Optimizing the boot time of Android on embedded system

Article *in* Digest of Technical Papers - IEEE International Conference on Consumer Electronics · June 2011

DOI: 10.1109/ISCE.2011.5973881

Gaurav Singh
Hi-Tech Robotic Systemz Ltd.
1 PUBLICATION **10** CITATIONS

Optimizing the Boot Time of Android on Embedded System

Gaurav Singh, Kumar Bipin, Rohit Dhawan
Computing Platforms and Tools
STMicroelectronics
Greater Noida, India
{gaurav-mmc.singh, kumar.bipin, rohit.dhawan}@st.com

Abstract—**Increasing hardware capabilities and application requirements in embedded systems demand additional software initialization and configuration during startup, which adversely affects system boot time. Fast boot is essential for consumer devices in automotive, medical and entertainment markets. This paper describes "system level" optimization of embedded software to achieve faster boot times. We select an embedded device running open source Android platform as the experimental setup for research. First, we describe an efficient bootloader design and explain how to optimally configure Android's Linux based kernel for embedded systems. Next, we detail Android userspace design changes to reach the home screen quickly and allow users to execute crucial applications first. We also discuss effects on memory consumption, application and feature availability caused by optimization changes in each part of the software stack. Finally, we show that our optimized Android stack boots 65 percent faster than the existing common approach**

Keywords— Fast boot, Android, Linux, bootloader, optimization.

I. INTRODUCTION

The scope of embedded systems is rapidly growing because of hardware innovations and increased penetration of embedded devices into diverse application areas such as automotive, telecommunications and entertainment. Embedded software needs to keep pace with growing hardware and application complexity and must also fulfil user demands for faster and more reliable services. Size, power and price constraints of embedded systems make software design and implementation uniquely difficult for engineers. Till recently, the demand for embedded software was met by a few proprietary platforms. In September 2008, an open source software platform primarily for smart mobile phones called Android was launched by the Open Handset alliance led by Google. Android is a complete embedded software stack, comprising of a modified Linux based kernel, middleware, application framework and applications. To run Android on their platform, Original Equipment Manufacturers (OEMs) need to make hardware specific changes such as adding the bootloader and writing kernel drivers for their devices. Android has quickly become a popular platform for mobile phones and is being adopted in other products as well.

Booting up is use case number one for embedded systems and fast boot times are important for impatient end-users. Embedded software platforms like Android are considerably large, and come packed with features to support converging applications. They take time to initialize after power-on, resulting in large system boot times.

Prior work to optimize embedded system boot time focussed on methods to individually speed up distinct software components: kernel, root filesystem selection, userspace init scripts 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 address this problem by using the Android platform for boot time optimization research. Our experimental setup consists of an ARM11 based automotive device for GPS navigation and entertainment, running Android ported for the hardware. Working on this setup, we generate a comprehensive list of system wide software optimizations to achieve faster boot times for Linux based embedded platforms.

We design a fast single-stage bootloader, tune the kernel configuration and intelligently select our platform filesystems (based on relative filesystem performance data) to reduce the kernel load and startup time by 76 percent. Android userspace initialization begins after kernel startup and we identify preloading of Dalvik classes (Android uses Dalvik virtual machine or DVM) along with scanning of application and system packages as main causes of startup latency in this stage. During Android boot, DVM preloads Dalvik 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. We collect dynamic class memory usage information and load time of individual classes on the experimental setup. The data is then fed into our algorithm-based optimizer engine to generate a specific class preloading list that reduces preloading time as much as possible while respecting system memory constraints. Another important contribution of our research is the design of an Android system application called Delayloading.apk that scans and installs Android application packages and starts low priority user services during periods of user inactivity. Deploying the bootloader, kernel and Android userspace optimizations together on the setup, achieves a 65 percent reduction in boot time.

II. Related Work

Research by Kyung Ho Chung et al. [1] attempts to analyze the startup time of bootloader and the Linux kernel and also compares the performance of several filesystems for fast boot. However, contents of this study are too brief to address the implementation challenges faced by engineers. For example, the paper does not examine complexities of userspace initialization. We have designed a bootloader with small memory footprint that is more efficient than the bootloader discussed by this work. We provide an experimental analysis of kernel configuration changes to transform general purpose Linux kernel to a form more suitable for embedded platforms. Our work also examines the userspace initialization of Android stack and recommends changes to reduce boot time.

Open source software developers' present a study of filesystems such as UBIFS, YAFFS2 and SQUASHFS, comparing mount times and listing their relative pros and cons in [4]. This work provides us guidelines for the intelligent selection of filesystems. The paper by Dey, S et al. [2] introduces a filesystem partitioning scheme along with a self-adaptive mounting order for filesystems. In our research we have worked towards delayed mounting of filesystem partitions and the selection of an eclectic mix of filesystems for partitions. For example we use SQUASHFS for read only partitions and most efficient UBIFS for read-write partitions (explained in Section VI).

Another case study by Heeseung Jo et al. [3] analyzes the startup latency of a commercial digital TV and describes ways to use free system resources for delayed software initialization. The work is specific for a particular product line and hardware platform. We have used the widely popular and accepted Android platform, analyzed its startup and delayed software initialization based on event generation by Android framework itself. This approach is generic and can be applied to any phone, set top box or tablet running Android.

III. Android Boot Procedure

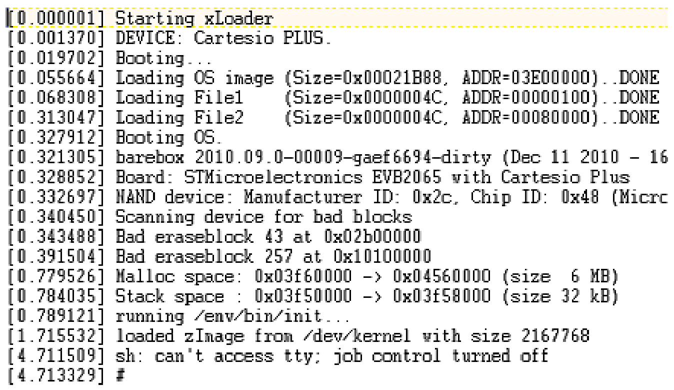
Booting of an Android based embedded system starts at power-on and ends when the Android "home screen" is displayed. At power-on, internal bootloader in embedded ROM checks non-volatile memory (such as NAND) for the vendor-supplied, vendor bootloader image. Vendor bootloader contains a device specific frame structure that is authenticated by the internal bootloader. Hence, internal bootloader acts as a gatekeeper and prevents unauthorized system boot. The valid vendor bootloader is loaded into Static RAM (SRAM) where it performs a few crucial operations such as system wide clock setting and Dynamic RAM (DRAM) configuration. Then, it loads the third stage bootloader from non-volatile memory into DRAM. The third stage bootloader initializes more system peripherals and copies the Android kernel image into DRAM, relinquishing control to the kernel after passing the appropriate boot parameters.

The kernel initializes drivers for device management, starts the process scheduler for process management, sets up the system memory allocator and mounts the root filesystem. Up till this stage Android boot is very similar to normal Linux system boot. The last step is Android userspace initialization (AUI) which begins with execution of Android's init program, and ends when user input commands to system can be accepted after display of "home screen". Unlike other Linux based embedded systems — which use combinations of /etc/inittab and init programs included in busybox — Android uses a custom init program which parses an init.rc script containing commands for filesystem mounting, property setting and starting of Android services [5]. For further details on Android userspace initialization refer to startup walkthrough slides in [14]. We now propose to optimize each step of boot starting with the bootloader.

IV. Bootloader Design

In embedded systems, existing bootloaders run in three stages. On our setup, there is a significant boot time contribution of ~0.8 seconds from the three stage bootloader (see Grabserial [16] output of kernel boot - Figure 1) that motivates us to improve the existing design. We focus our optimization effort on the vendor bootloader and the third stage bootloader. U-boot and Barebox are examples of third stage bootloaders that provide a rich set of features such as command prompt processing, device support and filesystem support – features that are required in development systems and not in products. Third stage bootloader is also responsible for copying the kernel into DRAM and transferring control to it. We propose to eliminate the third stage bootloader by combining its essential features into the vendor bootloader, creating a single stage bootloader. The single stage bootloader is small enough to run in the SRAM of 14KB on our setup and its design constructs are as follows:

- Add capability to load kernel from non-volatile memory. U-Boot or Barebox only duplicates these activities.
- Configure only those hardware devices that are genuinely indispensible for system startup such as system clock setting, RAM initialization, and configuration of NAND.
- System clock should be set to highest possible stable



```
[0.000001] Starting xLoader
[0.001370] DEVICE: Cartesio PLUS.
[0.019702] Booting...
[0.055664] Loading OS image (Size=0x00021B88, ADDR=03E00000)..DONE
[0.068308] Loading File1    (Size=0x0000004C, ADDR=00000100)..DONE
[0.313047] Loading File2    (Size=0x0000004C, ADDR=00080000)..DONE
[0.327912] Booting OS.
[0.321305] barebox 2010.09.0-00009-gaef6694-dirty (Dec 11 2010 - 16
[0.328852] Board: STMicroelectronics EVB2065 with Cartesio Plus
[0.332697] NAND device: Manufacturer ID: 0x2c, Chip ID: 0x48 (Micrc
[0.340450] Scanning device for bad blocks
[0.343488] Bad eraseblock 43 at 0x02b00000
[0.391504] Bad eraseblock 257 at 0x10100000
[0.779526] Malloc space: 0x03f60000 -> 0x04560000 (size  6 MB)
[0.784035] Stack space : 0x03f50000 -> 0x03f58000 (size 32 kB)
[0.789121] running /env/bin/init...
[1.715532] loaded zImage from /dev/kernel with size 2167768
[4.711509] sh: can't access tty; job control turned off
[4.713329] #
```

Figure 1. Grabserial output for standard Linux boot

frequency. It can later be set to an optimal value by power management module of kernel.
- Use optimized assembly code for register configuration and data payload copy.
- Interleaved Thumb and ARM mode compilation for optimal size footprint for lower bootloader load time and fitting into SRAM.
- Error Correction Code (ECC) checks and Bad Block table (BBT) generation are not compulsory for every boot when using pre-validated NAND.

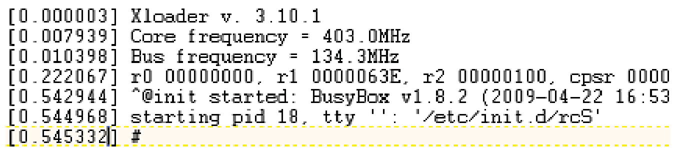
Our single stage bootloader boots in ~0.2 seconds that represents a 75 percent improvement over existing common approach (Comparison of Grabserial outputs in Figure 1 and Figure 2).

V. LINUX KERNEL

Android relies on open source Linux kernel version 2.6 for core operating system services such as security, memory management, process management, network stack and device driver model. Android has modified the Linux kernel [5] by adding components such as Binder (Inter-Process Communication mechanism), Ashmem, Pmem, Low Memory Killer (for memory management), Logger, Alarm Framework and Wakelocks (for aggressive power management).

To focus optimization effort on the slowest parts of kernel startup, we need to find the kernel startup time break-up. Enabling CONFIG_PRINTK_TIME option gives us initialization time of kernel components and helps derive the time break-up. Reducing kernel size will shorten the load time of kernel from slow non-volatile memory. The initial kernel load and startup time was ~4.7 seconds (Figure 1). We generate the following comprehensive list of procedures for kernel optimization to reduce this time:

- Uncompressed kernel image (Image) is more favourable than compressed one (zImage) on our setup. We found that the sum of zImage decompression time and zImage load time was larger than the load time of Image. Decision to use Image saved ~500ms during boot.
- Calibration of "loops_per_jiffy" during kernel initialization kernel consumes ~250 ms irrespective of CPU clock frequency. Find this value once and pass it to kernel through the command line in each subsequent startup [4].
- Root filesystem on flash deprecates the use of initrd or initramfs (RAM root file system). This eliminates effect of decompression time, load time and copy to buffer cache time.
- Scanning of whole NAND partition for BBT duringboot can be skipped. Filesystems such as UBIFS will handle bad blocks during NAND read/write operations after boot (Saving ~500ms).
- Embedded filesystems mostly use block device access to MTD layer for non-volatile memory (and not direct char device access). It is also possible to remove support for RAID, LVM as well as IDE support.
- Inhibit specific filesystem features such as Dnotify, Inotify and XFS. This will help reduce kernel size.
- Special use case of embedded system encourages us to use the memory slub allocator that is more space efficient.
- Highly threaded behaviour of small systems is inefficient for asynchronous I/O and paging of anonymous memory (swap). These can be disabled.
- Networking config options should be selected more intelligently according to product requirement. All functionalities are not required for most embedded applications. Options for kernel autoconf, multicast, advance router, tunnelling etc. can be disabled.
- Kernel hot-plugging support can be disabled for devices that don't use hot-plugging and firmware loading.
- The module unloading support can be removed, as once system has started it not to unload any active components.



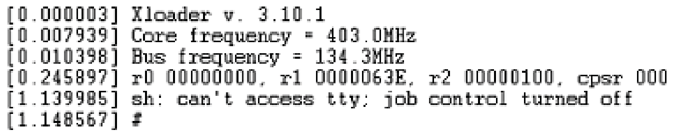
```
[0.000003] Xloader v. 3.10.1
[0.007939] Core frequency = 403.0MHz
[0.010398] Bus frequency = 134.3MHz
[0.222067] r0 00000000, r1 0000063E, r2 00000100, cpsr 0000
[0.542944] ^@init started: BusyBox v1.8.2 (2009-04-22 16:53
[0.544968] starting pid 18, tty '': '/etc/init.d/rcS'
[0.545332] #
```

Figure 2. Grabserial output for optimized Linux boot

During development, the kernel's debugging and profiling support is needed. However, in production systems we can omit the following debugging and profiling related kernel configuration options:

- Serial console is a slow device mostly used for debugging. Adding "quiet" option to kernel command line disables boot messages on console [4].
- Disable support for kernel symbol table. This is different from gcc-g.
- Removal of all configuration and functionalities used for debugging or profiling such as Oprofile, kprobe, kgdb as well as support for config information reduces kernel size substantially.
- A "brute force" approach - Disable printk support to completely eliminate kernel prints. This reduces kernel size significantly.

Optimizing the kernel configuration results in ~0.5 second boot of Linux kernel measured from power-on to console prompt of minimal root filesystem (Figure 2). The Android kernel boot time is 1.14 seconds (Figure 3). This difference is



```
[0.000003] Xloader v. 3.10.1
[0.007939] Core frequency = 403.0MHz
[0.010398] Bus frequency = 134.3MHz
[0.245897] r0 00000000, r1 0000063E, r2 00000100, cpsr 000
[1.139985] sh: can't access tty; job control turned off
[1.148567] #
```

Figure 3. Grabserial output for optimized Android kernel boot

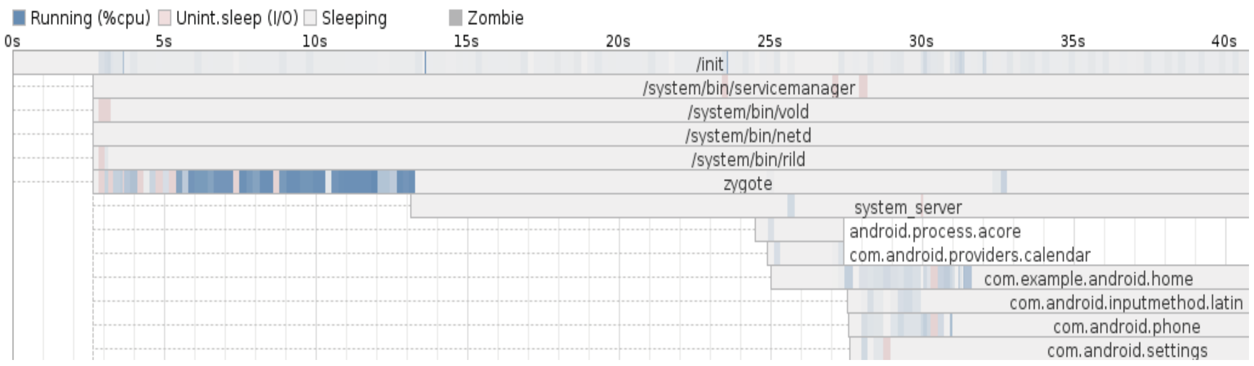


Figure 4. Bootchart output for un-optimized Android system

due to the Android kernel's Android specific components (pmem, wakelocks etc.), along with other components that fulfil user requirements (audio, video, USB, networking etc).

VI. Optimizing Android Userspace Intialization

Android's custom init program first mounts filesystems required by the platform. It is recommended to use a read-only System partition that contains static system information along with three read-write partitions namely Root, Data and User. SQUASHFS is a read-only filesystem that stores data in compressed form and gives best filesystem mount and read performance [4] while UBIFS gives the fastest mount and read-write performance among read-write filesystems [4]. Hence, we use SQUASHFS on the read-only System partition and UBIFS on the rest three partitions.

After mounting filesystems, init starts Android services such as vold (volume manager daemon), rild (radio interface layer) and adbd (Android debug bridge daemon). Then, init launches the nascent Dalvik virtual machine process called Zygote from which all Android applications are forked. DVM is designed to efficiently run multiple instances of virtual machine on embedded systems [6] and it executes Dalvik bytecodes that are smaller and faster than Java bytecodes [6]. System Server application is the first application to fork from Zygote and it starts essential Android application framework services - Power Manager (for power management), Package Manager (to manage application packages), Activity Manager (to manage application lifecycles), Account Manager, Battery Service etc. Finally the "home screen" is displayed by com.example.android.home process launched by the Activity Manager. For identifying bottlenecks in Android startup, we collect a Bootchart representation [12] of un-optimized Android boot (Figure 4). Potential areas of improvement are:

- Zygote process that executes in 10.4 seconds.
- System Server process that executes in 12 seconds.

com.example.android.home starts after 25 seconds which is the initial Android Userspace Initialization (AUI) time. The following sub-sections describe our techniques to minimize AUI time.

A. *Class Preloading Optimization*

Zygote preloads large number of Dalvik classes during AUI allowing Android applications that fork from Zygote to share common class memory sections [6]. Reduced memory usage — caused by this class memory sharing — is a bonus for embedded systems that have limited memory capacity. Class preloading also enables Android applications to startup faster, pleasing end users. Logcat output on our setup shows that class preloading takes a significant time of 8 seconds during AUI. Therefore, its advantages come at the cost of increased system boot time [7].

In our research, we generate a custom class preloading list to achieve fastest possible boot time while keeping extra memory overhead within a limit. Application startup time was excluded from our study, as it was less important for our users. In the experiment stage, we boot the Android setup with class preloading disabled and manually simulate the application usage pattern. All Dalvik classes are instrumented to give information on four parameters:

- Is the class loaded at boot time, either by zygote or applications that start before "home screen" display?
- Number of times class is loaded during the experiment.
- Average load time.
- Total heap memory size of class.

The first parameter is used to identify classes loaded during boot. Excluding such classes from class preloading list will not improve the system boot time and their inclusion in the list will save memory. Hence, these classes are unconditionally included in the list. The remaining classes each have a negative "weight", defined as the memory usage overhead incurred if the class memory is not shared using preloading. Also associated with each class is a positive "profit", defined as boot time saved if class is not preloaded. The "weight" is found using data for number of times the class is loaded and class heap size measurements while the "profit" is nothing but class load time. We need to generate an optimized class preloading list to maximize positive "profit" or time benefit while keeping negative "weight" increase or memory overhead within a particular limit. Hence, we apply a solution to the classical combinatorial optimization problem – the 0/1 Knapsack Problem (KP) to eliminate classes from the preloading list.

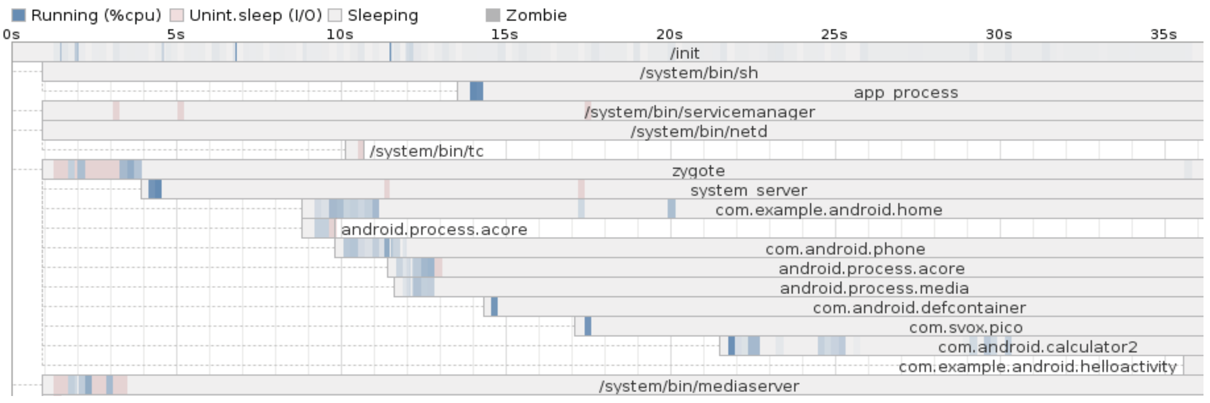


Figure 5. Bootchart output for optimized Android system

Memory overhead limit was fixed at 946 KB, the figure heuristically determined to be maximum memory usage increase that our system can absorb without affecting performance. A detailed description of KP and the algorithms used to solve it can be found in [15].

The final class preloading list consists of classes loaded during boot and classes not eliminated after we solved the KP algorithm step. On our setup, the final list gives a class preloading time of 0.8 seconds and peak runtime memory usage of 51 MB (measured by procrank utility). With the default preloading list, the peak runtime memory usage is 50.1 MB and class preloading time is 8 seconds. Therefore, our optimization saves 7.2 seconds during boot while causing a small 1.8 percent increase in memory usage.

B. *Delaying of Package Loading and Init Services*

Package Manager is called by the System Server to scan and install all available application packages. This operation takes 4.8 seconds on our un-optimized system as indicated by logcat utility. The services started by init process consume time and system resources during boot. We propose an implementation to perform delayed loading of packages and starting of services during user inactivity.

First, an application usage survey is used to identify the Android packages and services not required by users immediately after boot. Next, we move identified packages from System partition to the User partition (User partition is mounted after boot) and temporarily block non-essential services in init.rc script by making them wait on custom Android system properties. For example, consider a Service A waiting on property ro.user.start. We modify the system to include our custom Android application named Delayloading.apk that waits for an Intent generated by Application framework after boot called “android.intent.action.BOOT_COMPLETED”. After receiving the intent Delayloading.apk polls for user inactivity and sets ro.user.start property that wakes up the init process to start Service A (Figure 6). When user is inactive our application also invokes the “pm” utility that internally calls the Package Manager to install User filesystem packages (Figure 6). Use of Delayloading.apk helps reduce the System Server time to 4.9 seconds (Figure 5) as compared to 12 seconds in standard Android.

C. *Disabling of Java Native Interface (JNI) Checks, Android Debugging, Console and Bootanimation*

Dalvik Virtual Machine on Android uses JNI [8] to call native libraries from Java code. JNI checks test possible exceptions when transferring control to native code like null pointers, wrong array element types and invalid object parameters etc. The checks are time consuming and on production systems, where all applications are pre-tested, select Android’s “user” build configuration to disable JNI checks.

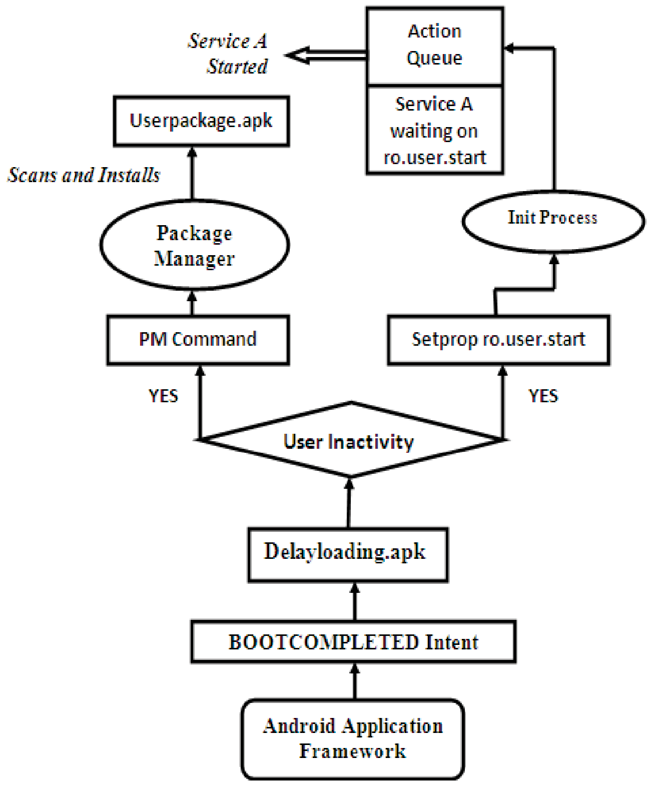


Figure 6. Design flow chart of Delayloading.apk

The Android debug bridge (adb) utility is used to debug Android target devices from the host's command line or via scripts. adb works as a client/server TCP-based application and a background process called adb daemon (adbd) runs on target devices during AUI to facilitate execution. Android debugging is not required for production systems and selecting Android's "user" build configuration disables adbd. Console can be disabled by removing the console service from the init.rc script. Bootanimation process is launched by default and can be disabled by setting appropriate system properties.

VII. Benchmarks And Evaluation

Our experimental setup is an ARM11 based device targeted for automotive navigation and entertainment that runs Android version 2.2 (Froyo) [11].

For determining kernel and bootloader startup time, we use the Grabserial open source tool that reads a serial port and writes timing information to standard host output. Figure 1, 2 and 3 are snapshots of Grabserial output taken while booting our setup with different configurations of kernel and bootloader. Figure 2 shows the startup time of optimized Linux system. We require ~0.5 sec (bootloader ~0.2 + kernel ~0.3 sec) to reach the console. This kernel startup time is competitive when compared to work done on Blackfin processors where a kernel startup time of 0.9 seconds is claimed [9].

Bootchart [12] is used to analyze AUI. Figure 4 shows that the AUI time is ~25 seconds initially. Our un-optimized setup takes 29.7 seconds to boot that includes kernel boot time of ~4.7 seconds and AUI time of ~25 seconds. Figure 5 shows that after optimization, AUI time is ~9.1 seconds. The optimized Android kernel takes ~1.1 seconds. Hence, the total boot time of the optimized setup is ~10.1 sec that represents a 65 percent improvement. Information on the internet reveals that the closest competitor to our boot performance is Pathpartner's Android fast boot implementation which takes ~18 seconds for boot [10].

VIII. Conclusion and Future work

Our proposed approach is aimed at tuning crucial sections of a complete system's software stack for achieving fast boot, with specific reference to the Android platform. The study and its results can be useful for manufacturers of Android based car navigation and entertainment systems for whom fast boot is extremely essential [13]. In consumer electronic devices like multimedia players and set top boxes implementation of a quick and efficient standby mode is important. Thus, suspend to RAM and suspend to non-volatile media can form subjects of further research.

Acknowledgment

We would like to thank Gian Antonio Sampietro, Giuseppe Gorgoglione, Melwyn Lobo and Abhijit (all from STMicroelectronics) for technical assistance. Special thanks are due to Sachin Verma and Saurabh Singh.

References

[1] Kyung Ho Chung; Myung Sil Choi; Kwang Seon Ahn; , "A Study on the Packaging for Fast Boot-up Time in the Embedded Linux," Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on , vol., no., pp.89-94, 21-24 Aug. 2007.

[2] Dey, S.; Dasgupta, R.; , "Fast Boot User Experience Using Adaptive Storage Partitioning," Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World: , vol., no., pp.113-118, 15-20 Nov. 2009.

[3] Heeseung Jo; Hwanju Kim; Jinkyu Jeong; Joonwon Lee; Seungryoul Maeng; , "Optimizing the startup time of embedded systems: a case study of digital TV," Consumer Electronics, IEEE Transactions on , vol.55, no.4, pp.2242-2247, November 2009.

[4] "Boot Time," http://elinux.org/Boot_Time.

[5] "Bring Up Android Open Source," http://source.android.com/porting/bring_up.html.

[6] Dan Bornstein, "Dalvik virtual machine internals- Google IO 2008," http://sites.google.com/site/io/dalvik-vm-internals.

[7] Tim Bird, "Improving Android Boot up time," http://elinux.org/images/4/4c/Android-boot up-time-linuxcon-2010-08.pdf.

[8] "The Java Native Interface Programmer's Guide and Specification," http://java.sun.com/docs/books/jni/html/intro.html#1811.

[9] "Boot Linux from Processor Reset into user space in less than 1 Second," https://docs.blackfin.uclinux.org/doku.php?id=fast_boot_example.

[10] "Fast boot up of Froyo edition of Android on Pathpartner mediaphone," http://www.youtube.com/watch?v=TzAlulCGqh4.

[11] "Android Developers," http://developer.android.com/index.html.

[12] "Bootchart," http://www.bootchart.org/.

[13] Sridharan Subramanian, "Leveraging Linuxo to Create an Auto Infotainment Platform – July 2009" http://www.freescale.com/files/training_pdf/VFTF09_AA114.pdf (turn to Page 7)

[14] Patrick Brady, "Anatomy and Physiology of an Android – 2008 Google I/O Session Videos and Slides", http://sites.google.com/site/io/anatomy--physiology-of-an-android.

[15] Silvano Martello and Paolo Toth, 1990, "Knapsack Problems: Algorithms and Computer Implementations," John Wiley & Sons, Inc., New York, NY, USA.

[16] "Grabserial," http://elinux.org/Grabserial.