Submitted for Publication - June 18, 2010

Reliable Real-Time Applications on Android OS

Bhupinder S. Mongia
Electrical and Computer Engineering
Georgia Tech
Atlanta, GA 30332
bhupinder.mongia@gmail.com

Vijay K. Madisetti, *Fellow, IEEE*
Electrical and Computer Engineering
Georgia Tech
Atlanta, GA 30332
vkm@gatech.edu

***Abstract* – The Android operating system (OS) is widely used within several types of embedded & mobile platforms, including mobile phones and tablets, and the industry is exploring the ability of Android within other embedded platforms, i.e., automotive or military, that require real-time guarantees and the ability to meet deadlines as a pre-requisite for reliable operation. In this paper, we present preliminary conclusions on Android's real-time behavior based on experimental measurements performed on a commercially available Android platform.**

***Index Terms* – Android OS, Realtime Software, OMAP**

I. Introduction

Traditional studies on the reliability of software focus on functional failures, and do not emphasize the time-related behavior of systems that can also cause the software to fail. The ability to meet deadlines and time constraints is critical to embedded systems software (as in automotive or robotic applications) that mandate response to stimuli within pre-specified real-time design specifications, and reliability considerations require a detailed evaluation of the ability of the system to meet these specifications [1-3]. The Android OS is an operating system primarily designed for mobile platforms by Google. It is an open source OS based on LINUX kernel (version 2.6) that enables developers to write applications primarily in Java with support for C/C++ as well [4]. Android is finding widespread acceptance in the mobile and portable computing market, and this study examines, for the first time, its performance & reliability in more demanding embedded real-time applications.

A. *Android Architecture*

An Android system is a stack of software components. At the bottom of the stack is Linux (kernel version 2.6). This provides basic system functionality like process and memory management and security. Also, the kernel handles all the things such as network interface and a vast array of device drivers, which make it easy to interface to peripheral hardware. On top of Linux is a set of libraries, including bionic (the Google libc), media support for audio and video, graphics (OpenGL ES), support for browsers (Webkit), and a lightweight database, SQLite [4].

A key component of an Android system is the runtime engine – the Dalvik Virtual Machine (VM). It was designed specifically for Android and is optimized in two ways. It is designed to be instantiated multiple times – each application has its own private copy running in a Linux process. The Dalvik VM makes full use of Linux for memory management and multi-threading, which is intrinsic in the Java language. The Application Framework provides many higher-level services to applications in the form of Java classes. This will vary in its facilities from one implementation to another.

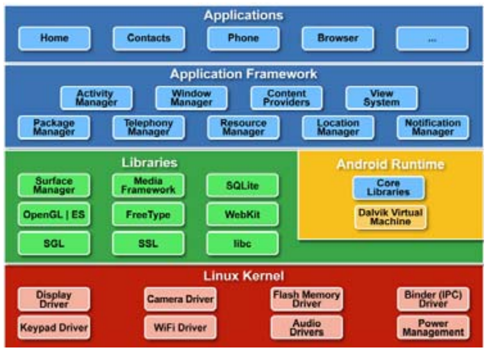

Fig. 1: Android OS Software Architecture [4]

B. *Android OS in Real-Time Embedded Applications*

We use the automotive application as an example of the type of reliable embedded software applications that are being investigated in the context of the use of Android. In the typical automotive application, there are different services (*Control Class*: drive control, braking; *Safety Class*: seatbelts, airbags; *Infotainment Class*: multimedia, climate control, communication services, etc.), that usually provide their own user interfaces. This might overwhelm and distract the typical driver restricting the user from exploiting the full capabilities of these devices. With all these features bundled together on a single platform, the unpredictability in response time of these simultaneously executing and interacting applications may cause the software to fail, resulting in unreliable operation. For instance, if the driver were using his GPS navigation while driving, and a higher priority phone call is received causing the GPS application to be de-scheduled for a long time, the GPS application might miss out on updating the driver on some turn that he should have taken, or if the time to respond to a phone call were too long, the call would be missed. Additional safety considerations come into play if the navigation system or the braking systems were also controlled by the Android OS, in the near future.

C. *Experimental Setup*

We have chosen Texas Instruments' "Zoom II Mobile Development Kit", featuring TI's OMAP 3430 processor as the experimental platform [5]. The OMAP 3430 has an ARM core, which is the most popular core for low power, hand-held general purpose micro-controllers.

Texas Instruments' OMAP Zoom II Platform

The source code for Android including its kernel can be obtained from a repository available at:

- git://git.omapzoom.org/platform/omapmanifest.git

A detailed guide on how to build and install Android on Zoom's OMAP platforms is available at

http://omappedia.org/wiki/Android_Getting_Started

D. Experiment and the Test Procedure

The real-time responsiveness or latency measurement on Android is broken down in two parts. The first part is the latency introduced in handling of an interrupt within the Linux kernel i.e., the time it takes for the linux kernel, after receiving an interrupt (timer interrupt in our experiment), to propagate this event to the event management layer in the kernel. The second part is the latency introduced by Dalvik VM, i.e., the time difference between when it receives the event from the kernel event management layer and passes it up to the Application running on top of the VM.

Another factor that has to be taken into account for deciding whether the system is reliable for real-time use, in addition to the latency incurred in handling of external events, is the "variation" in this latency i.e., for a system to be deemed reliable for real-time application, there has to be an upper bound on how much variation in the latency can be tolerated by the real-time application. We have analysed our experimental measurements with these criteria in mind.

Instrumentation of Android System:

This experiment involves system level latency measurements i.e., the delays introduced by the Linux kernel and the Dalvik virtual machine combined, in propagating the event (timer) up to the Java application running on top of Android. To achieve this purpose, two separate applications, ***Test*** and **Loading**, were developed on the Android Zoom II MDK. The **Test** application's task was to schedule a ***TimerTask*** which would run after the expiry of fixed timer interval (i.e., 10ms or 1ms in this experiment) and observe the error (*slippage* in deadline) by noting the time difference between when the task was scheduled from the Java application, and when the actual timer events were received. The application stores this slippage values onto a file which was read back for later analysis.

The **Loading** application's task was to exercise the CPU and other I/O resources on the system so that measurements could be taken under varying loads.

The **Loading** application schedules another ***TimerTask*** with a varying timer interval (10 ms for normal load and 1 ms for heavy load). In the timer event handler function, various dummy floating points operations are performed, some System APIs are called (for reading the time values), simulating an Android system under load, and the some values are written back onto a file thereby exercising all aspects of the system.

The **Test** application uses Java's High Precision Timer APIs to read in the current time, which gives the timer accuracy in nano seconds. The test application itself may be scheduled at 100ms or 1ms intervals.

The Test application was tested in three scenarios: (1) *Test* application alone, or "**no load**" (2) *Test* application with *Loading* application with 10ms timer scheduler interval simulating "**normal load**" (3) *Test* application with *Loading* application with 1ms timer scheduler interval, simulating "**heavy load**". The Test application itself was run with a)100 ms timer interval, and, b)1 ms timer interval.

II. Experimental Results

In all the observations (plots) shown below, the X-axis shows the reading number (timer event number) and Y-axis shows the corresponding slippage in nanoseconds (ns).

A. Experiments with a 100 ms Test Application Timer

In the experiments with the 100ms timer, the Test application was executed under no load, normal load and heavy load conditions. For the plots presented here, the x-axis represents the timer event number, while the y-axis, in units of ns, represents the error or "slippage". A larger slippage indicates a larger latency in responding to an interrupt, so a real-time application would be deemed less reliable in meeting its deadlines.

1. Behavior under No Load : The Test application was run alone on the Zoom II platform and the timer latency observations were plotted from the readings recorded by the application as shown below.

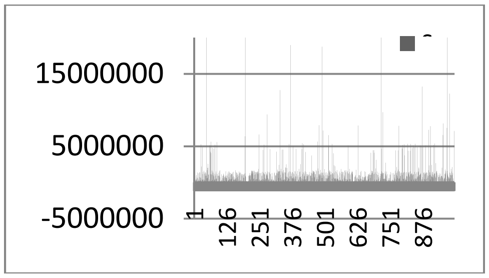

Fig. 2: Test Application under No Load.

In the no load condition, the OS background task tries to program the system in Low Power Mode to conserve energy, therefore, when an interrupt occurs, the entire system has to come back up and manage that event. Although there are some instances of higher delays (60 – 100 ms), much of the error is below 10 ms range.

2. Behavior under Normal Load: The Test application was run along with the Loading application with 10ms timer scheduled period on Zoom II platform and the timer latency observations were plotted from the readings recorded by the application as shown.

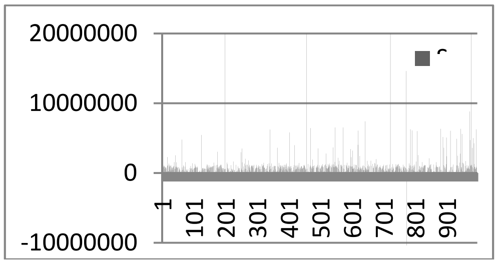

Fig. 3: Test Application under Normal Load.

Under normal system load, the system behaves ideally for hosting real time applications, as it may be seen from the plot that there are *very few occurrences of deadline slippage*, and most of them are contained within nearly 5ms interval. The reason for that could be attributed to the fact that; (i) System does not go into sleep mode as there are processes using the system resources for carrying out certain tasks and keeping the system up; and (ii) The system resources are not under severe contention, where one process using a resource may block up another process or cause an event to be noticed after a delay. Most of the slippage errors here are contained within the 5ms window, except for some very rare cases of high error, e.g., higher than 100ms, implying that the events might have been dropped altogether, leading to failures.

3. Behavior under Heavy Load: The Test application was run along with the Loading application with 1ms timer scheduled period to exert a heavy processing and I/O load on the system and the timer latency observations were plotted from the readings recorded by the application as shown.

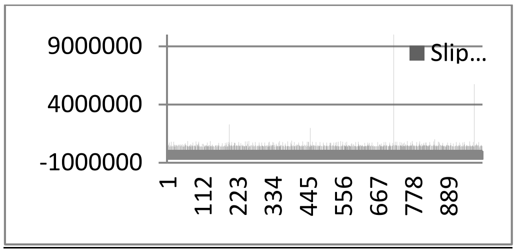

Fig. 4: Test application under Heavy Load

As it can be seen from the plot, there is a small but constant slippage of deadline in invocation of the application level timer event handler. Most of the error is contained within 1ms of the actual event time. This again can be attributed to the fact that system does not go into sleep mode. Furthermore, th high contention for system resources because of the Loading task running with a 1ms timed schedule in which it exercises the system for CPU and I/O usage, task scheduler has to de-schedule the currently running task almost every time to invoke the Test application’s timer event handler, *there is almost always a small but persistent slippage* in the actual time at which the handler is invoked.

B. *Experiments with a 1 ms Test Application Timer*

The same experiment was repeated with the Test application event generation being programmed to occur after every 1 ms. Also, the application was modified to store all the observations in a buffer and write the entire buffer onto a file after the experiment was finished. This was done so that application's “file write” might not block it from receiving the timer event in time, as that would amount to application's incapability to receive the event at the right time rather than delay being caused by the OS in propagating the event to the application.

1. Behavior under No Load : The Test application was run alone on the Zoom II Android Platform and the timer latency observations were plotted from the readings recorded by the application as shown below.

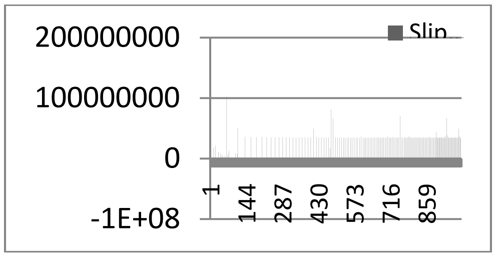

Fig. 5 – Test Application under Light Load, showing slippage propagation and accumulation

As it can be seen here, the deadline miss in this case is around 35ms for the cases when there is considerable deadline miss. It is also evident from the graph that the deadline slippage becomes more apparent during the later stages of the experiment as the previous delays keep *accumulating,* i.e, for every event that is received after some delay, the events after that also show up that delay and add to it – we call this *slippage propagation and accumulation.*

2. Behavior under Normal Load: The Test application was run along with the Loading application with 10ms timer scheduled interval and the timer latency observations were plotted from the readings recorded by the application as shown.

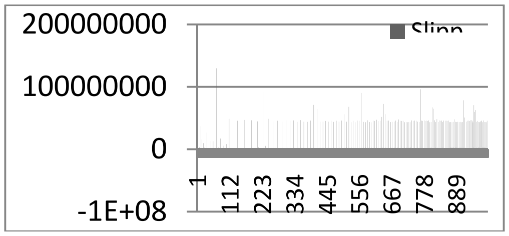

Fig. 6 – Application under Normal Load, with increasing slippage accumulation.

The above plot is consistent with the plot obtained for latency under no load except for the fact that deadline slippage in this case is almost always over 42 ms for the cases where there is a considerable deadline miss.

3. Behavior under Heavy Load: The Test application was run along with the Loading application with 1ms timer scheduled period to exert a heavy processing and I/O load on the system and the timer latency observations were plotted from the readings recorded by the application as shown.

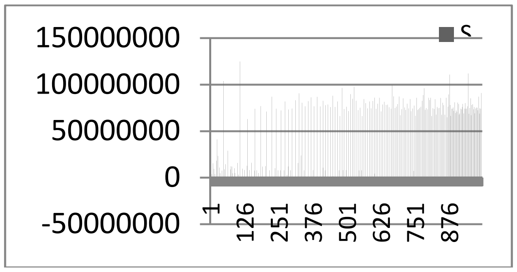

Fig. 7 – Application under Heavy Load – Increasing Slippage and Slippage Accumulation.

Consistent with our analysis, when the test application itself is putting a heavy load on the system, i.e., it is very demanding in terms of invocation of frequently occuring events, much of the CPU time is spend in scheduling the test application and hence, if there is some other process, which is also heavily loading the system resources running in parallel with the test application, the system is not able cope with the high frequency of schduling required by the test application.

III. Analysis of Experiments

We are now in a postion to make an analysis of the experimental behavior of Android under varying scenarios (deadlines, load) as described in Section II.

A. Frequency of slippages increase with increasing load

In all cases, increasing the load caused the frequency of slippage to increase. For instance, in Fig. 3, the slippage times occurred less frequently than in Fig. 4 (under Heavy Load).

B. Increasing frequency of interrupts increased slippage times

As shown in Fig. 4 & 7, when interrupts are infrequent (Fig. 4), the slippage times were small (i.e., 1ms). However, when the frequency of interrupts was increased, as in Fig. 7, the slippage times increased to around 80ms.

C. Increasing frequency of interrupts causes slippage accumulation

As noticed in Fig. 6 & 7, a frequent interrupt caused Android keep accumulating missed deadlines, and fall further back, with more frequent misses in meeting deadlines, which is exacerbated by a heavy load.

IV. Conclusion

Our experimental results and analyses show that deadline misses of between 1 and 5 ms are common when the frequency of interrupts is small (e.g., 10Hz). However, when the frequency of interrupts is increased (e.g., 1Khz), deadline misses or response times in the order of 0.5sec are observable. Furthermore, the frequency of these misses increases with time through the process of slippage accumulation, resulting in potentially a slowdown in the operation of the system. If there are more than a dozen interrupts per second under load, we observe that the Android OS may not demonstrate reliable behavior (e.g., response times increase significantly) with respect to real-time constraints. The addition of a real-time scheduler (e.g., a Rate Monotonic Scheduler) may increase the reliability of Android, and we are experimenting further along these lines of research.

While Android OS supports pre-emption and multi-tasking, our results indicate that designers of real-time applications that propose to use Android OS should conduct measurements of its behavior carefully to gauge the combined effects of slippage, its frequency and value, and its accumulation, on the reliability of their system.

References

[1] M. R. Lyu, Handbook of Software Reliability Engineering, McGraw Hill Publishing, 1995, ISBN 0-07-039400-8.
[2] J. Musa, "Operational Profiles in Software-Reliability Engineering, IEEE Software, March 1993.
[3] G. Vo et al, "Building Automotive Software Component with the AutoSAR Environment – A Case Study," Proc. 9th International Conference on Quality Software, Jeju, Korea, Aug 24-Aug 25, 2009.
[4] Google Android SDK, http://developer.android.com/sdk/index.html
[5] Texas Instruments' OMAP Zoom II: http://omapzoom.org.