Android and Real-Time Applications: Take Care!

Article · May 2013

CITATIONS
5

READS
4,117

3 authors:

Luc Perneel
Vrije Universiteit Brussel
20 PUBLICATIONS **164** CITATIONS
SEE PROFILE

Hasan Fayyad
Vrije Universiteit Brussel
15 PUBLICATIONS **156** CITATIONS
SEE PROFILE

Martin Timmerman
Dedicated Systems Experts NV
42 PUBLICATIONS **295** CITATIONS
SEE PROFILE

Vol. 4.Special Issue ICCSII ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

http://www.cisjournal.org

Android and Real-Time Applications: Take Care!

[1]Luc Perneel, [2]Hasan Fayyad-Kazan, [3]Martin Timmerman

[1]PhD Candidate, Department of Electronics and Informatics, Vrije Universiteit Brussel, Pleinlaan 2- 1050 Brussel, Belgium

[2]PhD Candidate, Department of Electronics and Informatics, Vrije Universiteit Brussel, Pleinlaan 2- 1050 Brussel, Belgium

[3]Professor, Department of Electronics and Informatics, Vrije Universiteit Brussel, Pleinlaan 2- 1050 Brussel, Belgium

[3]Professor, Department of Mathematics, Royal Military Academy Brussels, Hobbemastraat 8-1000 Brussels, Belgium

E-mail: [1]luc.perneel@vub.ac.be, [2]hafayyad@vub.ac.be, [3]martin.timmerman@vub.ac.be

ABSTRACT

Android is thought as being yet another operating system! In reality, it is a software platform rather than just an OS; in practical terms, it is an application framework on top of Linux, which 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 performance, boot-up time, real-time response, reliability, and no hidden maintenance costs.

In this paper, we present a preliminary conclusion about Android's real-time behavior and performance based on experimental measurements such as thread switch latency, interrupt latency, sustained interrupt frequency, and finally the behavior of mutex and semaphore. All these measurements were done on the same ARM platform (Beagleboard-XM). Our testing results showed that Android in its current state cannot be qualified to be used in real-time environments. Finally we provide some potential solutions for using Android in such environments.

Keywords: *Real-time, Android*

1. INTRODUCTION

Android [1] is considered today as one of the leading platforms for mobile devices. Being as an open source platform distinguishes it from most other mobile platforms such iOS, Blackberry and Windows Phone.

As Dan Morrill memorably explained in "On Android Compatibility", "*Android is not a specification or a distribution in the traditional Linux sense. It's not a collection of replaceable components. It is a chunk of software that you port to a device.*"[2]. Android is an open source platform built by Google that includes an operating system, middleware, and applications for mobile platforms. It is based on Linux kernel that enables developers to write applications primarily in Java with support for C/C++ as well.

A key to its likely success is licensing. It is open source and a majority of the source is licensed under Apache2, allowing third party adopters to do interesting developments and useful modifications of the platform.

Since its official release, this software platform has been constantly improved either in terms of features or supported hardware, and at the same time, extended to new types of devices different from the originally intended mobile ones [3].

Recently, efforts were undertaken by researchers to enhance the real-time capabilities of the Android platform [3] [16] [18] in order to be employed in a variety of other embedded systems.

In different embedded software systems (as in automotive or robotic applications), the ability to meet deadlines and time constraints is a critical part of the design specifications. These systems require response to stimuli within pre-specified real-time constraints. As such, the reliability of software has not to focus only on the functional failures but require as well a detailed evaluation of the ability of the system to meet these timing specifications [4].

The aim of our research is to test the real time behavior and performance of Android, in order to make it clear whether Android can be advised to be used in open real-time environments. For this evaluation, a testing suite of four performance tests and one behavior test are used. The performance tests are: thread switch latency, interrupt latency, sustained interrupt frequency, and semaphore acquire-release timing in contention case, while the behavior test checks the mutex locking behavior.

2. ANDROID ARCHITECTURE

An Android system (Figure 1 [5]) is an eco-system made up of a stack of software components. At the bottom of the stack, there is the Linux OS which provides basic system functionality such as process and memory management, security, and networking. It supports also a vast amount of

device drivers which are there for handling network interfaces, file systems, human machine interfaces and more.

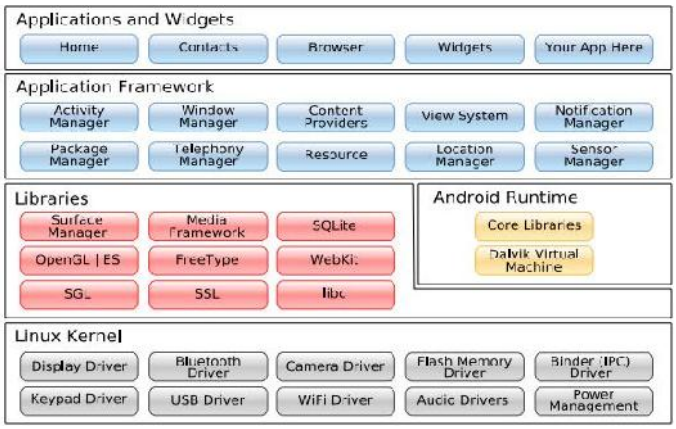

Figure 1: Android Architecture [5]

The Linux kernel used for supporting the Android system has been significantly modified by Google. For instance, a specific inter-process communication system (IPC), a kernel logging facilities, a low memory killer, a shared memory system and many other changes were developed. Therefore, running Android on top of the standard Vanilla Linux is not possible without merging these Android specific changes into the mainline kernel, which can only be done from Linux version 3.3. Before that, it was not easy to build a real-time Linux kernel for Android. Remark that until the moment of writing this paper, the latest official Android Linux version as released by Texas Instruments targeting the Beagle platform is still based on the Linux kernel v3.2 and Android v4.0.3 (Ice Cream Sandwich).

On top of the Linux OS, there is a set of libraries including Google's version of *libc* called *Bionic*, along with media and graphics (OpenGL ES) libraries, browsers (Webkit) support, and a lightweight database, SQLite. [5]

Alongside the libraries, on top of the OS, is the Android runtime "Dalvik Virtual Machine" (DVM) which is a key component of an Android system. It was designed specifically for Android and it is completely different compared to the original (Sun) Java Virtual Machine (JVM). Dalvik uses register based byte code instead of the stack based JVM as originally implemented by Sun and this to conserve memory and maximize performance. It is designed to be instantiated multiple times, where each application has its own private copy running in a Linux process. DVM makes full use of Linux for memory management and multi-threading, which is needed to support the Java language.

Above the libraries and Android-runtime layer, is the Application Framework layer which provides many higher-level services to applications in the form of Java classes. The use of it will vary from one implementation to another. [5]

The top layer is the application layer which contains a number of applications that are routinely distributed with Android, which may include email, SMS, calendar, contacts, and Web browser.

One of the reasons that pushed Google to develop a completely new JVM is to be totally independent from the original JVM developed by Sun. They started the Dalvik development from scratch. This "clean room" approach created a totally different Java VM implementation using a register based byte code instead of the original stack based one. Although a clean room approach doesn't protect against patent infringement, it reduces the risk significantly.
The Java language itself is not protected by patents, however the Sun JVM implementation is. Consequently, all existing real-time enhancements developed for the original JVM cannot be applied nor ported easily to Dalvik.
The Dalvik VM uses *bionic libc* instead of standard Linux *libc* used by Sun embedded JVM. *Bionic libc* is not compatible with *glibc*. All native libraries must be compiled against *bionic libc*. It has a very fast and small custom *pthread* implementation compared to *glibc*. [6].

Bionic has built-in support for Android specific services like system properties and logging capabilities. Writable data segments will be loaded in each process. Doing so, they made the size of these segments as small as possible. Also the code size is kept small. Linux will load the read-only pages only once in memory for each process using it. This approach is adapted to small footprint devices.

As a synthesis: the reasons to develop *Bionic* instead of using the *glibc* library are: a small memory footprint, high speed on CPUs at relatively low frequencies, and last but not least to avoid the inclusion of GPL code at user space level in its platform, where BSD is used instead.

What is now the impact of using *Bionic libc* if it comes to real-time applications?
Bionic libc does not handle C++ exceptions. Omitting lower level exceptions does not cause any problems to Android because its primary language is JAVA which handles exceptions in its internal runtime package [6] but this is not useful in a real-time-context.
We also observe that *Bionic* is missing a very essential object one can find in the traditional *glibc* implementation: priority inheritance mutexes. Although they are available in the kernel, one can access them only by building own library above the Linux system calls. The lack of some priority inversion avoidance mechanism is already enough to disqualify the system as real-time capable.

Remark that there are some alternative library providers showing up like CrystaX [7]; it uses the *newlib* variant, which

also does not support priority inheritance for mutexes yet. As a consequence, this is not a solution either.

3. EXPERIMENTAL SETUP

Linaro [8] Android Beagle release 11.09 is tested here. This image release uses Android version 2.3.5 and is based on Linux 3.0.4. At the time of the evaluation tests, this image was the only available to run on the chosen ARM platform (BeagleBoard-XM).
It is already known that Android has gone through radical changes and had much more recent releases (currently versions 4.x). However, our test results remain valid on these recent releases because the important parts that are influencing the real-time behavior of Android have not been modified. We go in more detail of the justification later on.
The used source code for Android including its kernel is obtained from a repository available at [9] and [10]. Our test policy is to use the kernel configuration as delivered and no adaptation is done.

The main hardware platform for Android is the ARM architecture. There is support for x86 from the Android x86 project [11] and Google TV uses a special x86 version of Android.
BeagleBoard-XM Rev C hardware [12] is used as our experimental platform. Its characteristics are: based on the Texas Instruments DM3730 Digital Media Processor, ARM Cortex A8 processor running at 1GHz, L1 instruction and data caches are 32KB each, L2 cache is 64KB, and 512MB RAM at 166MHz.

4. TESTING PROCEDURES AND RESULTS

A. Measuring Process

For tracing and measuring the performance results on the chosen hardware, an internal General Purpose timer (GPT) running at 13MHz is used. Reading out the values of this timer takes some overhead; however, there isn't any jitter at all in the overhead and therefore generates a constant bias which can be corrected. Although it is possible to let the GP timer run at a higher frequency, the clock attached to this GP timer is also distributed to other components on the chip. Therefore, we stick with the configuration that is used by the OS on this board and we don't need the extra resolution in our context.

In order to directly access the GP Timer from our test software, it is mapped into user space. Further, the measurements' samples are stored in RAM and are written on non-volatile storage after the completion of the test which is then transferred to a data analyzing software. We use the mlockall() call to prevent copy-on-writes upon first write in a virtual page. Also, the real-time run-away protection is disabled by setting the kernel configuration parameter (/proc/sys/kernel/sched_rt_runtime_us) to (-1).

The application and libraries are stored on RAM disk to avoid swapping out any read-only section. Finally, all tests are done using threads in the real-time scheduling classes.

B. Performance metrics

A quick online survey of RTOS metrics maintained by third party consultants, students, researchers, and official records (of distributor companies) reveals that the following three categories are used to evaluate an RTOS solution [13]:

- Memory footprint
- Latency
- Services performance

Among these measurements, footprint provides an estimated usage of memory by a RTOS on an embedded platform. The other two categories measure various types of RTOS overhead or runtime performance. Latency is reported in two different ways: interrupt and scheduling, and services performance is the minimum time taken by the RTOS interface to complete a system call [13].
In this paper, we test latency and services performance metrics. Before presenting the evaluation tests and the obtained results, we always perform two preliminary tests (the first 2 tests below) to assure the accuracy and precision of the tests.

1) Tracing overhead

This test calibrates the tracing system overhead which is essentially hardware related. The results presented in this paper are corrected from this constant bias.
Tracing precision depends on the GPT (13MHz), as this is the minimum time frame that can be detected. As a consequence, the results in this paper are correct to +/- 0.1 µseconds and are therefore rounded to 0.1 µseconds. The Y-Axis's in the charts are all in µseconds.

2) Clock tick processing duration

This test examines the clock tick processing duration in the kernel. The results are extremely important as the clock interrupt - being on a high level interrupt on the used hardware platform - will disturb all other performed measurements. Using a tickless kernel will not prevent this from happening as it will only lower the number of occurrences. The tested OS kernel is not using the tickless timer option.
The way we get the clock tick duration in this test is simple: we create a real-time thread with the highest priority. This thread does a finite loop of the following tasks: get time, start a busy loop that does some calculations, get time again. Having the time before and after the busy loop provides the time needed to finish the job. This busy loop will only be delayed by interrupt handlers. As we remove all other interrupt sources, only the clock tick timer interrupt can delay the busy

loop. When the busy loop is interrupted, its execution time increases.

Figure 2a shows the results over time of the "clock tick duration test". The X-axis indicates the time when a measurement sample is taken with reference to the start of the test. The Y-axis indicates the duration of the measured event; in this case the total duration of the busy loop. This description applies to all the figures below.

The bottom line of figure 2a represents the normal busy loop duration if no clock interrupt occurs during the busy loop. Due to the high values that we have in this figure, the busy loop duration is not that clear because of scaling reasons. Figure 2b is a zoomed version which can clearly show that the busy loop time, if no interrupts occurred, is 42µs. The other samples above the bottom line in figure 2a are measurements when a clock interrupt occurred during the busy loop. The difference between these points and the bottom line is in fact the clock tick processing duration.

A worst clock time duration of about 350 µs is detected (Figure 2a), which is extremely high. Clearly, each 100ms, the timer causes a major delay. Further, having a detailed look at the clock ticking (Figure 2b) (skipping the long 350µs delay), we see that it is split up into three parts. This seems to be a phenomenon in the kernel when using the low power 32 KHz OMAP timer to control the operating systems clock tick. By enabling a higher frequency GP timer as base for the operating system clock, the traditional clock tick behavior will be visible again.

We did our tests using the default configuration from Linaro, which uses the low power 32 KHz OMAP timer as OS system clock source and sets the operating system tick frequency of 128Hz. Although our policy is to do black box testing only, it shows that the tests are very efficient in uncovering specific behavior which could help system designers to fine-tune or enhance their system.

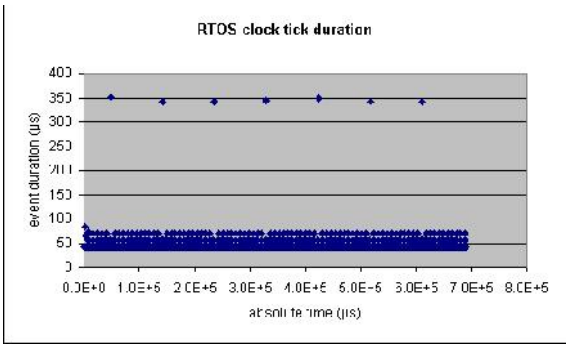

Figure 2a: Clock tick duration

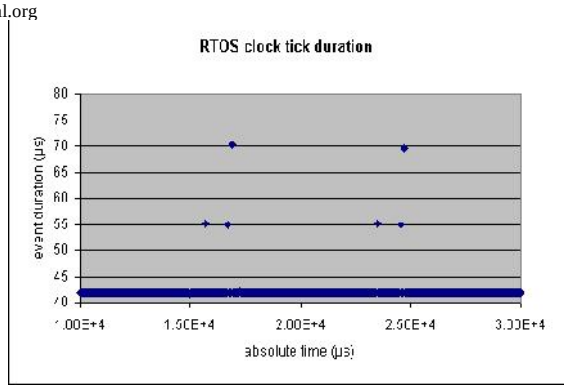

Figure 2b: Clock tick duration (zoomed view)

Remark that in the evaluation of the Linux PREEMPT_RT on the same platform (can be found in [14]) which was particularly fine-tuned for real-time use, a high frequency timer is used to control the operating system clock. Android however is designed for portable devices and is optimized for low power consumption. This is a bad idea when real-time performance is needed.

3) Thread switch latency between threads of same priority

Although real-time threads should be on different priority levels to be capable of applying rate monotonic scheduling theory [20], this test is executed with threads on the same priority level in order to easily measure thread switch latency without interference of something else.

For this test, threads must voluntarily yield the processor for other threads; so SCHED_FIFO scheduling policy is used. If we wouldn't use this policy, a round-robin clock event could occur between the yield and the trace, so that the thread activation is not seen in the trace.

Here is a brief explanation of this test: A "*creating*" thread starts creating a specific number of threads that have the same priority level which is higher than its priority. Whenever a thread is created, it will immediately lower its priority below the priority level of the *creating* thread in order for the "*creating*" thread to continue creating all the desired threads. Once all the threads are created, the *creating* thread lowers its priority below the priority level of the created threads. The first thread in the queue will start execution, does its job, and then yield the processor for the next thread (which does the same). The job of each created thread is to get the timer counter value at the beginning of its execution, do some calculations, and then again get the timer count value at the end of its execution. The difference between the ending timer count of the previous thread and the start value of the next thread is the switch latency.

This test looks for the worst-case behavior, and therefore it is done with an increasing number of threads, starting with two (2) and going up to 1000. As we increase the number of active threads, the caching effect becomes visible since the thread context will no longer be able to reside fully in the cache (on this platform the L1 caches are 32KB, both for

the data as the instruction cache). Further, you will clearly see the influence of clock interrupts (causing the maximum values in the figures below).

In the 1000 threads test, you will observe we were lucky not to catch the 300µs delay during the thread switch measurement. We publish this sub-test on purpose to show the random behavior of OS based system. Our tests are repeated multiple times (or bigger trace buffers are used if possible) in order to have a high degree of probability that all (bad) behavior has been caught.

The testing results (in µs) of this test are shown as follows:

Test	Avg	Max	Min
Thread switch latency between ***2 threads***	7.9	317	7.6
Thread switch latency between ***10 threads***	8.4	321	7.9
Thread switch latency between ***128 threads***	13.2	363	9.5
Thread switch latency between ***1000 threads***	14.3	62.8	12.8

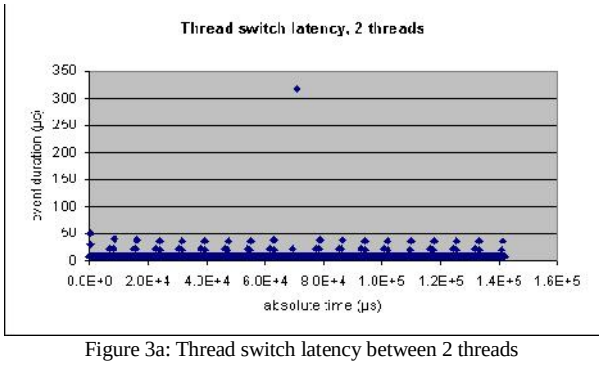

Figure 3a: Thread switch latency between 2 threads

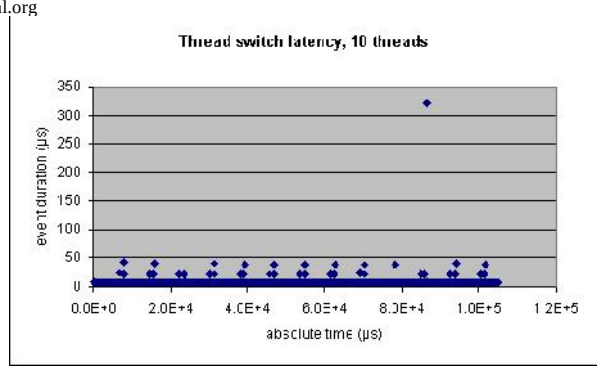

Figure 3b: Thread switch latency between 10 threads

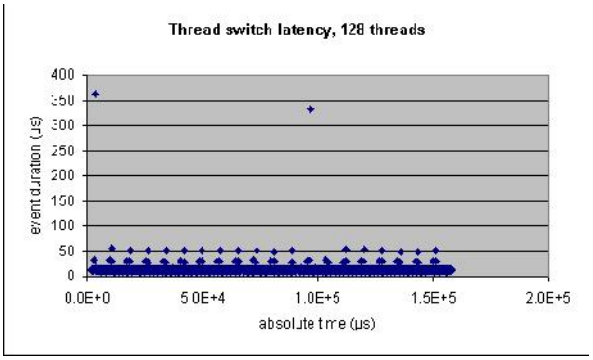

Figure 3c: Thread switch latency between 128threads

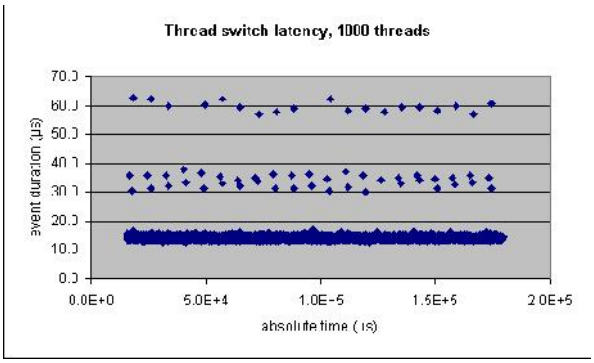

Figure 3d: Thread switch latency between 1000 threads

4) Interrupt latency

This test measures the time required to switch from a running thread to an interrupt handler. The time is measured from the moment the running thread stops executing up to the first instruction in the interrupt handler itself. So it does not measure the hardware interrupt latency, but only the software part. This measurement method will not detect how long the interrupt has been masked. The sustained interrupt test (next test metric 5) will do.

Note that in this test, another GPT on the board is used for generating the interrupts.

The clock time is easily detected again.

The results of this test are shown in Figure 4 and the following table.

Test	Avg	Max	Min
Interrupt dispatch latency	1.9 µs	30.9 µs	1.8 µs

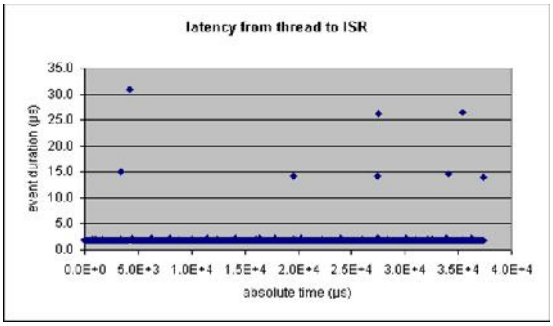

Figure 4: Latency from the interrupted thread to the interrupt handler

5) Maximum sustained interrupt frequency

This test detects when an interrupt cannot be handled anymore due to the interrupt overload. In other words, it shows a system limit depending on, for example, how long interrupts are masked, how long higher priority interrupts (the clock tick or other) take, and how well the interrupt handling is designed.

This test gives a very optimistic worst case value due to the fact that, because of the high interrupt rate, the amount of spare CPU cycles between the interrupts is limited or nil. Also, depending on the length of the interrupt handler, it might mostly be present in the caches. In a real world environment, the worst case will be greater.

In this test, 10 million interrupts are generated at specific interval rates. Our test suite measures whether the system under test misses any of the generated interrupts. The test is repeated with smaller and smaller intervals until the system under test is no longer capable handling the interrupt load.

The table below shows the results:

Interrupt period	#interrupts generated	#interrupts lost
100 µs	10 000	23
120 µs	10 000	17
150 µs	10 000	2
180 µs	10 000	0
310 µs	100 000	3
330 µs	100 000	2
350 µs	100 000	0
350 µs	1 000 000	4
370 µs	1 000 000	0
370 µs	10 000 000	6
390 µs	10 000 000	3
410 µs	**10 000 000**	**0**

As one can observe in the test results, the clock tick gives us a serious penalty here. On the long run, you can see that the guaranteed interrupt latency is around 410µs. This is much larger than the best case measured with the test metric 4 which was 1.8µs showing again that metric 4 is really optimistic.

6) Semaphore acquire-release timings in the contention case

This test checks the time needed to acquire and release a semaphore, depending on the number of threads pending on the semaphore. In other words, it measures the time in the contention case when the acquisition and release system call causes a rescheduling to occur.

The purpose of this test is to see if the number of pending threads has an impact on the durations needed to acquire and release a semaphore. It attempts to answer the question: "How much time does the OS needs to find out which thread should be scheduled first?" In real-time systems, this should be a constant to keep it predictable.
In this test, as each thread has a different priority, the question is how the OS handles these pending thread priorities on a semaphore.

Test scenario: we have a *"creating"* thread which creates a semaphore with count zero, and then starts to create 90 threads with different priorities. The *creating* thread has the lowest priority. When a thread is created, it starts execution immediately and tries to acquire the semaphore; but as the semaphore count is zero, the thread blocks and the kernel switches back to the *creating* thread. The time from the acquisition attempt (which fails) to the moment the *creating* thread is activated again is called here the "acquisition time". This time includes the thread switch time.

After the last thread is created and pending on the semaphore, the *creating* thread starts to release the semaphore repeating this action until there is no more any thread pending on the semaphore. So this action is repeated 90 times (the number of pending threads on the semaphore). The moment the semaphore is released, the "release duration" time is started. The highest priority thread that is pending on the semaphore becomes active and it will stop the "release duration" time for the current pending thread. We can see in figure 5b that the release duration for the first thread is around 450µs, and this value is decreasing for the following threads.

The "release duration" also includes the thread switch duration.

Now, the most important part of this test is to see if the number of threads pending on a semaphore has an impact on the release time periods. The answer is yes, which is clearly visible in figure 5b as the first thread pending on the semaphore required around 450 µs and the values decrease as the number of pending threads is decreasing. If only one thread is pending, the release time takes an acceptable 15µs which can be seen at the end of the figure. But with 90 pending threads, the release time is 30 times higher (around 450µs).

This means the *Bionic* library implementation have an extremely bad behaving semaphore. First, it is not priority based but FIFO based. Second, its release durations are extremely depending on the number of pending threads in a way that the OS has to look for the highest priority thread in the semaphore queue of pending threads to release it first. It is strange that a FIFO based implementation can still behave so badly upon semaphore release and disqualifies the system as being real-time.

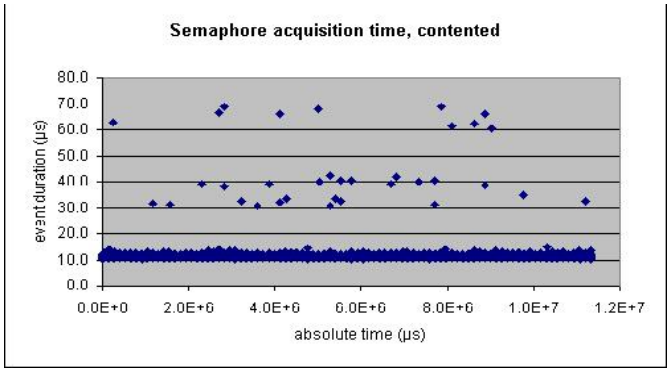

Figure 5a: Semaphore acquisition time-contention case

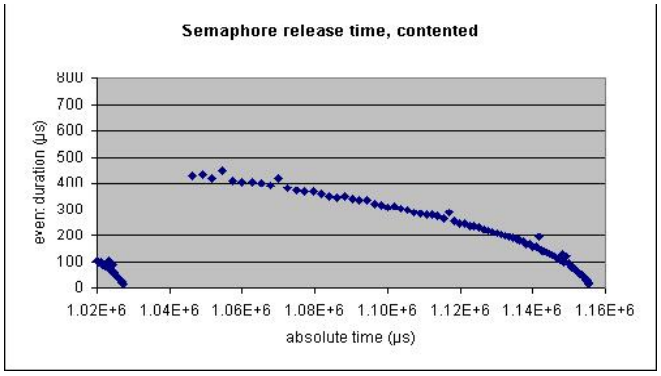

Figure 5b: Semaphore release time-contention case

An advantage of an open source platform is the possibility to have a look into the library code itself and try to understand why this behavior is so bad which deviates us from our black box testing policy. Checking the code of the sem_post() operation in *bionic* reveals the following [15]:

```
int sem_post(sem_t *sem)
{
 …
 old = __sem_inc(&sem->count);
 if (old < 0) {
   /* contention on the semaphore, wake up all waiters */
   __futex_wake_ex(&sem->count, shared, INT_MAX);
 }
 …
}
```

Thus upon a semaphore post operation, ALL pending threads are activated at once. In the sem_wait, all threads, except one, will start wait again:

```
int sem_wait(sem_t *sem)
{
   …
   for (;;) {
     if (__sem_dec(&sem->count) > 0)
     break;
    _futex_wait_ex(&sem->count,shared,
                   shared|SEMCOUNT_MINUS_ONE, NULL);
   }
   …
}
```

Conclusion is that the measured behavior matches the code. In the *glibc* implementation, only one thread is activated upon a normal semaphore post operation which is much better.

7) Mutex Locking behavior

This test checks the behavior of the mutex locking primitive using the `pthread_mutex_lock` and related POSIX calls. Although the Linux kernel supports priority inheritance as a system to avoid priority inversions (a must have in real-time systems), the *bionic* C libraries used in Android do not provide this mutex configuration. As a result, a C application cannot prevent priority inversions using this library.

Our mutex behavior test for verifying this feature is depicted in figure 6a.

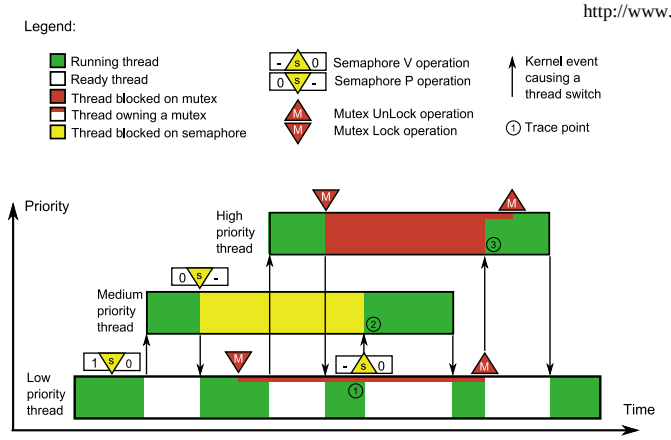

Figure 6a: Mutex locking behavior without priority inheritance

This test creates three threads which are locked to the same processor in order to avoid parallel execution on a multi-processor system. The thread being active at a certain time is shown in green in figure 6a. The low priority thread starts first. It creates a semaphore with count zero. Then the medium priority thread is created and activates immediately. The medium priority thread tries to acquire the semaphore, but as the semaphore count is zero, it blocks on the semaphore (shown in yellow). The low priority thread activates again and continues execution; it creates a mutex for which it takes ownership and then creates a high priority thread, which activates immediately. The high priority thread tries to acquire the mutex owned by the low priority thread, and thus blocks (shown in red). As a result, the low priority thread becomes active again. Now the low priority thread releases the semaphore where after it releases the mutex.

In the non-priority inheritance case (using the *Bionic* C-library), the medium priority thread will start upon the semaphore release and thus keeps the high priority thread blocked for a long time (until it blocks or a higher priority thread becomes runnable). This means, in such case, a lower priority thread delays unintentionally a higher priority thread, which is by definition an unwanted priority inversion. Finally the medium priority thread ends and the low priority thread activates again. At this moment, the low priority thread will release the mutex, which will unblock the high priority thread and activates immediately.

Our test generates the following tracing numbers: (1) before the semaphore release, in the low priority thread context; (2) after the semaphore request in the medium priority thread context; (3) after becoming mutex owner in the high priority thread context. These traces are shown in figure 6a. Without any priority inversion protection mechanism present, the tracing results will have this order: (1), (2) and (3).

Figure 6b shows the theoretical behavior of a system supporting a priority inversion protection mechanism: in this picture priority-inheritance. Some tested OS like Linux-PREEMT_RT, Windows Embedded Compact 7 and QNX show this behavior and can be found in [14].

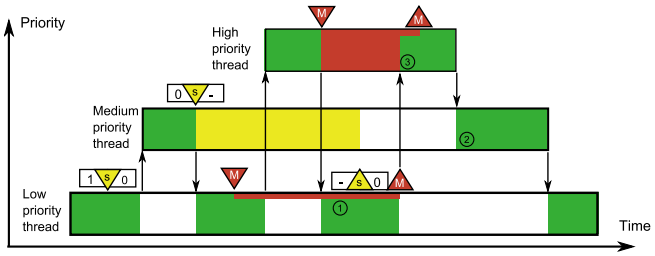

Figure 6b: Mutex locking behavior with priority inheritance

In case the mutex supports priority inheritance, the low priority thread will inherit the priority of the high priority thread when it requests the mutex. Thus the lock owner inherits the (higher) priority of the thread blocked on the lock. The low priority thread will in this case execute as it is a high priority thread. It releases the semaphore first. As a result, the semaphore release will not wake-up the medium priority thread, but the low priority thread will continue. It releases the mutex; at this point, the priority inheritance finishes and the low priority thread goes back to its original priority level (low). The high priority thread will be unblocked. Under such circumstances, the test tracing points are shown in this order (1), (3) and (2).

The lack of mutex priority inheritance support is another reason to disqualify Android as a candidate for real-time systems.

5. PROPOSED SOLUTIONS FOR MAKING REAL-TIME ANDROID

There are different approaches discussed in [3] [16] to incorporate the desired real-time behavior into the Android architecture. The first approach considers the replacement of the Linux operating system along with its Completely Fair Scheduler (CFS) [17], and replaces it with a real-time version of Linux. Remark that this was not a trivial task for Linux kernels before version 3.3 as the android kernel patches were not integrated in the mainline kernel yet. Starting from Linux v3.3, most important Android patches are now part of the vanilla kernel.

The second approach respects the Android standard architecture by proposing the extension of Dalvik as well as the substitution of the standard operating system by a real-time Linux-based operating system. [3][16]

The third approach simply replaces the Linux operating system by a Linux real-time version. This is however not enough: real-time threads should use the kernel directly or use *glibc/uClibc* instead of the *bionic* C library.

Finally, the fourth approach proposes the addition of a real-time hypervisor that supports the parallel execution of the

Android platform in one partition while the other partition is dedicated to the real-time applications. [3] [16]

When building the real-time applications, there is a need as well for a communication channel between these applications and the Android system. Such a communication channel should not affect the timing behavior of the real-time threads.
Each of these approaches has its advantages and drawbacks as explained in [3] [16]. This means there is not a ready mature solution that can let Android be qualified to be used for real time purposes.

Recently, a research work "A Real-time Extension to the Android Platform" [18] was published, in which the authors use the RT_PREEMPT patch to equip Android's Linux kernel with basic real-time support and improved the Dalvik garbage collection. Although, this is a step in the good direction to enhance Android for real-time applications, our research shows that the major problem is not only about scheduling but also about the *Bionic* libraries' implementation. As long as this library is not replaced or enhanced, priority inversion together with the long delays will result in deadline misses when trying to build an Android based real-time systems.

A solution could exist by combining both the Android stack and the Linux software stack, on one device. A lockless communication system could then be made between them. Such a system is shown in Figure 7 and is part of our future research work.

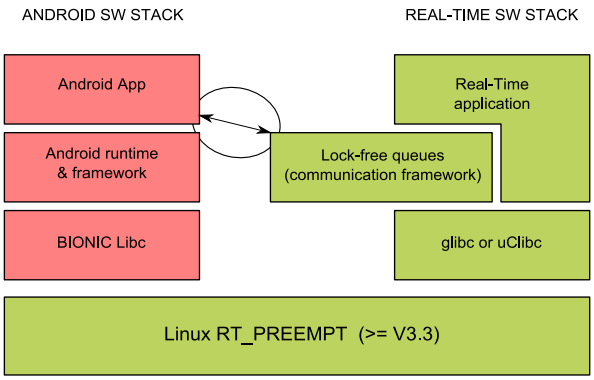

Figure 7: Combining both software stacks on one platform

6. CONCLUSION

Initially, Android was not meant to be used in real-time applications. It is a Java executable platform consuming as less power as possible for handheld devices with an attractive Graphical User Interface. Of course, one cannot avoid that some people try to use it in a real-time context based on its popularity only.

The current research about real-time Android is focusing on the real-time capabilities of the kernel. A real-time OS is however much more than just a scheduling issue. Although you can build an Android device using a PREEMPT_RT kernel [18], other real-time capabilities won't be present. This is because of the *Bionic* C library which does not implement the features and behavior that are mandatory to have a real-time system. For example, *Bionic* does not have the priority inversion protection mechanisms available on mutexes. Also the semaphore implementation behaves badly: it is implemented as a purely FIFO queued one without any prioritization and release times are going straight through the top once multiple threads are blocked on the same semaphore. This behavior is bad even for non-real-time systems.

Although Android has many recent releases, the *Bionic* library is still not enhanced to support at least the priority inheritance feature in the *pthread* header file [19].

Today, the Android platform is inappropriate to fulfil real-time requirements of any kind. An Android subsystem playing the role of a GUI towards another real-time subsystem is the only possible immediate system design scenario.

REFERENCES

[1] Google, "Android," [Online]. Available: www.android.com. [Accessed 2013].

[2] T. Bray, "Ongoing by Tim Bray-What Android Is," November 2010. [Online]. Available: http://www.tbray.org/ongoing/When/201x/2010/11/14/What-Android-Is

[3] C. Maia, L. M. Nogueira and L. M. Pinho, "Evaluating Android OS for Embedded Real-Time Systems," in Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels, Belgium, 2010.

[4] S. M. Bhupinder and K. M. Vijay, "Reliable Real-Time Applications on Android OS," 2010. [Online]. Available: http://users.ece.gatech.edu/~vkm/Android_Real_Time.pdf.

[5] Google, "Android SDK," [Online]. Available: http://developer.android.com/sdk/index.html.

[6] K. Tapas Kumar and P. Kolin, "Android on Mobile Devices: An Energy Perspective," in 2010 IEEE 10th International Conference on Computer and Information Technology (CIT), 2010.

[7] CrystaX, "CrystaX .NET," [Online]. Available: http://www.crystax.net/nl/android/ndk/7.

[8] Linaro, "Linaro: Open source software for ARM SoCs," [Online]. Available: http://www.linaro.org/.

[9] Android, "Linaro Android Build Service," [Online]. Available: https://android-build.linaro.org/.

[10] Android, "android.git.linaro.org Git," [Online]. Available: http://android.git.linaro.org/gitweb.

[11] S. Agam, "Google's Android 4.0 ported to x86 processors," 1 December 2011. [Online]. Available: http://www.computerworld.com/s/article/9222323/Google_s_Android_4.0_ported_to_x86_processors.

[12] BeagleBoard, "BeagleBoard.org-default," [Online]. Available: http://www.beagleboard.org

[13] F. Sheikh and D. Driscoll, "Measuring RTOS performance: What? Why? How?," 2011. [Online]. Available: http://www.eetimes.com/electrical-engineers/education-training/tech-papers/4219481/Measuring-RTOS-Performance-What-Why-How.

[14] D. S. Experts, "RTOS evaluation Reports and related papers," [Online]. Available: http://download.dedicated-systems.com.

[15] Google, "/libc/bionic/semaphore.c," [Online]. Available: https://android.googlesource.com/platform/bionic/+/android-4.2.2_r1/libc/bionic/semaphore.c.

[16] B. Cole, "Real-time Android: real possibility, really really hard to do - or just plain impossible?," 2012. [Online]. Available: http://www.embedded.com/electronics-blogs/cole-bin/4372870/Real-time-Android--real-possibility--really-really-hard-to-do---or-just-plain-impossible--.

[17] IBM, "Inside the Linux 2.6 Completely Fair Scheduler," [Online]. Available: http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/.

[18] I. Kalkov, B. Franke and J. Schommer, "A Real-time Extension to the Android Platform," in Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems, Copenhagen, Denmark, 2012.

[19] Google, "/libc/include/pthread.h," [Online]. Available: https://android.googlesource.com/platform/bionic/+/android-4.2.2_r1/libc/include/pthread.h

[20] M. H. Klein, T. Ralya, B. Pollak, R. Obenza and M. G. Harbour, A practitioner's Handbook for Real-Time Analysis, USA: Kumer Academic Publishers, 1994. ISBN 0-7923-9361-9.