

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	1
2	Android Components	2
3	Intents/Receivers	3
3.1	Single Process	3
3.2	Multiple Processes	3
3.2.1	Garbage Collector	3
3.2.2	Synchronization	4
3.2.3	Scheduler	4
4	Content Providers	4
5	Processes/Threads	4
6	AlarmManager and Handler	5
7	Parcelable/Serializable	6

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 applications around 2009, Android has been standing out in the so-called “smart wars” with its counterpart such as Apple iOS and Windows, for its being a ready-made, low-cost, and customizable operating system (OS). According to Strategy Analytics’ 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.

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, its ability to 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, 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 majority of studies on the reliability of Android system focus upon 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 Android system for real-time applications. More specifically, we design/apply benchmarks suitable for Android construct – targeting at 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's predictability. At the end of the process, we hope to come up with a set of improvement suggestion to help Android more real-time friendly.

2 Android Components

- Intents and Intent Filters (receivers)

We will evaluate intent delivery mechanism of Android. The intents will appear for almost all multiple processes experiments.

- Activities and App Widgets

We will not directly evaluate activities and app widgets because they're only related with UI, it's hard to produce convincing result because there're too many elements out of control, like the GPU power, the screen resolution.

- Services

We will not directly evaluate services as well. The reason is same as activities.

- Content Providers

We will evaluate content providers related **Garbage Collector**, **Synchronization**, and **Scheduler**. Because it's shared data mechanism Android provided.

- Processes and Threads

We will evaluate processes and threads directly. They're both our targets and mechanisms to use.

3 Intents/Receivers

3.1 Single Process

To directly evaluate intents and receivers on single process doesn't make sense. But this can provide **baseline for our experiments**. We can compare the results of same experiment on single process and multiple processes. Then we can get reasoning of different behaviour.

3.2 Multiple Processes

We can use process(es) to generate bunch of intents, then use other process(es) to receive the intents. In addition, there're always background process(es) with some computation to produce pressure for Android system. So we can evaluate the order and time of intent delivery. It can provide some pressure for **Garbage Collector**, **Synchronization**, and **Scheduler**.

3.2.1 Garbage Collector

We can use one sender and one receiver to test Garbage Collector, So we can 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

So where pressure come from? To evalute the behavior of Android system, it need some pressure for different components. So that we can infer the predictability of different components, and the interaction between different components. The pressure can come from:

- Other background process(es) with computation
- Computation inside senders
- Computation inside receivers

We will divide our experiments into three phases:

Phase 1. we only have pressure from background process(es). It's easier to implement and tune for different memory pressure types.

Phase 2. we'll add additional computation to senders and receivers to compare whether computation source affect Android's performance.

Phase 3. we combined the different pressure together to get final evaluation.

The computation can be the task from SPECjvm2008 (Java Virtual Machine Benchmark) or DaCapo Benchmark. In addition, we can assoicate the Parcelable vs. Serializable with experiment phase 3.

3.2.2 Synchronization

This task need other Android components. Because we can not pass an object as extra of an intent, we need serialize the object first. So there no directly synchronized mechanism between sender and receiver, but we can pass some metadata to let receivers use something need synchronization like **Content Provider**. We'll discuss in next section.

3.2.3 Scheduler

We can use multiple background processes to provide pressure for scheduler. Then we use the order of intent delivery to evaluate scheduler and intent delivery mechanism. More details can be found in Section Processes/Threads.

4 Content Providers

Content provider is an Android system's mechanism to manage access to a central repository of data. But Android system does not synchronize access to the Content Provider. So we must implement thread-safe for accessing content providers. One way is using synchronized. We will figure out whether there exist other synchronization mechanism.

But it provides use cases to test synchronization in Android. 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 access the content provider for a fixed times with a fixed interval, then record the finish time. We can gradually increase the number of processes to compare the results.

So our experiments should have such configurable elements:

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

Apparently, we can evaluate the scheduler in the meantime. For example, 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 and obtain comprehensive data on 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 is some Android specific behaviour related to processes and threads. 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 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 seeing 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. Don't assign too much work to the UI thread and don't 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. From this data, it appears that the benchmarks `avrora9`, `hsqldb6`, `lusearch6`, and `eclipse9` 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 performance tests related to the Eclipse IDE

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.

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 something in Android:

- AlarmManager
- Handler

Don't miss deadline for certain task is a critical factor for real-time system. So we'll evaluate the two approaches respectively. But because the functionality is same. So we call them scheduled tasks. To evaluate how well Android handle scheduled tasks. We can create lots of threads/processes with light computation run simultaneously, then to evaluate how many times they miss deadline.

Based on the two hypotheses:

- 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 evaluate how workload affect scheduler

7 Parcelable/Serializable

According to this blog, parcelable mechanism have 10 times better performance than serializable mechanism. But parcelable need 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 for Android system. In the meantime, we can evaluate the performance of them. The result may improve static code analysis of Andorid codes.