a really good alternative to the default recency-based scheme, given the tendency of most smart-phone users to stick with the same set of popular applications and use them in rotation [31]. Under these circumstances, the frequency of a select few applications would be really high and would always be cached in memory as they're requested often. Frequency based caching scheme is sometimes combined with a recency-based caching scheme to form a frequency-recency hybrid [30].

3.2.2.4 Random

Much like the name suggests this caching scheme would randomly select an application and cache it in memory as a CBP. This scheme does not require keeping any information about the access history. For its simplicity, it has been used in ARM processors [32]. When applied to our scenario, it does'nt seem like a viable option, mainly because there is'nt a benefit in randomly caching applications in memory.

3.2.2.5 Default-Context Hybrid

So far we've analyzed the standard caching schemes that focus on caching items that were either accessed recently, frequently or in other ways that analyze past behavior. Our approach combines the applications that Android caches through a recency of usage scheme, with a separate context-analyzer module that reads the user's calendar and predicts which applications the user is likely to use. When the

user requests an application, the combined set of applications (default applications based on recency and the list of predicted applications) is used as the base to compute whether a cache hit or a cache miss occurred. We'll analyze in detail how each module functions but before that we need to examine two things:

- Whether Android can pro-actively cache application components in memory.
- Which applications will be removed from memory when it runs too low.

Pro-actively Adding CBP(s) In order for the proposed Default-Context Hybrid (henceforth referred to as Hybrid) caching scheme to work, there must be provision for the Android OS to pro-actively cache application components in memory as CBP(s). Presuming that the context-analyzer module does its job and suggests a list of applications the user might use in the near future, there must a mechanism for Android to utilize that information and pro-actively cache these applications in RAM as CBP(s). This mechanism can be verified at the user level through a small experiment. As previously mentioned, the Settings application provides a visual interface for the list of active processes and services running in memory and the CBP(s) that reside in memory at any moment in time. As part of the experiment, each and every CBP was forcefully removed from the cache (there is provision to do this at the user level (Refer Figure 3-4)).

Once each CBP was removed, the phone was untouched for three minutes. After three minutes, when the Settings application was launched to look at the list of CBP(s), it wasn't empty. Instead, there were several CBP(s), some of which were previously removed and some of which were'nt in the list to begin with (Refer Figure 3-5).

This indicates that Android pro-actively caches applications that were'nt in memory to begin with, as CBP(s). This is also the reason why task killer applications fell out of favor [33]. This demonstrates that not only does Android provide a mechanism

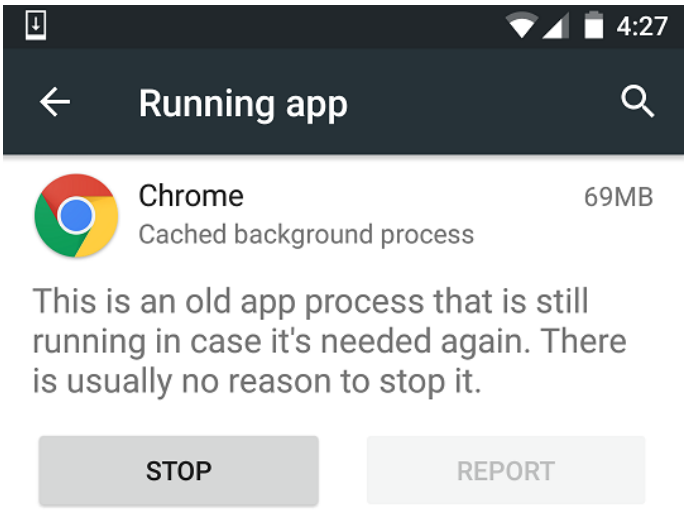

Figure 3-4: Pressing the *STOP* button removes the CBP from memory

to pro-actively cache applications in memory, it already does so. The applications suggested by the Hybrid scheme can be introduced into the memory as CBP(s) in the same way. This is discussed in Chapter 7.

Prioritizing Applications in Hybrid Scheme The Hybrid scheme would involve storing as CBP(s), both default application components cached by Android's recency-based scheme and the application components of the list of applications obtained from analyzing the user's calendar. So the question now becomes, who gets kicked out when memory is running low? Since the applications inferred from user's contextual information do not have a recency associated with them, the overall list of CBP(s) cannot be prioritized based on recency of usage. A simple solution is to prioritize the CBP(s) obtained from the list of suggested applications over the default ones i.e. all the default CBP(s) are removed from memory before the CBP(s) cached as a result of context inference. There are two advantages to this approach:

- Only few applications are suggested by the context module.
  - For instance, say the user has a Calendar entry that reads *Call Mom*.

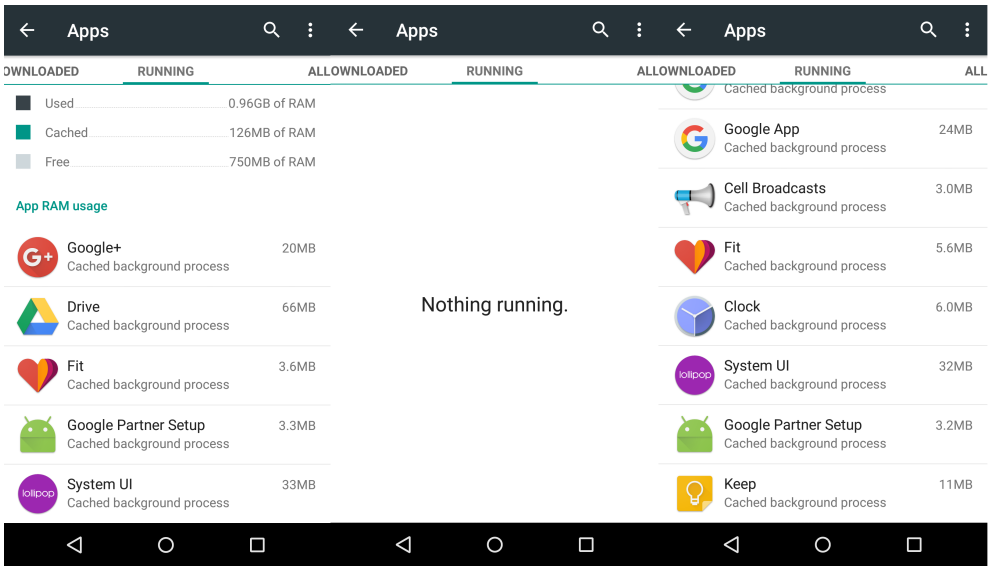

Figure 3-5: The leftmost image shows the snapshot of the CBP(s) before forceful deletion. After each and every CBP was removed manually, the screen displayed is shown in the middle image. The image farthest to the right shows the repopulated list of CBP(s), 3 minutes after the middle image was taken. This demonstrates Android's ability to utilize free RAM space and pro-actively cache CBP(s)

There will be two or at most three applications suggested by the context-module that would be potentially useful to the user in the context of calling his/her mother. This makes it more convenient to target CBP(s) cached by the default recency-based scheme in case of low memory.

- The probability of user requesting an application related to the Calendar event is quite high.
  - This is due to the fact that a recorded event in a user's Calendar is highly likely to occur and the applications needed in this context have a higher chance of being utilized as opposed to CBP(s) cached by the recency of usage.
  - This is explored further in Chapter 7 where we discuss the likelihood of an

application being used in conjunction with a Calendar event.

Alternatively, the CBP(s) with the least memory footprint can be prioritized i.e. the most memory-heavy CBP(s) are removed first. While this approach releases memory quickly, it does'nt take into account the likelihood of application usage in the immediate future.

3.2.3 Supplementary Data

We've talked about the fundamental metrics involved in measuring the efficiency of any caching scheme, namely cache hits, cache misses and CHR. We explored potential candidates for serving as the caching scheme in managing the list of CBP(s) cached in memory. There are other subtle measurements that are'nt as significant as the fundamental metrics but are important nonetheless as they shed light on certain attributes of the experiment. They are listed below:

- Number of Installed Applications
- Number of Unique Applications Requested (Clicked)
- Number of Calendar Entries During the Course of the Experiment
- Phone Model
- Android OS Version

These are elaborated upon in Chapter 5 under the demographic data section.

3.3 Challenges

Now that we've established what the desired metrics are, let's examine some of the challenges in collecting them.

3.3.1 Deciding Between System vs User Level Approach

Firstly, we need to determine whether to approach the problem from a user level perspective or not. Attempting to solve the problem i.e. collecting the aforementioned metrics at the user level would involve building Android applications to gather relevant data. On the other hand, approaching the problem from a system level would involve altering the source code of the Android OS (facilitated by AOSP) and recompiling it to build our own custom distribution, much like the CyanogenMod community [34]. It is easier to solve the problem (if possible) at the user level for two reasons:

- The complexity involved in altering OS source code is significantly higher compared to writing Android applications.
- It would be significantly harder to convince potential research volunteers to reboot their smart-phones with a custom distribution of the Android OS as opposed to asking them to install an Android application or two.

Given that it's beneficial to gather relevant data at the user level, we need to identify the complexities involved and determine whether they are solvable at the user level. In order to effectively determine which caching scheme is ideal for caching CBP(s) in memory, we need to identify whether the following problems can be approached from a user level perspective:

- Getting the list of processes currently in memory ✓
- Knowing when a new process is launched by the user ✓
- Reading the User's Calendar ✓

We'll inspect in detail, why and how we need to solve these problems (and others) in the next section but the '✓' indicates that they are indeed solvable at the user level and do not require system level changes.

3.3.2 Getting Processes in Memory

Let's first address why it's necessary to get the list of processes that reside in memory. In our scenario, we are trying to determine the most efficient caching scheme among the following:

- Default Recency based Scheme
- Pure Context Scheme (Applications will be cached solely based on context inference)
- Default-Context Hybrid (or simply Hybrid)

In order to determine which scheme is most efficient, we need the ability to measure CHR data and by extension if and when a cache hit or miss occurs. We can only determine if a cache hit occurred if we had access to the list of processes in the cache i.e. the active processes and the CBP(s) in memory. The *ActivityManager* class in Android provides two relevant helper methods [35]:

- *getRunningTasks()*
- *getRunningAppProcesses()*

Unfortunately, as of Android 5.0, it has become increasingly difficult to get a list of applications running in memory. *getRunningTasks()* has been deprecated and *getRunningAppProcesses()* has been killed (It only returns the requesting application rather than all applications running in memory). An alternate solution is to use the *UsageStatsManager* class [36]. However, it requires special permission from the user in the Settings application and some OEM (Original Equipment Manufacturers)(s) have removed this setting explicitly rendering this solution unreliable. The best solution involves using the ADB (Android Debug Bridge) shell to retrieve the desired information which can then be parsed into a suitable list of processes residing in

memory. ADB is a versatile command line tool that enables communication with an APMD. It is a client-server program that includes three components [37]:

- A client, which runs on the APMD. The client can be invoked from a shell by issuing an ADB command.
- A server, which runs as a background process on the APMD. The server manages communication between the client and the ADB daemon running on the APMD.
- A daemon, which runs as a background process on the APMD.

The ADB provides a Unix shell that can be used to run a variety of commands on the APMD. The command binaries are stored in the APMD's file system at */system/bin/* [38]. We eventually obtain the list of processes residing in memory by running the *toolbox* command [42] in the shell. The Android *toolbox* command encapsulates the functionality of many common Linux commands (and some special Android commands) into a single binary [39]. Of these, we are most interested in the *toolbox* and *ps* commands as they provide a direct window into the processes residing in memory.

With the help of *libsuperuser* library which provides a convenient framework to run shell commands [40] and the *AndroidProcess* library which provides a framework to parse the process-related shell outputs [41], the following algorithm (Refer Algorithm 1) was written to obtain a list of processes residing in memory (active processes and CBP(s)). It is worth noting that this list corresponds to the list of processes displayed by the Settings application (Refer Figure 3-1) as part of the running applications and CBP screens, and that the Settings application has system privileges that third party Android applications don't.

```
Result: LIST Process
INITIALIZE LIST Process - processes
LIST String - stdout — Shell.SH.run("toolbox ps -p -P -x -c")
Integer myPid — android.os.Process.myPid()
for String line : stdout do
    INITIALIZE Process - process
    if process matches APP-ID-PATTERN then
        if (process.ppid EQUAL TO myPid)
        OR (process.name EQUAL TO "toolbox") then
            CONTINUE
        else
            ADD TO processes - process
        end
    else
        CONTINUE
    end
end
RETURN processes;
```

Algorithm 1: In the above algorithm, *Process* is a custom data structure that parses the output format of the *toolbox ps* command. Its class structure is discussed in detail in Chapter 4. The *APP-ID-PATTERN* is a custom regular expression that maps to application ID patterns for the Android OS in versions 4.0 and above. Each process is added to a result list that is returned by the algorithm. Note that the requesting application represented by *myPid* (discussed in 4.4) and *toolbox* related processes represented by process names that match *toolbox*, are'nt added to the result.

3.3.3 Obtaining the Foreground Application

In order to update the CHR metric, we need the ability to detect new cache hits and cache misses. A new cache hit/miss occurs when there are new application requests i.e. changes in the foreground application. Before we analyze how to detect change in the foreground application, we first need a way to obtain the current foreground application. Had we approached this problem at the system level, we could've made use of system level privileges such as listening to a new screen-touch by the user and detecting which application was launched but this is not possible from a third party user-level application. Fortunately, there is a workaround to detecting the foreground application at the user level. We obtain every process currently running in memory and through a process of elimination, obtain the foreground application. To elaborate, the foreground application has certain unique properties in its process name, UID (Unique Identifier) and other fields, that can be exploited to eliminate processes that are not associated with the current foreground application. The following algorithm (Refer Algorithm 2) illustrates this process.

Result: Foreground Application

INITIALIZE LIST *files* — LIST */proc* File

for *File file : files* **do**

  if *file IS A DIRECTORY* **then**
    CONTINUE

  end

  pid — *file.getName()*

  cgroup — FILE READ (*/proc/%d/cgroup*)

  cpuSubSystem, cpuAccountSubSystem — *cgroup*

  if *cpuAccountSubSystem ENDS WITH pid OR cpuSubSystem ENDS WITH bg_non_interactive* **then**
    CONTINUE

  end

  cmdline — FILE READ (*/proc/%d/cmdline*)

  uid — PARSE FROM *cpuAccountSubSystem*

  if *cmdline CONTAINS com.android.systemui OR uid BETWEEN 1000 AND 1038* **then**
    CONTINUE

  end

  oomAdj — FILE READ (*/proc/%d/oom_score_adj*)

  oomScore — FILE READ (*/proc/%d/oom_score*)

  if *LOWEST oomScore AND oomAdj EQUAL TO 0* **then**
    RETURN cmdline

  end

end

Algorithm 2: This algorithm obtains the current foreground process i.e. the process associated with the application that the user is interacting with.

Firstly, all files with path */proc* are obtained and each iterated over, to find the foreground application. We eliminate file directories as these are of no concern to us. If the CPU account sub-system ends with *pid*, it's not an application process and is thus eliminated. Similarly, if the CPU sub-system ends with *bg_non_interactive*, it's a background process and is eliminated as well. The application stored in *cmdline* is a potential candidate for the foreground application and is the right choice as long as it clears four more hurdles:

- *cmdline* does not contain *com.android.systemui*
- *uid* does not lie between 1000 and 1038
- *oomAdj* i.e. the OOM (Out Of Memory) adjustment level [43] equals 0
- *oomScore* is the lowest among all iterated values

If *cmdline* contains *com.android.systemui* or if *uid* lies between 1000 and 1038, it's a system application and is thus eliminated from consideration. The crucial attribute of the foreground application is that it has the lowest OOM score of all applications. When the Android OS is running too low on memory, it is the job of the Linux OOM killer [44] to sacrifice one or more processes in order to free up memory for the system when all else fails. A low OOM score indicates that the process is less likely to be killed and that it is lower on the OOM killer's radar. It quite intuitive that the process with the lowest OOM score would be the foreground process as it is of paramount importance not to remove the process, the end-user is interacting with, unless it's the very last resort. This unique property of the foreground process enables us to distinguish it from the rest of the pool.

3.3.4 Updating the Statistics

So far, we've established ways to collect the list of processes in memory and the foreground application. The next step is to figure out a way to update the metrics as and when the user requests new applications. The rate at which the end-user toggles between applications varies from person to person. It also depends on the level of usage. For instance, there are times when the end-user is actively using their APMD and others when the APMD is idle or is switched off. Therefore it's hard to predict when there could be a potential change in the foreground application. There is no inherent mechanism for user-level applications to detect third party application launches (That privilege is reserved for system level processes). Therefore, our solution comprises of starting a service [8] that retrieves the current foreground process (Refer Algorithm 2) and the current list of processes residing in memory (Refer Algorithm 1) every second. If there is a change detected in the foreground process (compared to the last second), then the cache metrics are updated. The time interval of one second is chosen as it deemed to be reasonably accommodative of the average time it takes an end-user to switch from one application to another. This is further discussed in the Points of Weakness section. The design of the service and the corresponding application is discussed in sections 4.4 and 4.5. The following algorithm (Refer Algorithm 3) illustrates the process of updating the cache metrics.

Result: Update Cache Metrics

for *EACH SECOND* **do**

newForegroundApp — *getNewForegroundApp()*

currentListOfProc — *getListOfProcesses()*

if *newForegroundApp EQUALS oldForegroundApp* **then**

CONTINUE

end

if *oldListOfProc CONTAINS newForegroundApp* **then**

cacheHit++

else

cacheMiss++

end

updateMetrics()

oldListOfProc — *newListOfProc*

oldForegroundApp — *newForegroundApp*

end

Algorithm 3: This algorithm retrieves the foreground application and checks if it has changed since the last second. If it has, it verifies whether it was a cache hit or a cache miss and updates the cache metrics. Finally, it updates the old foreground application and the previous list of processes to the current one so that the next iteration uses these values as the base.

3.3.5 Backing Up the Data

Previously, we discussed the mechanism by which the background service updates the cache metrics. The other aspect of this issue is storing this data. Before we inspect how to store the data, let's discuss the necessity for such a provision. Our background service is designed to run throughout the duration of the experiment and retrieve the foreground application and the list of processes in memory every second.

This results in it's OOM score increasing i.e. it's likelihood of getting killed by the Linux OOM killer [44] which would result in a loss of data. It is worth noting that the OOM killer selects the *best* process to kill which is decided by the combination of how long a process has been running and how much memory would be released if the process is killed. Even though our process scores poorly on the time duration aspect, it takes up relatively low memory compared to some of the other common background processes (Refer Figure 3-6).

Android provides several options for saving persistent application data. There are five ways to accomplish this task [45]:

- Shared Preferences - It is used in cases where primitive data needs to be stored in Key Value pairs.
- Internal Storage - It is used to store private data on device memory.
- External Storage - It is used to store public data on shared external storage
- SQLite Databases - They are used to store structured data in a relational database.
- Network Connection - It is used to store data over the Internet.

For our scenario, Internal Storage is the best fit as it keeps the data private to the application and since the data is quite primitive (cache hits and misses), we do not need SQLite databases or network connections. The data is written to a file every minute. That way, we don't have to constantly perform I/O operations from the service and in the worst case scenario, fifty nine seconds of Cache data would be lost which is relatively a small amount of time. Algorithm 3 can be tweaked to incorporate the file write operation (Refer Algorithm 4).

Result: Update Cache Metrics AND Write to File

for *EACH SECOND* **do**

```
...
...
updateMetrics()
if END OF MINUTE then
    writeMetricsToFile()
end
...
...
```

end

Algorithm 4: In addition to updating the metrics every second, it also writes the data to internal storage every minute. Any time the application is closed and launched again the metrics are initialized with the values in file.

3.3.6 Reading the User's Calendar

To compute the CHR of the Hybrid scheme, we need the ability to obtain the list of events from the user's calendar and the ability to parse that information to come up with a list of suggested applications. We'll discuss the parsing aspect in the next segment and focus on finding a way to read the user's calendar. The solution to this problem is quite simple, Google provides a public API (Application Programming Interface) [46] to obtain access to the user's calendar information, including the list of events. This operation does'nt have to be carried out with the same frequency as reading the list of processes in memory or obtaining the foreground application. In our solution, we query the calendar every four hours (for events occurring in that four hour period) and feed the data (events list) into a module that parses the information and produces a suggested list of applications (Refer Algorithm 5).

Result: Update Cache Metrics, Write to File AND Update List of Suggested Applications

```
for EACH SECOND do
    ...
    ...
    updateMetrics()
    if END OF MINUTE then
        writeMetricsToFile()
    end
    if END OF 4 HOURS then
        updateListOfSuggestedApplications()
    end
    ...
    ...
end
```

Algorithm 5: In addition to updating the metrics every second and writing them to file every minute, it also updates the list of applications suggested by the Calendar Parser (discussed in next segment)

The following algorithm (Algorithm 6) illustrates our implementation of the module that queries the user's calendar.

Result: Get List of Events from Calendar

INITIALIZE List *result*

List *allEvents* — QUERY *content://com.android.calendar/events*

for *EACH event IN allEvents* **do**

  if *event.startTime() IS WITHIN 4 hours FROM NOW* **then**
    result.add(event)

  end

end

RETURN result

Algorithm 6: This algorithm obtains all events in the user's calendar that start within 4 hours from the time of the algorithm invocation. Since this algorithm is invoked every four hours, every event in the user's calendar is parsed although it's worth noting that once read, a change in the event details is not translated to the application. This is discussed further in the Points of Weaknesses section.

3.3.7 Parsing the Calendar Information

We've examined the mechanism behind reading the user's calendar, thereby solving one half of the puzzle. The other half is utilizing this information to compute the Hybrid scheme's metrics. In order to achieve that, we need to build a module that parses the list of events obtained from reading the calendar and produces a suggested list of applications. Every event in the user's calendar is split into individual keywords and a predetermined mapping between keywords and applications builds the list of suggested applications. This mapping is determined by a two factors:

- There is a default application [58] that directly maps to the keyword.
  - For instance the keyword *mail* is mapped to the *GMail* application, which

is the default e-mail application in APMD(s).

  - Similarly the keyword *call* is mapped to the *Google Dialer* application, which is the default application to make and receive calls.

- Additionally, when the Google Play Store [48] is searched with the keyword, the top result is mapped to it.

  - For instance, when the Play Store was searched with the keyword *skype*, Skype application was the top result. Once it is verified that Skype is part of the installed applications [49], it is added to the list of suggested applications.

Finally, if there are no events in the user's calendar for the four hour period, then the list of suggested applications is populated with five default applications. These five applications are derived from the list of most popular Android applications in USA (United States of America) [47].

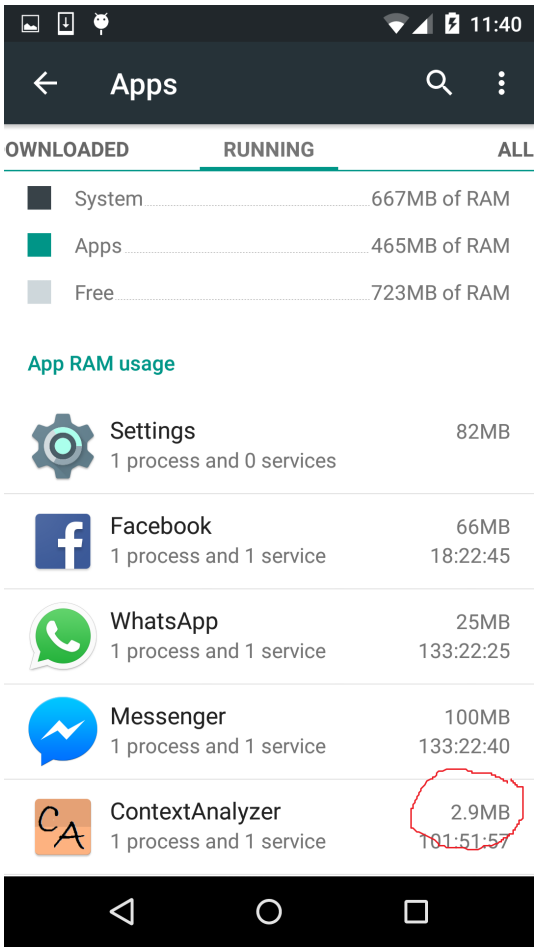

Figure 3-6: *ContextAnalyzer* is our custom Android application that collects cache metrics. The service mentioned in the diagram is the background service that updates the metrics every second. We'll discuss the design of this application in Section 4.4 but it's worth noting here that the memory footprint of the application is quite low. This is not to suggest that it won't get killed by the OOM killer (as its longevity is still a factor), which is why we're backing the data up.

Chapter 4

Experiment Setup and Application Design

4.1 Introduction

In the previous chapter, we examined the metrics we needed to analyze the best caching scheme for caching application components in memory as CBP(s). We looked into some of the common caching schemes and analyzed some of the challenges in collecting the metrics needed to determine if contextual information can help better the CHR of the Hybrid approach. To analyze the efficiencies of the default recency-based approach and the Hybrid approaches, an Android application namely *ContextAnalyzer* was developed to collect the desired cache metrics. It was installed in the APMD(s) of research volunteers. We'll dissect *ContextAnalyzer*'s design in Section 4.4.

4.2 Eligibility for Volunteers

The following criteria had to be met to qualify as a research volunteer:

- The volunteer must own an APMD with Android OS version greater than 5.0

(Lollipop and above).

- The volunteer must be willing to install a third party application on their APMD.

The algorithm that obtains the list of processes currently residing in memory (Refer Section 3.3.2) and the algorithm that retrieves the current foreground application (Refer Section 3.3.3) do not apply to Android OS versions below 5.0, hence the necessity for APMD(s) with OS versions that are Lollipop and above.

4.2.1 Addressing Privacy Concerns

Since the research volunteers had to install a third party application (*ContextAnalyzer*), it was crucial to address any privacy concerns that arose. In order to gain the trust of the volunteers, we adopted a policy of full transparency and took the following measures:

- We fully explained to each volunteer, the nature of the data we were collecting i.e. cache metrics.
- We open sourced the Android application [51] in case the volunteers wanted to verify that there were no nefarious activities going on in the background.
- We did not automatically store the cache metrics in a server. Instead, we asked the volunteers to send us the data in order to convey the message that we were'nt collecting any private data that the volunteers were'nt aware of.

Although the aforementioned points address the privacy issues, we realize that divulging the nature of the experiment has the potential to bias the volunteers in their APMD usage. We explore this aspect in detail in the Points of Weaknesses section.

4.3 Process and Duration of Experiment

A total of twenty volunteers signed up and installed the application. They were divided into two groups:

- Those without a set duration of observation.
- Those with a set duration of observation (1 week).

The first group comprised of 9 volunteers and they were not monitored over a set duration of time. They installed the application at some point of time and had it running on their APMD(s). The only stipulation that group one was placed under was that they had to run the application for at least a week although there was no upper bound. Group two comprised of eleven volunteers. They had the application running for exactly one week. Both groups sent the data from the application and additionally provided the number of entries they entered in their calendar during the monitoring period. The link to the APK file was posted online [56] and each volunteer downloaded and installed the application as a third-party application. In this context, the term third-party refers to the fact that the Android application was'nt downloaded from the Google Play store. The volunteers were instructed not to manually stop the application for the duration of the experiment [52].

4.4 Context Analyzer Design

ContextAnalyzer computes the metrics for both the default scheme and the Hybrid scheme. In order to accomplish this, *ContextAnalyzer* reads the user's calendar (Section 3.3.6) and produces a list of applications that the user is likely to use. For the sake of simplicity, let's refer to this list as the calendar list. It also reads the list of processes residing in memory (Section 3.3.2). Let's refer to this list as the memory

list. Given this information, every application that the user requests falls in one of four categories:

- The application is present in neither list.
- The application is present in memory list but not the calendar list.
- The application is present in calendar list but not the memory list.
- The application is present in both lists.

If the application is present in neither list, it results in an overall cache miss. To elaborate, an overall cache miss implies that neither the list of processes in memory (cached by the default recency-based caching scheme) nor the list of applications predicted by reading the calendar, contained the requested application. If the application was present only in memory list, it results in an overall cache hit and a suggested cache miss i.e. the Hybrid scheme would've still had a hit but it was not present in the list of applications suggested by the calendar list. If the application was present only in the calendar list, it results in an overall cache hit and a suggested-exclusive cache hit, implying that the hit is solely a result of the calendar list's suggestion. This metric is important as it reveals the significance of the list of applications suggested by the calendar list. We'll discuss this further in Chapter 5. Finally, if the application was present in both lists, it results in an overall cache hit and suggested non-exclusive cache hit, implying that the hit would've occurred regardless of whether the application was present in the calendar list or not. The following snapshot (Figure 4-3) illustrates *ContextAnalyzer*'s UI.

It is worth noting that *ContextAnalyzer* collects sufficient data to compute the CHR of three distinct caching schemes:

- Hybrid Scheme

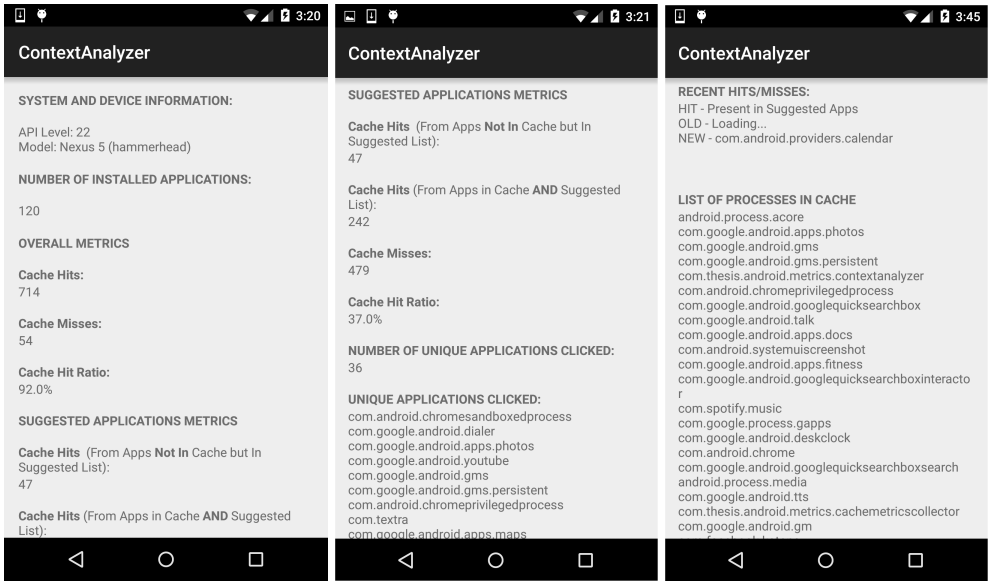

Figure 4-1: The images to the left, middle and right represent the top, middle and bottom portions of the UI respectively. Note that the Cache Hits in the Suggested Applications Metrics section is divided into suggested-exclusive and suggested non-exclusive. In addition to the cache metrics, it also displays some supplementary data such as the number of unique applications clicked, phone model, OS version and the number of installed applications. These details are discussed in Chapter 5 and are presented as demographic data. Finally, it also displays recent cache hits/misses, memory list and the calendar list purely for the user's benefit. It is not used in data analysis in any way.

- Pure Prediction Scheme
- Default Scheme

Since it collects the overall cache hits and overall cache misses, it is quite straightforward to see how it would compute the CHR of the Hybrid scheme.

$$HybridCHR = \frac{OverallHits}{OverallMisses} \tag{4.1}$$

The total hits for the pure prediction approach can be obtained by summing the suggested-exclusive hits and the suggested non-exclusive hits as their total comprises of all the hits that resulted due to the calendar list. The application also collects the

suggested misses, therefore the CHR for the pure prediction approach can be easily obtained.

$$TotalSuggestedHits = SuggestedExclusiveHits + SuggestedNonExclusiveHits \tag{4.2}$$

$$PurePredictionCHR = \frac{TotalSuggestedHits}{SuggestedMisses} \tag{4.3}$$

The total hits and misses for the default scheme can be derived from the data that *ContextAnalyzer* directly collects. Since it collects the overall cache hits (that includes default cache hits) and the suggested-exclusive cache hits, the default cache hits can be obtained by computing their difference:

$$DefaultHits = OverallHits - SuggestedExclusiveHits \tag{4.4}$$

Alternatively, we can obtain the default-exclusive cache hits by computing the difference of the overall misses and the suggested misses. This can then be added to suggested non-exclusive cache hits to obtain the total cache hits for the default scheme.

$$DefaultExclusiveHits = SuggestedMisses - OverallMisses \tag{4.5}$$

$$DefaultHits = DefaultExclusiveHits + SuggestedNonExclusiveHits \tag{4.6}$$

Every application click that results in a suggested-exclusive cache hit would by extension, also result in a default cache miss. Additionally, every application click that

results in an overall cache miss would also result in a default cache miss. Therefore the sum of these two fields would give us the default cache misses.

$$DefaultCacheMisses = OverallCacheMisses + SuggestedExclusiveCacheHits \tag{4.7}$$

The CHR for the default scheme can be computed using the following equation:

$$DefaultCacheHitRatio = \frac{DefaultCacheHits}{DefaultCacheMisses} \tag{4.8}$$

We've established that the application collects sufficient data to compute all the necessary metrics for the three caching schemes. Let's inspect the design of this application. *ContextAnalyzer* comprises of the following modules:

- Main Activity [7]
- Background Service [8]
- Process Manager
- Calendar Reader
- Calendar Parser

The main activity is the heart of the application. It is responsible for launching the application and creating a background service. The background service retrieves the list of processes in memory and the current foreground application. The background service is bound to the activity i.e. it's life-cycle is tied to that of the activity's [52]. It is for this reason that the volunteers are instructed not to forcefully close *ContextAnalyzer* for the duration of the experiment as that would destroy the service along with the activity and all metric collection would stop. The process manager module

contains a *Process* data structure that encapsulates a running process and all of its attributes (name, user identifier, process identifier to list a few). It also contains the algorithm that fetches the list of processes residing in memory. The background service launches a separate thread that invokes this algorithm every second, to compute the cache metrics and update the user interface. It also has the additional task of obtaining the calendar list from the calendar parser. The calendar parser gets the list of calendar events from the calendar reader and produces the calendar list (that the service consumes). The calendar reader queries the user's calendar API and obtains the list of events for the next four hours. The following class diagram (Refer Figure 4-4) illustrates *ContextAnalyzer*'s design.

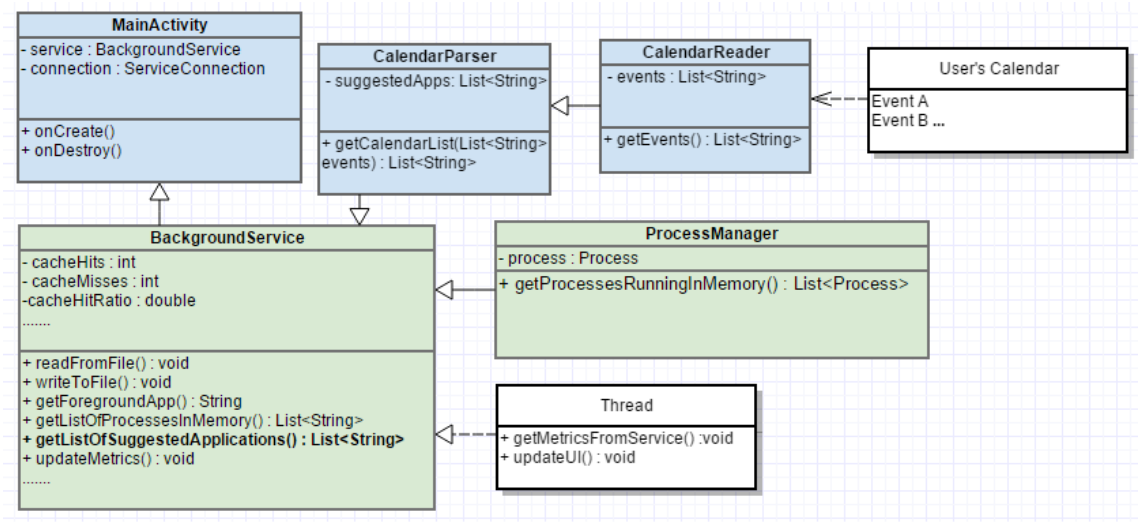

Figure 4-2: Context Analyzer Class Diagram

ContextAnalyzer was developed using Android Studio [53] and the source code was uploaded on Github [51]. The class diagram was built using Gliffy [54]. The APK file was uploaded on our Github I/O page [55][56].

Chapter 5

Data Analysis and Results

5.1 Introduction

5.2 Demographic Data

5.3 Experiment Results

5.4 Significance Testing

5.5 Points of Weakness

Chapter 6

Conclusion

The typical user is not facing a desktop machine in the relatively predictable office environment anymore. Smart-phones are well suited to utilize contextual information in enhancing the behavior of their applications. However, the full potential of context analysis is not harnessed by Android devices. Currently, the Android OS caches application components in memory, based on recency of application usage. We have demonstrated that incorporating context information improves the efficiency of this caching scheme. For this purpose, we set up an experiment comprising of twenty volunteers. We developed an Android application to measure the cache metrics of both the default recency-based caching scheme and a default-context hybrid scheme, which in addition to relying on recency of usage, parsed the user's calendar for contextual clues. We analyzed the data and proved that on an average, the hybrid scheme had a significantly higher cache hit ratio. By virtue of its superior efficiency, the hybrid scheme would result in a lower startup time for relatively more applications, thereby enhancing the overall user experience.

Chapter 7

Future Work

In addition to the user's calendar, other sources of information can be utilized to obtain context. These days, most APMD(s) have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful in monitoring three-dimensional device movement or positioning, or changes in the ambient environment near a device [21]. Information from multiple sensors can significantly improve the reliability of contextual data. Shi et al.'s work focused on constructing a user behavior model [22] based on analyzing user behavior. A similar approach can be used to predict the list of applications the user is likely to use in the immediate future. Currently the calendar parser relies on predetermined mapping between keywords and applications. This can me improved upon by learning the user's preferences. If the user predominantly requests an application, associated with a particular event in the calendar, then the mapping can be altered by learning the user's preferences. Finally, this thesis merely verifies whether a caching scheme that incorporates context information is more efficient than the default recency-based scheme. The next step would be to implement this caching scheme by building a custom distribution of the Android OS, facilitated by the AOSP [59].

References

[**1**] "The Android Source Code"
http://source.android.com/source/index.html, October 2015. Online; Last Accessed: October 22, 2015.

[**2**] "The Most Popular Smartphone Operating Systems Globally,"
http://www.lifehacker.com.au/2014/09/the-most-popular-smartphone-operating-systems-around-the-world/, September 2014. Online; Last Accessed: October 22, 2015.

[**3**] "Android System Architecture"
https://mahalelabs.files.wordpress.com/2013/03/androidstack-1.jpg, March 2013. Online; Last Accessed: October 22, 2015.

[**4**] "Best custom ROMs for the Samsung Galaxy S2"
http://www.cnet.com/uk/how-to/best-custom-roms-for-the-samsung-galaxy-s2/, April 2012. Online; Last Accessed: October 22, 2015.

[**5**] "Widgets — Android Developers"
http://developer.android.com/design/patterns/widgets.html, October 2015. Online; Last Accessed: October 22, 2015.

[**6**] "Application Fundamentals — Android Developers"
http://developer.android.com/guide/components/fundamentals.html, October 2015. Online; Last Accessed: October 22, 2015.

[**7**] "Activities — Android Developers"
http://developer.android.com/guide/components/activities.html, October 2015. Online; Last Accessed: October 22, 2015.

[**8**] "Services — Android Developers"
http://developer.android.com/guide/components/services.html, October 2015. Online; Last Accessed: October 22, 2015.

[**9**] "Wikimedia Upload"

http://www.extremetech.com/wp-content/uploads/2015/04/MooresLaw2.png, October 2015. Online; Last Accessed: October 22, 2015.

[**10**] "A Brief History of Android Phones"
http://www.cnet.com/news/a-brief-history-of-android-phones/, August 2011. Online; Last Accessed: October 22, 2015.

[**11**] "GSM Arena"
http://www.gsmarena.com/t_mobile_g1-2533.php, October 2008. Online; Last Accessed: October 22, 2015.

[**12**] "Nexus 6P"
http://www.androidcentral.com/nexus-6p, October 2015. Online; Last Accessed: October 22, 2015.

[**13**] "Phone Egg — LG T585"
http://us.phonegg.com/phone/4783-LG-T585, November 2013. Online; Last Accessed: October 22, 2015.

[**14**] "Android PSA: Stop Using Task Killer Apps"
http://phandroid.com/2011/06/16/android-psa-stop-using-task-killer-apps-now/, June 2011. Online; Last Accessed: October 22, 2015.

[**15**] "Processes and Application Life Cycle"
http://developer.android.com/guide/topics/processes/process-lifecycle.html, October 2015. Online; Last Accessed: October 22, 2015.

[**16**] "Managing Your App's Memory"
http://developer.android.com/training/articles/memory.html, October 2015. Online; Last Accessed: October 22, 2015.

[**17**] "Theodore Johnson, Dennis Shasha - A Low Overhead High Performance Buffer Management Replacement Algorithm"
http://www.vldb.org/conf/1994/P439.PDF, January 1994. Online; Last Accessed: October 22, 2015.

[**18**] "Hannu Verkasalo - Contextual patterns in mobile service usage"
http://lib.tkk.fi/Diss/2009/isbn9789512298440/isbn9789512298440.pdf, March 2008. Online; Last Accessed: October 23, 2015.

[**19**] "Mika Raento, Antti Oulasvirta, Renaud Petit, and Hannu Toivonen - ContextPhone: A Prototyping Platform for Context-Aware Mobile Applications"

http://dl.acm.org/citation.cfm?id=1070628, May 2005. Online; Last Accessed: October 23, 2015.

[20] "Anind K. Dey, Gregory D. Abowd and Daniel Salber - A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications "
http://www.cc.gatech.edu/fce/ctk/pubs/HCIJ16.pdf, January 2001. Online; Last Accessed: October 23, 2015.

[21] "Sensors — Android Developers"
http://developer.android.com/guide/topics/sensors/sensors_overview.html, October 2015. Online; Last Accessed: October 23, 2015.

[22] "Elaine Shi, Yuan Niu, Markus Jakobsson, and Richard Chow - Implicit Authentication through Learning User Behavior"
markus-jakobsson.com/wp-content/uploads/2010/11/ShiNiuJakobssonChow2010.pdf, November 2010. Online; Last Accessed: October 23, 2015.

[23] "Alan Jay Smith - Design of CPU Cache Memories"
http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-357.pdf, August 1987. Online; Last Accessed: October 23, 2015.

[24] "Cache Hit — Techopedia"
https://www.techopedia.com/definition/6306/cache-hit, October 2015. Online; Last Accessed: October 23, 2015.

[25] "Cache Miss — Techopedia"
https://www.techopedia.com/definition/6308/cache-miss, October 2015. Online; Last Accessed: October 23, 2015.

[26] "Najeeb A. Al-Samarraie - And Page Replacement Algorithms: Anomaly Cases"
http://www.iasj.net/iasj?func=fulltext&aId=50712, January 2003. Online; Last Accessed: October 23, 2015.

[27] "LRU Replacement Policy"
https://www.seas.upenn.edu/ cit595/cit595s10/handouts/LRUreplacementpolicy.pdf, October 2015. Online; Last Accessed: October 24, 2015.

[28] "Hong-Tai Chou, David J. Dewitt - An Evaluation of Buffer Management Strategies for Relational Database Systems"
http://www.vldb.org/conf/1985/P127.PDF, October 1985. Online; Last Accessed: October 24, 2015.

[**29**] "Shaul Dar, Michael J. Franklin, Bjorn T. Jonsson, Divesh Srivastava, Michael Tan - Semantic Data Caching and Replacement"
http://www.vldb.org/conf/1996/P330.PDF, October 1996. Online; Last Accessed: October 24, 2015.

[**30**] "Donghee Lee, Member, IEEE, Jongmoo Choi, Member, IEEE, Jong-Hun Kim, Member, IEEE, Sam H. Noh, Member, IEEE, Sang Lyul Min, Member, IEEE, Yookun Cho, Member, IEEE, and Chong Sang Kim, Senior Member, IEEE - LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies"
http://u.cs.biu.ac.il/ wiseman/2os/lru/lrfu.pdf, December 2001. Online; Last Accessed: October 24, 2015.

[**31**] "An Upper Limit For Apps? New Data Suggests Consumers Only Use Around Two Dozen Apps Per Month"
http://techcrunch.com/2014/07/01/an-upper-limit-for-apps-new-data-suggests-consumers-only-use-around-two-dozen-apps-per-month/, July 2014. Online; Last Accessed: October 24, 2015.

[**32**] "ARM Documentation"
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.cortexr/index.html, July 2014. Online; Last Accessed: October 24, 2015.

[**33**] "Does Your Android Device Really Need a Task Killer?"
http://www.technorms.com/40664/android-task-killer-apps, September 2014. Online; Last Accessed: October 24, 2015.

[**34**] "CyanogenMod"
http://www.cyanogenmod.org/, October 2015. Online; Last Accessed: October 24, 2015.

[**35**] "Activity Manager — Android Developers"
http://developer.android.com/reference/android/app/ActivityManager.html, October 2015. Online; Last Accessed: October 24, 2015.

[**36**] "Usage Stats Manager — Android Developers"
https://developer.android.com/reference/android/app/usage/UsageStatsManager.html, October 2015. Online; Last Accessed: October 24, 2015.

[**37**] "Android Debug Bridge — Android Developers"
http://developer.android.com/tools/help/adb.html, October 2015. Online; Last Accessed: October 24, 2015.

[**38**] "ADB Shell Commands — Android Developers"
http://developer.android.com/tools/help/shell.html, October 2015. Online; Last Accessed: October 24, 2015.

[**39**] "Android Toolbox"
http://elinux.org/Android_toolbox, October 2015. Online; Last Accessed: October 24, 2015.

[**40**] "Chainfire — Github — libsuperuser"
https://github.com/Chainfire/libsuperuser, October 2015. Online; Last Accessed: October 24, 2015.

[**41**] "Jared Rummler — Github — AndroidProcesses"
https://github.com/jaredrummler/AndroidProcesses, October 2015. Online; Last Accessed: October 24, 2015.

[**42**] "toolbox — Github"
https://github.com/android/platform_system_core/tree/master/toolbox, October 2015. Online; Last Accessed: October 24, 2015.

[**43**] "Low RAM Configuration — Android Source Code"
https://source.android.com/devices/tech/config/low-ram.html, October 2015. Online; Last Accessed: October 24, 2015.

[**44**] "OOM Killer"
http://linux-mm.org/OOM_Killer, October 2015. Online; Last Accessed: October 24, 2015.

[**45**] "Storage Options — Android Developers"
http://developer.android.com/guide/topics/data/data-storage.html, October 2015. Online; Last Accessed: October 24, 2015.

[**46**] "Calendar Provider — Android Developers"
http://developer.android.com/guide/topics/providers/calendar-provider.html, October 2015. Online; Last Accessed: October 24, 2015.

[**47**] "These are the 25 most popular mobile apps in America"
http://qz.com/481245/these-are-the-25-most-popular-2015-mobile-apps-in-america/, August 2015. Online; Last Accessed: October 24, 2015.

[**48**] "Google Play"
https://play.google.com/store, October 2015. Online; Last Accessed: October 24, 2015.

[**49**] "Package Manager — Android Developers"
http://developer.android.com/reference/android/content/pm/PackageManager.html, October 2015. Online; Last Accessed: October 24, 2015.

[**50**] "Chapter 4 - Processes"
http://www.tldp.org/LDP/tlk/kernel/processes.html, October 2015. Online; Last Accessed: October 24, 2015.

[**51**] "Context Analyzer — Github"
https://github.com/Buzz-Lightyear/Context-Analyzer, October 2015. Online; Last Accessed: October 24, 2015.

[**52**] "Bound Services — Android Developer Guide"
http://developer.android.com/guide/components/bound-services.html, October 2015. Online; Last Accessed: October 24, 2015.

[**53**] "Android Studio — Android Developer Guide"
http://developer.android.com/tools/studio/index.html, October 2015. Online; Last Accessed: October 24, 2015.

[**54**] "Gliffy"
https://www.gliffy.com/, October 2015. Online; Last Accessed: October 24, 2015.

[**55**] "Github Pages"
https://pages.github.com/, October 2015. Online; Last Accessed: October 24, 2015.

[**56**] "Buzz-Lightyear Github — Cache-Analyzer-APK"
http://buzz-lightyear.github.io/Cache-Analyzer-APK/, October 2015. Online; Last Accessed: October 24, 2015.

[**57**] "API Level 21 — Android Developers"
https://developer.android.com/about/versions/android-5.0.html, October 2015. Online; Last Accessed: October 28, 2015.

[**58**] "Google Inc Apps"
https://play.google.com/store/apps/developer?id=Google+Inc.&hl=en, October 2015. Online; Last Accessed: October 28, 2015.

[**59**] "Android Open Source Project"
https://source.android.com/, October 2015. Online; Last Accessed: October 28, 2015.