Technical Report

Evaluating Android OS for Embedded Real-Time Systems

Cláudio Maia

Luis Miguel Nogueira

Luis Miguel Pinho

HURRAY-TR-100604

Version:

Date: 06-29-2010

Evaluating Android OS for Embedded Real-Time Systems

Cláudio Maia, Luis Miguel Nogueira, Luis Miguel Pinho

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: crrm@isep.ipp.pt, luis@dei.isep.ipp.pt, lmp@isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

Since its official public release, Android has captured the interest from companies, developers and the general audience. From that time up to now, 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. However, there is a feature that has not been explored yet - its real-time capabilities.

This paper intends to explore this gap and provide a basis for discussion on the suitability of Android in order to be used in Open Real-Time environments. By analysing the software platform, with the main focus on the virtual machine and its underlying operating system environments, we are able to point out its current limitations and, therefore, provide a hint on different perspectives of directions in order to make Android suitable for these environments.

It is our position that Android may provide a suitable architecture for real-time embedded systems, but the real-time community should address its limitations in a joint effort at all of the platform layers.

Evaluating Android OS for Embedded Real-Time Systems

Cláudio Maia, Luís Nogueira, Luís Miguel Pinho
CISTER Research Centre
School of Engineering of the Polytechnic Institute of Porto
Porto, Portugal
Email:{crrm,lmn,lmp}@isep.ipp.pt

***Abstract*—Since its official public release, Android has captured the interest from companies, developers and the general audience. From that time up to now, 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. However, there is a feature that has not been explored yet - its real-time capabilities.**

This paper intends to explore this gap and provide a basis for discussion on the suitability of Android in order to be used in Open Real-Time environments. By analysing the software platform, with the main focus on the virtual machine and its underlying operating system environments, we are able to point out its current limitations and, therefore, provide a hint on different perspectives of directions in order to make Android suitable for these environments.

It is our position that Android may provide a suitable architecture for real-time embedded systems, but the real-time community should address its limitations in a joint effort at all of the platform layers.

***Keywords*-Android, Open Real-Time Systems, Embedded Systems**

I. Introduction

Android [1] was made publicly available during the fall of 2008. Being considered a fairly new technology, due to the fact that it is still being substantially improved and upgraded either in terms of features or firmware, Android is gaining strength both in the mobile industry and in other industries with different hardware architectures (such as the ones presented in [2] and [3]). The increasing interest from the industry arises from two core aspects: its open-source nature and its architectural model.

Being an open-source project, allows Android to be fully analysed and understood, which enables feature comprehension, bug fixing, further improvements regarding new functionalities and, finally, porting to new hardware. On the other hand, its Linux kernel-based architecture model also adds the use of Linux to the mobile industry, allowing to take advantage of the knowledge and features offered by Linux. Both of these aspects make Android an appealing target to be used in other type of environments.

Another aspect that is important to consider when using Android is its own Virtual Machine (VM) environment. Android applications are Java-based and this factor entails the use of a VM environment, with both its advantages and known problems.

Nevertheless, there are features which have not been explored yet, as for instance the suitability of the platform to be used in Open Real-Time environments. Taking into consideration works made in the past such as [4], [5], either concerning the Linux kernel or VM environments, there is the possibility of introducing temporal guarantees allied with Quality of Service (QoS) guarantees in each of the aforementioned layers, or even in both, in a way that a possible integration may be achieved, fulfilling the temporal constraints imposed by the applications. This integration may be useful for multimedia applications or even other types of applications requiring specific machine resources that need to be guaranteed in an advanced and timely manner. Thus, taking advantage of the real-time capabilities and resource optimisation provided by the platform.

Currently, the Linux kernel provides mechanisms that allow a programmer to take advantage of a basic preemptive fixed priority scheduling policy. However, when using this type of scheduling policy it is not possible to achieve real-time behaviour. Efforts have been made in the implementation of dynamic scheduling schemes which, instead of using fixed priorities for scheduling, use the concept of dynamic deadlines. These dynamic scheduling schemes have the advantage of achieving full CPU utilisation bound, but at the same time, they present an unpredictable behaviour when facing system overloads.

Since version 2.6.23, the standard Linux kernel uses the Completely Fair Scheduler (CFS), which applies fairness in the way that CPU time is assigned to tasks. This balance guarantees that all the tasks will have the same CPU share and that, each time that unfairness is verified, the algorithm assures that task re-balancing is performed. Although fairness is guaranteed, this algorithm does not provide any temporal guarantees to tasks, and therefore, neither Android does it, as its scheduling operations are delegated to the Linux kernel.

Android uses its own VM named Dalvik, which was specifically developed for mobile devices and considers memory optimisation, battery power saving and low frequency CPU. It relies on the Linux kernel for the core operating system features such as memory management and

scheduling and, thus, also presents the drawback of not taking any temporal guarantees into consideration.

The work presented in this paper is part of the CooperatES (Cooperative Embedded Systems) project [6], which aims at the specification and implementation of a QoS-aware framework, defined in [7], to be used in open and dynamic cooperative environments. Due to the nature of the environments, the framework should support resource reservation in advance and guarantee that the real-time execution constraints imposed by the applications are satisfied.

In the scope of the project, there was the need of evaluating Android as one of the possible target solutions to be used for the framework's implementation. As a result of this evaluation, this paper discusses the potential of Android and the implementation directions that can be adopted in order to make it possible to be used in Open Real-Time environments. However, our focus is targeted to soft real-time applications and therefore, hard-real time applications were not considered in our evaluation.

The remainder of this paper is organised as follows: Section II briefly describes the Android's architecture. Section III presents a detailed evaluation along with some of the Android internals and its limitations when considering real-time environments. The different perspectives of extension are detailed in Section IV. Finally, Section V concludes this paper.

II. Android's Architecture

Android is an open-source software architecture provided by the Open Handset Alliance [8], a group of 71 technology and mobile companies whose objective is to provide a mobile software platform.

The Android platform includes an operating system, middleware and applications. As for the features, Android incorporates the common features found nowadays in any mobile device platform, such as: application framework reusing, integrated browser, optimised graphics, media support, network technologies, etc.

The Android architecture, depicted in Figure 1, is composed by five layers: Applications, Application Framework, Libraries, Android Runtime and finally the Linux kernel.

The uppermost layer, the Applications layer, provides the core set of applications that are commonly offered out of the box with any mobile device.

The Application Framework layer provides the framework Application Programming Interfaces (APIs) used by the applications running on the uppermost layer. Besides the APIs, there is a set of services that enable the access to the Android's core features such as graphical components, information exchange managers, event managers and activity managers, as examples.

Below the Application Framework layer, there is another layer containing two important parts: Libraries and the Android Runtime. The libraries provide core features to

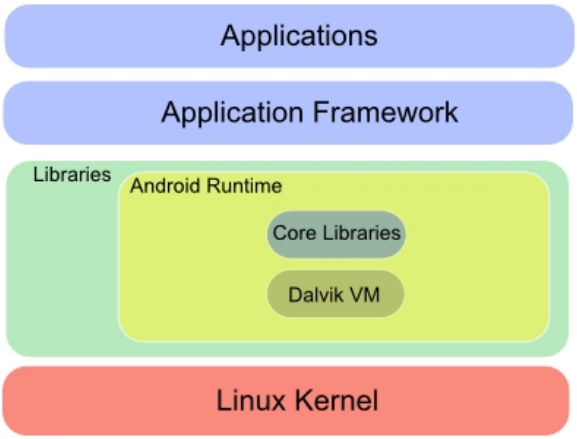

Figure 1. Android Architecture

the applications. Among all the libraries provided, the most important are *libc*, the standard C system library tuned for embedded Linux-based devices; the Media Libraries, which support playback and recording of several audio and video formats; Graphics Engines, Fonts, a lightweight relational database engine and 3D libraries based on OpenGL ES.

Regarding the Android Runtime, besides the internal core libraries, Android provides its own VM, as previously stated, named Dalvik. Dalvik [9] was designed from scratch and it is specifically targeted for memory-constrained and CPU-constrained devices. It runs Java applications on top of it and unlike the standard Java VMs, which are stack-based, Dalvik is an infinite register-based machine. Being a register-machine, it presents two advantages when compared to stack-based machines. Namely, it requires 30% less instructions to perform the same computation as a typical stack machine, causing the reduction of instruction dispatch and memory access; and less computation time, which is also derived from the elimination of common expressions from the instructions. Nevertheless, Dalvik presents 35% more bytes in the instruction stream than a typical stack-machine. This drawback is compensated by the consumption of two bytes at a time when consuming the instructions.

Dalvik uses its own byte-code format name Dalvik Executable (*.dex*), with the ability to include multiple classes in a single file. It is also able to perform several optimisations during *dex* generation when concerning the internal storage of types and constants by using principles such as minimal repetition; per-type pools; and implicit labelling. By applying these principles, it is possible to have *dex* files smaller than a typical Java archive (*jar*) file. During install time, each *dex* file is verified and optimisations such as byte-swapping and padding, static-linking and method in-lining are performed in order to minimise the runtime evaluations and at the same time to avoid code security violations.

The Linux kernel, version 2.6, is the bottommost layer and is also a hardware abstraction layer that enables the

interaction of the upper layers with the hardware layer via device drivers. Furthermore, it also provides the most fundamental system services such as security, memory management, process management and network stack.

III. Suitability of Android for Open Real-Time Systems

This section discusses the suitability of Android for open embedded real-time systems, analyses its architecture internals and points out its current limitations. Android was evaluated considering the following topics: its VM environment, the underlying Linux kernel, and its resource management capabilities.

Dalvik VM is capable of running multiple independent processes, each one with a separate address space and memory. Therefore, each Android application is mapped to a Linux process and able to use an inter-process communication mechanism, based on Open-Binder [10], to communicate with other processes in the system. The ability of separating each process is provided by Android's architectural model. During the device's boot time, there is a process responsible for starting up the Android's runtime, which implies the startup of the VM itself. Inherent to this step, there is a VM process, the Zygote, responsible for the pre-initialisation and pre-loading of the common Android's classes that will be used by most of the applications. Afterwards, the Zygote opens a socket that accepts commands from the application framework whenever a new Android application is started. This will cause the Zygote to be forked and create a child process which will then become the target application. Zygote has its own heap and a set of libraries that are shared among all processes, whereas each process has its own set of libraries and classes that are independent from the other processes. This model is presented in Figure 2. The approach is beneficial for the system as, with it, it is possible to save RAM and to speed up each application startup process.

Android applications provide the common synchronisation mechanisms known to the Java community. Technically speaking, each VM instance has at least one main thread and may have several other threads running concurrently. The threads belonging to the same VM instance may interact and synchronise with each other by the means of shared objects and monitors. The API also allows the use of synchronised methods and the creation of thread groups in order to ease the manipulation of several thread operations. It is also possible to assign priorities to each thread. When a programmer modifies the priority of a thread, with only 10 priority levels being allowed, the VM maps each of the values to Linux *nice* values, where lower values indicate a higher priority. Dalvik follows the *pthread* model where all the threads are treated as native *pthreads*. Internal VM threads belong to one thread group and all other application threads belong to another group. According to source code analysis, Android does

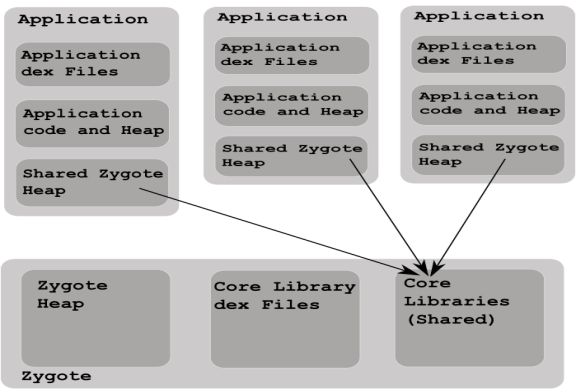

Figure 2. Zygote Heap

not provide any mechanisms to prevent priority inversion neither allows threads to use Linux's real-time priorities within Dalvik.

Threads may suspend themselves or be suspended either by the Garbage Collector (GC), debugger or the signal monitor thread. The VM controls all the threads through the use of a internal structure where all the created threads are mapped. The GC will only run when all the threads referring to a single process are suspended, in order to avoid inconsistent states.

The GCs have the difficult task of handling dynamic memory management, as they are responsible for deallocating the memory allocated by objects that are no longer needed by the applications. Concerning Android's garbage collection process, as the processes run separately from other processes and each process has its own heap and a shared heap - the Zygote's heap - Android runs separate instances of GCs in order to collect memory that is not being used anymore. Thus, each process heap is garbage collected independently, through the use of parallel mark bits that sign which objects shall be removed by the GC. This mechanism is particularly useful in Android due to the Zygote's shared heap, which in this case is kept untouched by the GC and allows a better use of the memory.

Android uses the mark-sweep algorithm to perform garbage collection. The main advantage provided by the platform is that there will be a GC running per process, which wipes all the objects from the application heap of a specific process. This way, GCs belonging to other processes will not impact the GC running for a specific process. The main disadvantage arises from the algorithm used. As this algorithm implies the suspension of all the threads belonging to an application, this means that no predictability can be achieved as that specific process will be freezed while being garbage collected.

Android's VM relies on the Linux kernel to perform all the scheduling operations. This means that all the threads running on top of the VM will be, by default, scheduled with

SCHED_OTHER, and as such will be translated into the fair scheme provided by the kernel. Therefore, it is not possible to indicate that a particular task needs to be scheduled using a different scheduling scheme.

Interrupt/event handling plays another important role when concerning real-time systems, as it may lead to inconsistent states if not handled properly. Currently, Android relies on the Linux kernel to dispatch the interrupt/event via device drivers. After an interrupt, the Java code responsible for the event handling will be notified in order to perform the respective operation. The communication path respects the architecture layers and inter-process communication may be used to notify the upper event handlers.

Currently, Dalvik does not support Just-in-Time (JIT) compilation, although a prototype has already been made available in the official repositories, which indicates that this feature will be part of one of the next versions. Other features that are also being considered as improvements are: a compact and more precise garbage collector and the use of ahead-of-time compilation for specific pieces of code.

As previously stated, Android relies on the Linux kernel for features such as memory management, process management and security. As such, all the scheduling activities are delegated by the VM to the kernel.

Android uses the same scheduler as Linux, known as Completely Fair Scheduler (CFS). CFS has the objective of providing balance between tasks assigned to a processor. For that, it uses a red-black binary tree, as presented in Figure 3, with self-balancing capabilities, meaning that the longest path in the tree is no more than twice as long as the shortest path. Other important aspect is the efficiency of these types of trees, which present a complexity of $O(logn)$, where n represents the number of elements in the tree. As the tree is being used for scheduling purposes, the balance factor is the amount of time provided to a given task. This factor has been named virtual runtime. The higher the task's virtual runtime value, the lower is the need for the processor.

In terms of execution, the algorithm works as follows: the tasks with lower virtual runtime are placed on the left side of the tree, and the tasks with the higher virtual runtime are placed on the right. This means that the tasks with the highest need for the processor will always be stored on the left side of the tree. Then, the scheduler picks the left-most node of the tree to be scheduled. Each task is responsible for accounting the CPU time taken during execution and adding this value to the previous virtual runtime value. Then, it is inserted back into the tree, if it has not finished yet. With this pattern of execution, it is guaranteed that the tasks contend the CPU time in a fair manner.

Another aspect of the fairness of the algorithm is the adjustments that it performs when the tasks are waiting for an I/O device. In this case, the tasks are compensated with the amount of time taken to receive the information they needed to complete its objective.

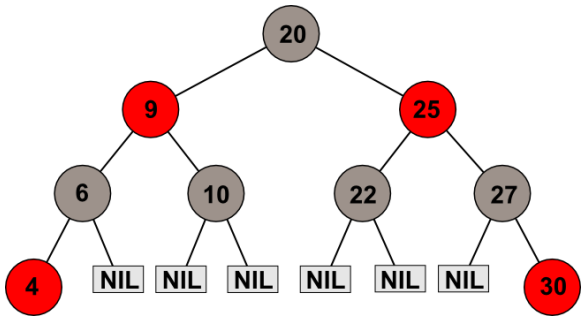

Figure 3. Red-Black Tree example

Since the introduction of the CFS, the concept of scheduling classes was also introduced. Basically, these classes provide the connection between the main generic scheduler functionalities and the specific scheduler classes that implement the scheduling algorithms. This concept allows several tasks to be scheduled differently by using different algorithms for this purpose. Regarding the main scheduler, it is periodic and preemptive. Its periodicity is activated by the frequency of the CPU clock. It allows preemption either when a high priority task needs CPU time or when an interrupt exists. As for task priorities, these can be dynamically modified with the *nice* command and currently the kernel supports 140 priorities, where the values ranging from 0 to 99 are reserved for real-time processes and the values ranging from 100 to 139 are reserved for normal processes.

Currently, the Linux kernel supports two scheduling real-time classes, as part of the compliance with the POSIX standard [11], SCHED_RR and SCHED_FIFO. SCHED_RR may be used for a round robin scheduling policy and SCHED_FIFO for a first-in, first-out policy. Both policies have a high impact on the system's performance if bad programming applies. However, most of the tasks are scheduled with SCHED_OTHER class, which is a non real-time policy.

The task scheduling plays one of the most important roles concerning the real-time features presented by a particular system. Currently, Linux's real-time implementation is limited to two scheduling real-time classes, both based on priority scheduling. Another important aspect to be considered in the evaluation is that most of the tasks are scheduled by CFS. Although CFS tries to optimise the time a task is waiting for CPU time, this effort is not enough as it is not capable of providing guaranteed response times.

One important aspect that should be remarked is that although the Linux kernel supports the real-time classes aforementioned, these classes are only available for native[1] Android applications. Normal Android applications can only take advantage of the synchronisation mechanisms described

[1]A native application in Android is an application that can run on top of the Linux kernel without the need of the VM.

earlier in this paper.

Regarding synchronisation, Android uses its own implementation of *libc* - named *bionic*. *bionic* has its own implementation of the *pthread* library and it does not support process-shared mutexes and condition variables. However, thread mutexing and thread condition variables are supported in a limited manner. Currently, inter-process communication is handled by Open-Binder. In terms of real-time limitations, the mechanisms provided by the architecture do not solve the old problems related with priority inversion. Therefore, synchronisation protocols such as priority ceiling and inheritance are not implemented.

In terms of interrupt/event handling, these are performed by the kernel via device drivers. Afterwards, the kernel is notified and then is responsible for notifying the application waiting for that specific interrupt/event. None of the parts involved in the handling has a notion of the time restrictions available to perform its operations. This behaviour becomes more serious when considering interrupts. In Linux the interrupts are the highest priority tasks, and therefore, this means that a high priority task can be interrupted by the arrival of an interrupt. This is considered a big drawback, as it is not possible to make the system totally predictable.

Resource management implies its accounting, reclamation, allocation, and negotiation [12]. Concerning resource management conducted at the VM level, CPU time is controlled by the scheduling algorithms, whereas memory can be controlled either by the VM, if we consider the heaps and its memory management, or by the operating system kernel. Regarding memory, operations such as accounting, allocation and reallocation can be performed. All these operations suffer from an unbounded and non-deterministic behaviour, which means that it is not possible to define and measure the time allowed for these operations. The network is out of scope of our analysis and thus was not evaluated.

At the kernel level, with the exception of the CPU and memory, all the remaining system's hardware is accessed via device drivers, in order to perform its operations and control the resources' status.

Nevertheless, a global manager that has a complete knowledge of the applications' needs and system's status is missing. The arbitration of resources among applications requires proper control mechanisms if real-time guarantees are going to be provided. Each application has a resource demand associated to each quality level it can provide. However, under limited resources not all applications will be able to deliver their maximum quality level. As such, a global resource manager is able to allocate resources to competing applications so that a global optimisation goal of the system is achieved [7].

IV. Possible Directions

This section discusses four possible directions to incorporate the desired real-time behaviour into the Android architecture. The first approach considers the replacement of the Linux operating system by one that provides real-time features and, at the same time, it considers the inclusion of a real-time VM. 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. The third approach simply replaces the Linux operating system for a Linux real-time version and real-time applications use the kernel directly. 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.

Regarding the first approach, depicted in Figure 4, this approach replaces the standard Linux kernel with a real-time operating system. This modification introduces predictability and determinism in the Android architecture. Therefore, it is possible to introduce new dynamic real-time scheduling policies through the use of scheduling classes; predict priority inversion and to have better resource management strategies. However, this modification entails that all the device drivers supported natively need to be implemented in the operating system with predictability in mind. This task can be painful, specially during the integration phase. Nevertheless, this approach also leaves space for the implementation of the required real-time features in the Linux kernel. Implementing the features in the standard Linux kernel requires time, but it has the advantage of providing a more seamless integration with the remaining components belonging to the architectures involved.

The second modification proposed, within the first approach, is the inclusion of a real-time Java VM. This modification is considered advantageous as, with it, it is possible to have bounded memory management; real-time scheduling within the VM, depending on the adopted solution; better synchronisation mechanisms and finally to avoid priority inversion. These improvements are considered the most influential in achieving the intended deterministic behaviour at the VM level. It is important to note that the real-time VM interacts directly with the operating system's kernel for features such as task scheduling or bounded memory management. As an example, if one considers task scheduling, the real-time VM is capable of mapping each task natively on the operating system where it will be scheduled. If the operating system supports other types of scheduling policies besides the fixed priority-based scheduler, the VM may use them to schedule its tasks. This means that most of the operations provided by real-time Java VMs are limited to the integration between the VM's supported features and the supported operating system's features.

Other advantage from this approach is that it is not necessary to keep up with the release cycles of Android, although some integration issues may arise between the VM and the kernel. The impact of introducing a new VM

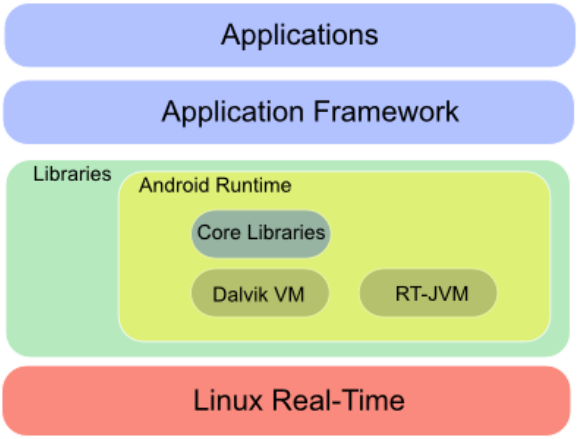

Figure 4. Android full Real-Time

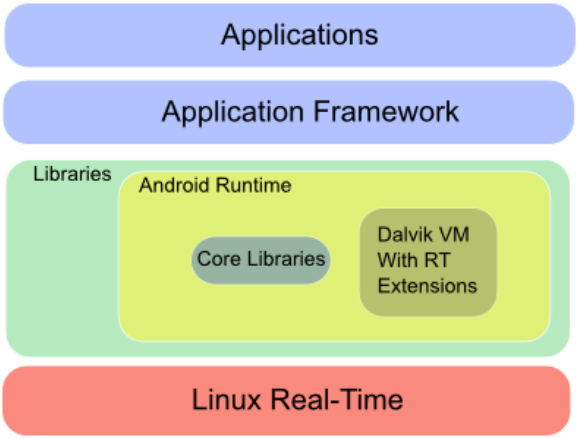

Figure 5. Android Extended

in the system is related to the fact that all the Android specificities must be implemented as well as *dex* support in the interpreter. Besides this disadvantage, other challenges may pose such as the integration between both VMs. This integration possibly entails the formulation of new algorithms to optimize scheduling and memory management in order to be possible to have an optimal integrated system as a whole and also to treat real-time applications in the correct manner.

The second proposed approach, presented in Figure 5, also introduces modifications in the architecture both in the operating system and virtual machine environments. As for the operating system layer, the advantages and disadvantages presented in the first approach are considered equal, as the principle behind it is the same. The major difference lies on the extension of Dalvik with real-time capabilities based on the Real-Time Specification for Java (RTSJ) [13]. By extending Dalvik with RTSJ features we are referring to the addition of the following API classes: *RealTimeThread*, *NoHeapRealTimeThread*, as well as the implementation of generic objects related to real-time scheduling and memory management such as *Scheduler* and *MemoryAreas*. All of these objects will enable the implementation of real-time garbage collection algorithms, synchronization algorithms and finally, asynchronous event handling algorithms. All of these features are specifically related to the RTSJ and must be considered in order to be possible to have determinism and predictability. However, its implementation only depends on the extent one wishes to have, meaning that a full compliant implementation may be achieved if the necessary implementation effort is applied in the VM extensions and the operating system's supported features. This extension is beneficial for the system as with it, it is possible to incorporate a more deterministic behaviour at the VM level without the need of concerning about the particularities of Dalvik. Nevertheless, this approach has the disadvantage of having to keep up with the release cycles of the Android, more specially the VM itself, if one wants to add these extensions to all the available versions of the platform.

Two examples of this direction are [14] and [15]. The work in [14] states that the implementation of a resource management framework is possible in the Android platform with some modifications in the platform. Although the results presented in this work are based on the CFS scheduler, work is being done to update the scheduler to a slightly modified version of EDF [16], that incorporates reservation-based scheduling algorithms as presented in [17].

The work reported in [15] is being conducted in the scope of CooperatES project [6], where a proof of concept of a QoS-aware framework for cooperative embedded real-time systems has already been developed for the Android platform. Other important aspect of this work is the implementation of a new dynamic scheduling strategy named Capacity Sharing and Stealing (CSS) [18] in the Android platform.

Both works show that it is possible to propose new approaches based on the standard Linux and Android architectures and add real-time behaviour to them in order to take advantage of resource reservation and real-time task scheduling. With both of these features, any of these systems is capable of guaranteeing resource bandwidth to applications, within an interval of time, without jeopardising the system.

The third proposed approach, depicted in Figure 6, is also based in Linux real-time. This approach takes advantage of the native environment, where it is possible to deploy real-time applications directly over the operating system. This can be advantageous for applications that do not need the VM environment, which means that a minimal effort will be needed for integration, while having the same intended behaviour. On the other hand, applications that need a VM environment will not benefit from the real-time capabilities of the underlying operating system.

Finally, the fourth approach, depicted in Figure 7, em-

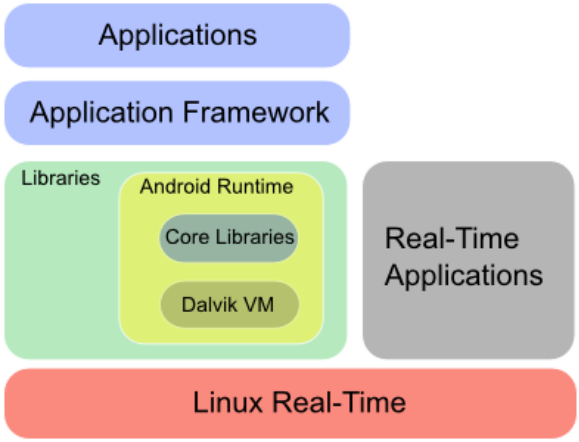

Figure 6. Android partly Real-Time

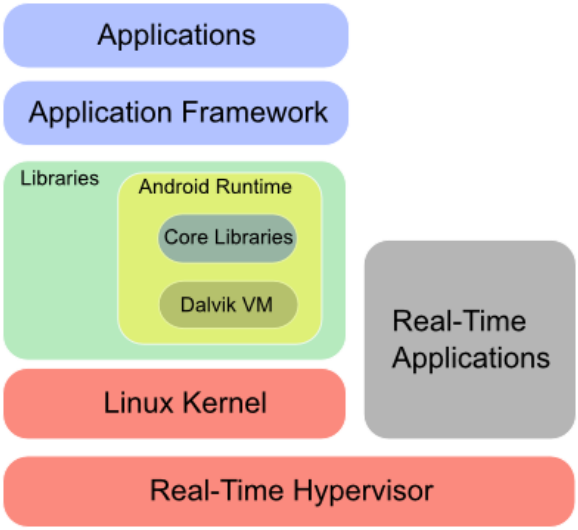

Figure 7. Android with a Real-Time Hypervisor

ploys a real-time hypervisor that is capable of running Android as a guest operating system in one of the partitions and real-time applications in another partition, in a parallel manner. This approach is similar to the approach taken by the majority of the current real-time Linux solutions, such as RTLinux [19] or RTAI [20]. These systems are able to run real-time applications in parallel to the Linux kernel, where the real-time tasks have higher priority than the Linux kernel tasks, which means that hard real-time can be used. On the other hand, the Linux partition tasks are scheduled using the spare time remaining from the CPU allocation. The main drawback from this approach is that real-time applications are limited to the features offered by the real-time hypervisor, meaning that they can not use Dalvik or even most of the Linux services. Other limitation known lies on the fact that if a real-time application hangs, all the system may also hang.

V. Conclusion

At first glance, Android may be seen as a potential target for real-time environments and, as such, there are numerous industry targets that would benefit from an architecture with such capabilities. Taking this into consideration, this paper presented the evaluation of the Android platform to be used as a real-time system. By focusing on the core parts of the system it was possible to expose the limitations and then, to present four possible directions that may be followed to add real-time behaviour to the system.

Android was built to serve the mobile industry purposes and that fact has an impact on the way that the architecture might be used. However, with some effort, as proven by the presented approaches, it is possible to have the desired real-time behaviour on any Android device. This behaviour may suit specific applications or components by providing them the ability of taking advantage of temporal guarantees, and therefore, to behave in a more predictable manner.

However, this effort must be addressed at the different layers of the architecture, in a combined way, in order to allow for potential extensions to be useful for the industry.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. This work was supported by FCT (CISTER Research Unit - FCT UI 608 and CooperatES project - PTDC/ EIA/ 71624/ 2006) and RESCUE - PTDC/EIA/65862/2006, and by the European Commission through the ArtistDesign NoE (IST-FP7-214373).

References

[1] Android, "Home page," Jan. 2010. [Online]. Available: http://www.android.com/

[2] Android-x86, "Android-x86 project," Jan. 2010. [Online]. Available: http://www.android-x86.org/

[3] G. Macario, M. Torchiano, and M. Violante, "An in-vehicle infotainment software architecture based on google android," in *SIES*. Lausanne, Switzerland: IEEE, July 2009, pp. 257–260.

[4] RTMACH, "Linux/rk," Mar. 2010. [Online]. Available: http://www.cs.cmu.edu/~rajkumar/linux-rk.html

[5] A. Corsaro, "jrate home page," Mar. 2010. [Online]. Available: http://jrate.sourceforge.net/

[6] CooperatES, "Home page," Jan. 2010. [Online]. Available: http://www.cister.isep.ipp.pt/projects/cooperates/

[7] L. Nogueira and L. M. Pinho, "Time-bounded distributed qos-aware service configuration in heterogeneous cooperative environments," *Journal of Parallel and Distributed Computing*, vol. 69, no. 6, pp. 491–507, June 2009.

[8] O. H. Alliance, "Home page," Jun. 2010. [Online]. Available: http://www.openhandsetalliance.com/

[9] D. Bornstein, "Dalvik vm internals," Mar. 2010. [Online]. Available: http://sites.google.com/site/io/dalvik-vm-internals

[10] P. Inc., "Openbinder 1.0," Mar. 2010. [Online]. Available: http://www.angryredplanet.com/~hackbod/openbinder/

[11] IEEE, "Ieee standard 1003.1," Mar. 2010. [Online]. Available: http://www.opengroup.org/onlinepubs/009695399/

[12] M. T. Higuera-Toledano and V. Issarny, "Java embedded real-time systems: An overview of existing solutions," in *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 392–399.

[13] R.-T. S. for Java, "Rtsj 1.0.2," Jan. 2010. [Online]. Available: http://www.rtsj.org/specjavadoc/book_index.html

[14] R. Guerra, S. Schorr, and G. Fohler, "Adaptive resource management for mobile terminals - the actors approach," in *Proceedings of 1st Workshop on Adaptive Resource Management (WARM10)*, Stockholm, Sweden, April 2010.

[15] C. Maia, L. Nogueira, and L. M. Pinho, "Experiences on the implementation of a cooperative embedded system framework," CISTER Research Centre, Porto, Portugal, Tech. Rep., June 2010.

[16] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[17] D. Faggioli, M. Trimarchi, and F. Checconi, "An implementation of the earliest deadline first algorithm in linux," in *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC09)*. New York, NY, USA: ACM, 2009, pp. 1984–1989.

[18] L. Nogueira and L. M. Pinho, "Capacity sharing and stealing in dynamic server-based real-time systems," in *Proceedings of the 21th IEEE International Parallel and Distributed Processing Symposium*, Long Beach,CA,USA, March 2007, p. 153.

[19] W. R. Systems, "Real-time linux," Jun. 2010. [Online]. Available: http://www.rtlinuxfree.com/

[20] P. d. M. Dipartimento di Ingegneria Aerospaziale, "Realtime application interface for linux," Jun. 2010. [Online]. Available: https://www.rtai.org/