Empirical assessment of the real-time capabilities of Android

CSE-605 Checkpoint 1 - Rex

Xia Wu
xiawu@buffalo.edu

Xingxiao Yuan
xingxiao@buffalo.edu

Michael Wehar
mwehar@buffalo.edu

Yihong Chen
ychen78@buffalo.edu

March 13, 2015

Contents

1 Introduction

Android is a mobile operating system based on the Linux kernel and is developed by Google. Along with the popularity of smartphones, 3G devices, and other advanced mobile technologies that emerged around 2009, Android has been standing out in the so-called "smart wars" against its counterparts Apple iOS and Windows, for its being a ready-made, low-cost, and customizable operating system (OS). According to Strategy Analytics's latest report (2014), Android is getting dangerously close to worldwide domination with a record 80% of all smartphone running the mobile OS. What's more, Android has gone a step further – it's the main driver behind the continuing climb

of the smartphone industry; Android now accounts for an impressive 8 in 10 of all smartphones shipped on the planet[1].

Enjoying such a big fat market share, Android facilitates its rapid deployment in many domains. Android was originally designed to be used in mobile computing applications, from handsets to tablets to e-books. But developers are also looking to employ Android in a variety of other embedded systems that have traditionally relied on the benefits of true real-time operating systems. What makes an operating system real-time is that on the top of functioning correctly it is also able to effectively meet deadlines. Time constraints for system performance are critical to such real-time embedded systems that mandate response to stimuli within pre-specified real-time design specifications. Reliability consideration of utilizing a system as real-time OS requires a detailed evaluation of the ability of this system to meet these specifications, in other words, to be predictable.

Android enables developers to write applications primarily in Java. Because of its managed memory system, traditional Java Virtual Machine (JVM) always raises concerns when it comes to real-time applications such as how well does Android's Process Virtual Machine Dalvik act in the real-time domain? This is what our project "Empirical assessment of the real-time capabilities of Android" is going to find out! While the majority of studies on the reliability of the Android system focus on the functional failures of activities; our research sets emphasis on studying the time-related behavior of Android for memory exhausting activities, analyzing the possible intrinsic obstacle(s) of the Android system for real-time applications. More specifically, we design/apply benchmarks suitable for the Android construct. We target the Android DragonBoard™ 800 development board that is based on the Qualcomm® Snapdragon™ 800 processor (APQ8074), observe Android garbage collector's performances, and test Android program predictability. At the end of the process, we hope to come up with a set of improvement suggestions that will help make Android more real-time friendly.

2 Android Components

- Intents and Intent Filters (receivers)

  We will evaluate the Intent delivery mechanism of Android. The Intents will be used for almost all the multiple processes experiments.

- Activities and App Widgets

  We will not directly evaluate the Activities and App Widgets because they're only related with UI. it's hard to produce convincing result as there are too many elements that can be out of control, e.g. the GPU power and the screen resolution.

- Services

  We will not directly evalute Services as well. The reason is the same as why we cannot directly evaluate Activities.

- Content Providers

  We will evaluate the **Garbage Collector**, **Synchronization**, and **Scheduler** by means of the Content Provider, because we can make use of the shared data mechanism that Android provides.

[1] `http://bgr.com/2014/07/01/android-market-share-2014/`

- Processes and Threads

  We will evaluate Processes and Threads directly. They're not only our targets of testing, but also the mechanisms that we are going to use in our experiments.

3 Intents/Receivers

3.1 Single Process

Although directly evaluating intents and receivers on single process deosn't make sense, it can provide a **baseline for our experiments**. We can compare the different results from the same experiment that is applied to single process and multiple processes; and we will analyze the different behaviors and try to reason about them.

3.2 Multiple Processes

We plan to use processes to generate a bunch of intents, then use different processes to receive the intents. In addition, there are always background processes working on some computations to produce different pressures on the Android system. This way we can observe the order and time of the intent delivery. These background working processes can also provide some observable pressure on **Garbage Collector**, **Synchronization**, and **Scheduler**.

3.2.1 Garbage Collector

We can use one sender and one receiver to test the Garbage Collector. We would like to evaluate how garbage collector works:

- frequency of grabage collection and memory pressure
- running time of grabage collection and memory pressure

The memory pressure should contains different types:

	big objects	medium objects	small objects
long live time	X	X	X
short live time	X	X	X

The key here is we need to generate pressures. To evaluate the behavior of Android system, it needs some pressure on different components so that we can infer the predictability of different components and the interaction between them. We think the pressure can come from:

- Other background processes with computation
- Computation inside senders
- Computation inside receivers

We will divide our experiments into three phases:
Phase 1. we only have pressure created from background process(es). It's easier to implement and tune for different memory pressure types.

Phase 2. we'll burden additional computations on senders and receivers so that we can compare whether the computation sources affects Android's performance.

Phase 3. we take into consideration the combination of the different pressures in our final evaluation.

The computation can be the benchmarks from SPECjvm2008 or DaCapo benchmark suites. Also, we can assoicate the Parcelable vs. Serializable with phase 3 of the experiments.

3.2.2 Synchronization

This task needs to involve other Android components. The reason is that we can not just pass an object as an extra intent, we need to serialize the object first. More difficultly, there is no directly synchronized mechanism between the sender and receiver. We are planning on passing some metadata to allow receivers to do some synchronization, for example, **Content Provider**. We will discuss this in more detail in the next section.

3.2.3 Scheduler

We can use multiple background processes to provide pressure on the scheduler. And next, we will use the order of the intent delivery to evaluate the scheduler and intent delivery mechanism. More on this topic is discussed in Section Processes/Threads.

4 Content Provider

Content provider is an Android system's mechanism to manage access to a central repository of data. However, Android system does not synchronize access to the Content Provider. So we must implement in a thread-safe way for accessing content providers. One method is to use synchronization. Normally, We are able to figure out whether or not there exists other synchronization mechanism.

But Android provides use cases to test synchronization. We can use different processes to access one content provider. Then we can evaluate the performance of synchronization in Android.

The experiments should contains **single process** as baseline and **multiple processes**. Every process accesses the content provider for a fixed times with a fixed interval, then the finish time is recorde. We plan to gradually increase the number of processes to compare the results

So our experiments should have such configurable elements as following:

Process Number	Times to Access Content Provider	Interval
How many processes run simultaneously	The fixed num	Interval

Obviously, we can evaluate the scheduler at the same time. For example, we can evaluate how many processes can run without lots of them missing deadline.

5 Processes/Threads

As we move forward with our project, we will be concerned with computational limitations in regards to memory, threads, and processes. We would especially like to understand how the Android system responds and performs as we approach the limits so that we can obtain a comprehensive set of data

about this performance. Before discussing the benchmarks that we will be running to obtain this data, we have to explore some basic information on how the Android System manages processes and threads.

There are some Android specific behaviours related to processes and threads[2]. For example, each component is associated with one or many processes. When memory is low some processes are killed. Preference is given to components that the user is currently interacting with. When a process is killed, it starts to back up when the user is again interacting with the component.

As a result, processes will be killed according to the following rules.

- Foreground processes are only killed as a last resort.
- Any process that affects what the user is currently working on will generally not be killed as well.
- Service processes may be killed if necessary.
- Background and empty processes are often killed first.

A thread is launched when an application starts. This thread is often called the UI thread (or the main thread). All components within that app will be instantiated within the UI thread. Applications are vulnerable to performance issues when large computations are a result of user interaction, because the application's UI thread will take on the task of handling these computations rather than handle simple UI tasks for a smooth user experience. In response to this vulnerability, Android has two principles to protect the user experience. One is don not assign too much work to the UI thread and another one is don not let other threads update the UI. So to handle large computations as a result of user interaction, one should spawn off worker threads.

In our project, we won't be too concerned with the application life cycle, the UI thread, components, and app related services. We will be more concerned with worker threads and computation done below the UI and service level. Since worker threads are often killed as a result of runtime configuration changes that result from user interactions, we will need to fix our runtime configuration and execute our application in a fixed environment where user interaction is limited or none.

Now, we are ready to discuss the benchmarks that we will be testing in Android. We were able to find some data on thread density for the the Dacapo benchmarks[3]. From this data, it appears that the benchmarks avrora_9, hsqldb_6, lusearch_6, and eclispe_9 all spawn off a lot of threads and will provide us with interesting and valuable data.

- avrora simulates the evaluation of programs on a grid of microcontrollers
- hsqldb has been replaced by h2 which simulates a model of banking with many transactions
- lusearch searches for keywords among a collection of large texts
- eclipse runs peformance tests related to the Eclipse IDE

[2]All of the specifc info on Android's system came from the Android Developers Guide found here: `http://developer.android.com/guide/components/processes-and-threads.html`

[3]Dacapo Benchmark Thread Density Data: `http://www.mm-net.org.uk/workshop230412/kalibera.pdf`

Also, we will investigate thread usage for the SPEC benchmarks such as compiler, compress, and crypto. We've found some relevant data on the 2015 SPECjvm2008 SPEC Summary Report[4].

In addition to these benchmarks, if we decide to further explore Android Services and Android thread management at a higher level, then we will look into bound services and interprocessor communication using remote procedure calls.

6 AlarmManager and Handler

There're two ways to schedule works in Android:

- AlarmManager
- Handler

Not missing the deadline for certain task is a critical factor for real-time system. So we'll evalute the two approaches respectively. But because these two approches have the same functionality, we call them as one name – scheduled tasks. To evaluate how well Android handle scheduled tasks, We can create lots of threads/processes with light computation running simultaneously, then we evaluate how many times they miss deadline.

Based on the two hypotheses that

- the number of threads/processes affect performance of scheduler
- the workload for each tasks affect performance of scheduler

We configure our experiments as follows:

1. only use light workload for lots of threads/processes
2. use configurable workload for fixed number of threads/processes to evalute how workload affects the scheduler

7 Parcelable/Serializable

According to this reference blog, parcelable mechanism have 10 times better performance than serializable mechanism. But parcelable needs developers to implement writeToParcel and createFromParcel manually. So parcelable can save the overhead to iterate all fields of object. But we can compare the two mechanisms by how much pressure they generate to garbage collector.

The approach is to pass same amount of objects from one process to another process (either the same process or alien), then we compare the different behaviors of garbage collector. It's possible to evaluate scheduler as well.

In conclude, the parcelable and serializable mechanisms are methods to provide pressure on Android system. In the meantime, we can evalute the performance of them. The result may improve static code analysis of Andorid codes.

[4]SPECjvm2008 SPEC Summary Report: `https://www.spec.org/jvm2008/results/res2015q1/jvm2008-20150120-00018.base/SPECjvm2008.base.html`