Minimizing Boot Time of Android Based Devices

Stage I Report

Submitted in partial fulfillment of the requirements for the degree of

Master of Technology

by

Nimit D. Kalaria
Roll No: 10305904

under the guidance of

Prof. D. B. Phatak

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
Contents

1 Introduction 1

2 Approaches to Minimize Boot Time 3
   2.1 Hibernation Based Boot . . . . . . . . . . . . . . . . . . . . . . . . . . 3
   2.2 Boot Optimization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

3 Android System 4
   3.1 Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
   3.2 Boot Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
      3.2.1 Init Script . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
      3.2.2 Zygote . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

4 Experiment/Implementation 9
   4.1 Experiment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      4.1.1 Modifying Zygote preloading . . . . . . . . . . . . . . . . . . . . . 9
      4.1.2 Changing Init.rc Script . . . . . . . . . . . . . . . . . . . . . . . . 12

5 Future Work 13

6 Conclusion 14
Abstract

Smart devices are becoming popular day by day due to its different features. These different features have their own initialization time which affects the boot time of the device. Boot time of the device should be minimal. Smart devices like phones, tablets are used by different people of different fields like industries, entertainment, students, etc. Where devices should boot quickly. Thus, it is necessary to minimize the boot time of smart devices. Boot time of the device can be minimized by either using a better hardware, or by optimizing the device boot sequence. We have gone with later approach to optimize the device boot sequence. For experimental setup we have selected an Open Source Android based emulator.
Chapter 1

Introduction

Usage of smart device is rapidly growing because of its different features. As the number of features are increasing, the initialization time of software modules is also increasing. Different hardware (i.e., Cameras, Bluetooth, WiFi, etc.) have their own long initialization time. This leads to increase in boot time of devices. To fulfill user demands for faster and reliable services, quick boot of time is necessary. Android, an Open Source project launched by Google in September 2008, became a popular platform for mobile phones and other embedded devices. Android system architecture composes of modified Linux kernel, middle-ware library, application framework and applications.

Quick boot up is important for good end-user performance. Android stack is packed with different functions which initialize after the device is switched on, and thus, results into long boot time of the device. To minimize the boot up time, we modify the list of functions which are initialized during boot time. This has some other disadvantages (like, increase in memory usage) but are not essential for performance. Various changes in the system can improve boot time of the device. For example, file system of storage will also be responsible for boot up time as, bunch of data are read from the storage. Reading kernel in compression and decompression mode also affects boot time. If device processor is slow, the device takes longer time to decompress, if it is in compression mode. It will take less time if it is in uncompressed mode. Android userspace is more time consuming module in boot sequence. How much time is consumed by userspace modules that we will see in next section. Android userspace contain parsing of “init.rc” script, starting up Zygote, package scanning and initializing core services.

When android OS is started it will start Zygote process, it preloads classes to create a shared memory area for applications. Android developers generate a default class preloading list with the aim of reducing overall memory usage of the system and application startup time, but they have not considered the effects of class preloading time on overall boot time[9]. Thus, this way memory usage is decreased on the cost of high boot time. Changes in Zygote class preloading can also reduce the boot time. But, changes in the number of preloaded classes results into increase in memory usage which is dis-
advantage of minimizing boot time. How much memory usage is increased while trying to reduce boot time by changing the number of preloaded classes is shown in experimental section. Changes in “init.rc” file can also reduce the boot time. Thereafter, avoiding package scanning and delaying startup of few process results in to low boot time. In this stage, we will focus on Zygote preloading and “init.rc” script.

Next chapters of paper is as follow. In chapter 2, we will see two different approaches to minimize boot time of Android devices. Chapter 3 will talk about Android system architecture and boot up process of Android devices. In boot up process section, we will see Zygote startup. Chapter 4 comprises of experiments and implementation. Chapter 5 and chapter 6 contains future work and conclusion respectively.
Chapter 2

Approaches to Minimize Boot Time

2.1 Hibernation Based Boot

This approach\cite{1}, takes snapshot of memory image of a fully booted Android device. Now, when users start their devices, the device will start from hibernation mode instead of normal boot up. This method reads a snapshot image from memory, and loads it into RAM. Thereafter, it will start the device. Here, the boot up sequence is modified, where the snapshot image is loaded and then the device is stared, rather than the core startup. The main disadvantage of this approach is to maintain the latest snapshot image. Thus, whenever an application is installed on the tablet and user want to start that application during boot time, the snapshot needs to be recreated.

2.2 Boot Optimization

In this approach\cite{9}, we optimize every component responsible for boot up. For example, boot-loader, kernel, Zygote class preloading, etc.. However, a system level approach to solve the boot time problem is absent because of the unavailability of a common, Open Source embedded software stack. We follow this approach but we will also modify $\text{init.rc}$ script of android device. In this stage, we modify two files $\text{init.rc}$ and $\text{Zygote}$ class preloading. Explanation about modification is given in next chapter. The disadvantage of this approach is the increase in memory usage while avoiding Zygote preloading. The next stage will overcome this disadvantage and will also reduce boot up time without affecting the memory usage.
Chapter 3

Android System

Android is an Open Source mobile device Operating System developed by Google, based on the Linux 2.6 kernel.[7] Since, Linux kernel have different driver model, memory and process management, network support, and other core services, it is easy to develop new OS for android devices.

3.1 Architecture

Android OS is a stack consisting of different modules. “Linux-kernel”, “Middleware layer(mainly library)”, “Application framework”, and “Applications”. All these layers are written in different languages. Android is using 2.6 Linux kernel written in ‘C’language. The Linux kernel was chosen due to its proven driver model, existing drivers, memory and process management, networking support along with other core operating system services. In addition to the Linux kernel, various libraries were added to the platform, in order to support higher functionality. The kernel enhancement of Android include, alarm driver, ashmem (Android Shared Memory Driver), binder driver(Inter-Process Communication Interface), power management, low memory killer, kernel debugger and logger[7].

Middleware layer is mainly a library layer which contains different libraries written in C and C++. Android has created their own C library called Bionic. Application-framework layer framework is written using C++ and JAVA. The application framework was created to provide the system libraries in a concise manner to the end-user applications. Application later is a third party software layer, where all applications are written using JAVA.

The main part of the Android is Java Runtime Engine(JRE), i.e. Dalvik Virtual Machine(DVM). DVM is used to optimize limited resources available on a mobile platform. Applications written in JAVA are presented as byte code. This byte code uses more memory footprint to execute the application. Android converts byte code to dex file format i.e. Dalvik executable format. This takes less memory footprint to execute the application. Apart from this, each application execute in its own DVM, so, if any one application crashes then, it will not affect the other running applications. DVM for every
application created by Zygote, the library will be shared among the applications.

![Android System Architecture](image)

Figure 3.1: Android System Architecture[7]

### 3.2 Boot Process

Boot up Android device is a time consuming process. As more functions will be added, it will take more time to boot. Android boot sequence is similar to Linux boot sequence. The only difference is in the initialization of user-space applications. Figure 3.2 shows Android boot sequence. The Android boot sequence[8][3] is explained below.

- When the device switches on, boot ROM code is executed, which is placed at a fixed location. This boot ROM code will load the first stage of the boot loader into the internal RAM.
- This first stage boot-loader will set up an external RAM and will then load the second stage boot-loader. Second stage will set up file system, low level memory protection, and network support.
- Once the second stage execution is completed, it will look for Linux kernel and thereafter, load it in to the RAM. Linux kernel will initialize memory management and caches. It will look into the root file system for init process and will then launch it.
- “Init” is the first user space process to execute with PID(Process ID) as 1. Init is placed at root node in the process tree. Every other process of the system will be launched by init process.
Init will execute the *init.rc* script and launch the system service processes. This is a script that describes the system services, file system and other parameters that needs to be set up.

- Init will start some daemon processes which are needed to start the system. It will also start “usbd”, “adbd”, “debuggerd”, and “rild daemon”.
- The first process to launch by init is Zygote, which will execute and initialize the Dalvik VM.
- Therafter, init will start runtime service and Service Manager. Service Manager is the context manager for Binder that handles service registration.
- Therafter, that runtime service will invoke Zygote to start System Server. Zygote will create copy of its DVM and will load System Server code.
- System Server application is the first application to fork from Zygote and it starts essential Android application framework services. It will also start Package Manager which will scan different packages.
- Activity Manager will then start applications like, phone application (com.android.phone), home, and few core applications.

After completing these steps, the device will display home screen. This takes normally 40-43 seconds on ARMv7 based emulator. Table 3.1 shows boot process time distribution. Output is collected by “*adb -d logcat*” commands. As referred in Table 3.1, *pms* stand for Package Manager Service and *ams* is Activity Manager Service. To get graphical output of boot process *bootchart*[6] tool is used. Figure 3.3 shows bootchart output of boot up process.

As shown in Figure 3.3, kernel initialization indicated by black line “No. 1” is takes approximately 3 seconds time. Zygote initialization indicated by black line “No. 2” is takes
Table 3.1: Logcat output of unoptimized Android Boot up.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Time(msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/boot_progress_start</td>
<td>3938</td>
</tr>
<tr>
<td>I/boot_progress_preload_start (Zygote)</td>
<td>5726</td>
</tr>
<tr>
<td>I/boot_progress_preload_end</td>
<td>18746</td>
</tr>
<tr>
<td>I/boot_progress_system_run (System Server)</td>
<td>19314</td>
</tr>
<tr>
<td>I/boot_progress_pms_start (Package Manager Service)</td>
<td>19655</td>
</tr>
<tr>
<td>I/boot_progress_pms_system_scan_start</td>
<td>20214</td>
</tr>
<tr>
<td>I/boot_progress_pms_data_scan_start</td>
<td>22856</td>
</tr>
<tr>
<td>I/boot_progress_pms_scan_end</td>
<td>22857</td>
</tr>
<tr>
<td>I/boot_progress_pms_ready</td>
<td>23128</td>
</tr>
<tr>
<td>I/boot_progress_ams_ready (Activity Manager Service)</td>
<td>26711</td>
</tr>
<tr>
<td>I/boot_progress_enable_screen</td>
<td>41329</td>
</tr>
</tbody>
</table>

Figure 3.3: Bootchart output of unoptimized Android Boot up.

approximately 13 seconds out of which 6.7 seconds are for preloading 2280 classes and 6.1 seconds are for preloading 278 resources. Table 3.2 displays this output. Therafter, that Packager manager scans packages, which takes around 3 seconds to scan packages. Activity Manager takes 14 seconds to start different services. Home screen is then displayed after some delay. Thus, optimization is needed at different phases of the boot sequence process. We will optimize init.rc script and Zygote preloading classes.

Table 3.2: Logcat output of preloading classes and resources

<table>
<thead>
<tr>
<th>Tag</th>
<th>Time(msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/Zygote (37): ...preloaded 2280 classes</td>
<td>6797</td>
</tr>
<tr>
<td>I/Zygote (37): ...preloaded 278 resources</td>
<td>6101</td>
</tr>
</tbody>
</table>
3.2.1 Init Script

Init process parses “init.rc” and “init.<machine_name>.rc” and starts other processes listed in these files. “Init.rc” file contains many processes, out of which we can avoid their startup. For example, Bluetooth and WiFi device startup can be avoided. Init will start boot animation which is CPU consuming process and we can avoid it’s startup and can use that amount of CPU for other process initialization. In experiment section, it is shown that avoiding only boot up animation will save around 5 seconds of time.

3.2.2 Zygote

Zygote[4] is the first process started by the init process. All applications and services start only through Zygote. Before starting an application, it share running DVM code and memory information for reducing application starting time. When Zygote gets started, the first task of Zygote, is to register a socket to listen fork request for new application. Activity Manager sends request for creating applications on this socket. Once Zygote is started, it will preload the commonly used classes of different applications during Zygote’s startup. It preinitializes and preloads common Android classes in its heap, which increases the boot time. The reason is to share memory between different processes which is forked by Zygote. Reduced memory usage caused by these classes memory sharing, is a bonus for embedded systems that have limited memory capacity[9]. Thus, Android will save memory by preinitializing large number of classes during boot up. One can avoid the preloading of classes which will save boot up time, but will increase memory usage of the process. Preloading of classes can be avoided or reduced if the memory usage is not important.

Once class preloading is done, Zygote will create new DVM for System Server process. It will then load System Server process code into newly created DVM. Therafter, Zygote will go in loop mode to listen to the socket for creating new application request. Whenever a request is received, it will fork the process and generate a new process. Zygote will share memory code and library with this newly created process. Therafter, it will load requested application code into Dalvik VM dynamically[2]. Once this is done, Zygote will pass process control to the requested application.
Chapter 4

Experiment/Implementation

4.1 Experiment

The experiment uses an emulator which is based on Linux Kernel 2.6.29. Android 4.0.9 is emulated on 512MB RAM and ARM cortex-a8 processor. While conducting experiment, bootchart was not executing as it affects the boot time. For conducting this experiment, help is taken from [10][12][11]. File system used by emulator is yaffs2. Experiment is conducted by changing number of preloaded classes in Zygote process. By default, Zygote will preload 2280 classes in Android 4.0.9. Experiment is conducted for 0, 263, 515, 969 and 2280(default) classes. Which classes to choose for preloading was explained in previous section.

Memory utilization is monitored while reducing the number of preloaded classes. Since, less number of classes are preloaded, sharing will be less between different processes. Thus, individual memory utilization will increase. For monitoring memory usage “procrank” command is used. That gives total memory usage and also memory usage of individual processes.

Procrank command gives information about PSS(Proportional Set Size) and USS(Unique Set Size) for all running processes. Uss is a set of pages that are unique to a process. It is the amount of memory that would be freed if the application was terminated. Pss is the amount of memory shared equally among all processes, accounted in a way that the amount is divided evenly between the processes that share it. This is memory that would not be released if the process was terminated, but is indicative of the amount that this process is “contributing”.

4.1.1 Modifying Zygote preloading

In this experiment we change the number of classes preloaded by Zygote process which is loaded at boot up time. By default, preloading time is around 6.4 seconds which has reduced to 0.3 seconds by reducing the number of classes that are preloaded. No resources
are preloaded, thus, saved resources preloading time which is approximately 6.1 seconds. Table 4.1 is shows preloading time for different classes. It is observed that, by reducing the number of classes to be preloaded, the preloading time reduces.

As shown in table 4.1, preloading time of 263 classes is 0.324 seconds, which is 10 times of preloading time of 0 class. Still boot time of 268 classes is less than the boot time of 0 class. The reason is that, while preloading 263 classes most of the classes are already in memory. Thus, start-up process does not need to fetch all classes. Thus, achieving less boot up time. Also note that, while preloading 0 classes, process needs to fetch most of the classes from storage to memory, which in turn overall increases all over boot time. Thus, boot time of preloading 0 classes is slightly more than boot time of preloading 263 classes.

<table>
<thead>
<tr>
<th>Number of classes preloaded</th>
<th>Time to preload classes (sec)</th>
<th>Achieved Boot time (sec)</th>
<th>Boot time saved (sec) [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.036</td>
<td>30.557</td>
<td>9.900 [24.47%]</td>
</tr>
<tr>
<td>263</td>
<td>0.324</td>
<td>29.927</td>
<td>10.535 [26.03%]</td>
</tr>
<tr>
<td>515</td>
<td>0.926</td>
<td>30.738</td>
<td>9.725 [24.03%]</td>
</tr>
<tr>
<td>969</td>
<td>2.097</td>
<td>32.313</td>
<td>8.149 [20.13%]</td>
</tr>
<tr>
<td>2280 (Default)</td>
<td>6.400</td>
<td>40.463</td>
<td>0 [0%]</td>
</tr>
</tbody>
</table>

Table 4.1: Preloading different number of classes and preload time.

Table 4.1 is also shows by changing Zygote preloading how much boot time is saved. The boot time saved is different between default boot time and achieved boot time. In this, preloading 263 classes will save more than others. It is saving around 10.5 seconds boot time by preloading 263 classes and 0 resources. Now, the increase in memory usage should be monitored for saving this boot time. For this, we have collected total memory usage, total PSS and USS usage. Table 4.2 displays the total memory usage, total PSS, and USS usage, while preloading different classes.

<table>
<thead>
<tr>
<th>Number of classes preloaded</th>
<th>Memory Usage (MB)</th>
<th>Total PSS Memory Usage (MB)</th>
<th>Total USS Memory Usage (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>231.75</td>
<td>154.16</td>
<td>128.61</td>
</tr>
<tr>
<td>263</td>
<td>228.86</td>
<td>160.55</td>
<td>124.10</td>
</tr>
<tr>
<td>515</td>
<td>216.37</td>
<td>162.44</td>
<td>115.59</td>
</tr>
<tr>
<td>969</td>
<td>211.54</td>
<td>167.10</td>
<td>112.28</td>
</tr>
<tr>
<td>2280 (Default)</td>
<td>220.96</td>
<td>169.16</td>
<td>111.80</td>
</tr>
</tbody>
</table>

Table 4.2: Memory Usage of preloading different number of classes.
The memory usage data is collected after the device displays home screen and contains only default process that are started during boot up process. We can observe that memory usage is increasing while preloading less number of classes. But when preloading 969 and 515 number of classes, memory usage is less than the default memory usage. The reason is that preloading of other classes are avoided which are not used by process, started during boot up time. Thus, it decreases memory usage while preloading 969 and 515 classes. When we are preloading 263 classes, memory usage is increased, because, this class list does not contain most of the classes which are used by process started during boot up time. From Table 4.2 we can see that, memory usage for preloading 0 classes is increased by around 11MB(4.97%) and where as memory usage for preloading 263 classes is increased by around 7.93MB(3.57%). Thus, if we combine Table 4.1 and Table 4.2 data then it can be observed that it is good to preload 263 classes which is better in terms of boot up time and memory usage while comparing with preloading 0 classes.

Now, we will observe the utilization of memory is changed for normal process, while changing the number of preloading classes. For this, we have collected memory usage of Browser and Calculator. Table 4.3 displays results of the memory used for Browser and Calculator. Memory usage is collected in terms of PSS and USS of individual process. It is observed that PSS usage of both processes decreasing as the number of preloaded classes decrease. On the other side, USS usage of both process is increasing, as the number of preloaded classes is decrease. The reason for this is that as we are reducing the number of preloaded classes, the sharing of classes will reduce, This decrease PSS(Proportional Set Size) and the unshared classes are loaded by individual process which will increase USS(Unique Set Size) of processes.

<table>
<thead>
<tr>
<th>Number of classes</th>
<th>Browser</th>
<th>Calculator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PSS (MB)</td>
<td>USS (MB)</td>
</tr>
<tr>
<td>0</td>
<td>31.69</td>
<td>32.93</td>
</tr>
<tr>
<td>263</td>
<td>32.97</td>
<td>32.39</td>
</tr>
<tr>
<td>515</td>
<td>34.17</td>
<td>31.58</td>
</tr>
<tr>
<td>969</td>
<td>34.88</td>
<td>30.24</td>
</tr>
<tr>
<td>2280(DefaultValue)</td>
<td>34.96</td>
<td>28.76</td>
</tr>
</tbody>
</table>

Table 4.3: Memory Usage of Browser and Calculator.

While preloading 263 classes, USS memory usage for Browser is increased by 3.63 MB(12.62%) and for Calculator is increased by 0.3 MB(8.4%). While preloading 0 classes, USS memory usage for Browser is increased by 4.17 MB(14.49%) and for Calculator is increased by 0.73 MB(20.44%). Thus, it is observed that, when we are changing the number of preloaded classes, memory usage of individual process is not increased by a great extent. Good result is given by preloading 263 classes as compared to preloading 0 classes.
Thus, only changing in Zygote class preloading is saving approximately 26% of boot time with cost of 4% increase in memory usage.

<table>
<thead>
<tr>
<th>Number of classes preloaded</th>
<th>Achieved Boot time (sec)</th>
<th>Boot time saved (sec)</th>
<th>Boot time saved (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25.749</td>
<td>14.714</td>
<td>36.36%</td>
</tr>
<tr>
<td>263</td>
<td>25.516</td>
<td>14.946</td>
<td>36.93%</td>
</tr>
<tr>
<td>515</td>
<td>26.531</td>
<td>13.932</td>
<td>34.43%</td>
</tr>
<tr>
<td>969</td>
<td>26.639</td>
<td>13.823</td>
<td>34.16%</td>
</tr>
<tr>
<td>2280(Default)</td>
<td>35.563</td>
<td>4.900</td>
<td>12.10%</td>
</tr>
</tbody>
</table>

Table 4.4: Avoiding boot up animation and calculating boot up time.

4.1.2 Changing Init.rc Script

Apart from avoiding Zygote class preloading, we can also reduce time by making changes in init.rc file, which is executed by init file during boot up. Initial changes are made in this script to avoiding boot up animation. Boot up animation in android takes approximately around 5 seconds of time. It is consuming CPU for displaying animation which can be used for other process to initialize. Avoiding boot up animation will not affect any memory usage thus we can avoid it and reduce the boot time. Table 4.4 displays result recorded for avoiding boot animation. In table 4.4, Boot time saved is different between default boot time(40.463 seconds) and achieved boot time without boot animation.

It is observed from table 4.4 that, while preloading 263 classes(without using boot animation), saves 14.946 seconds as compared to preloading 0 classes(without using boot animation), saves 14.714 second. Thus, saving approximately 5 seconds without any memory increase in memory usage. Now, all over approximately 37% of the boot time is saved.
Chapter 5

Future Work

Apart from the Zygote class preloading, the other time consuming part is scanning of different packages by Package Manager and starting of different application by Activity Manager. Thus, the next stage will contain optimizing Package Manager scan and delaying starting of different processes by Activity Manager. The other task is to optimize kernel and boot-loader which are mainly in low level hardware. But optimizing of kernel and bootloader will save around 1.5-2 second of boot up time.

Next stage will also avoid increase in memory usage due to Zygote preloading. This can be done either by creating Zygote as multi-threaded and one thread will load all the preloaded classes while other threads do other task or make Zygote memory dynamic. Former implementation will be complex since we must septate task of both threads. In later case whenever any process tries to load classes which used to be preloaded, those classes will preload it in Zygote memory area and link it with that process. This is more reliable then former approach, because every application is started by Zygote only and when creating DVM image for application Zygote can load required classes in it’s own heap and then it will create DVM image for application. This way it is possible to save memory usage which is increased while trying to save boot time.
Chapter 6

Conclusion

Modifying Zygote preloading, and avoiding boot up animation time, we have saved 15 seconds(or 37%) of boot up time. And, to save this much boot time 4% of memory usage is increased. Now, our boot up time is reduced from 40 seconds to 25 seconds around from 40 seconds of boot up time. While decreasing boot up time we have one side effects of memory usage which is increased as we have decreased Zygote preloading. But this will be not an issue for devices which have enough memory. We can reduce more boot up time by changing Package Manger and Activity Manager and can be possible to boot up in around 5 seconds.
Acknowledgements

I would like to thank Prof. D. B. Phatak for helping and suggesting appropriate reading material, and for giving me his valuable time for clearing my doubts. I would also like to thank Nagesh Karmali for his continuous inputs in my work.
Bibliography


