A Real-time Extension to the Android Platform

Igor Kalkov
Embedded Software Laboratory
Ahornstr. 55
52074 Aachen, Germany
kalkov@embedded.rwth-aachen.de

Dominik Franke
Embedded Software Laboratory
Ahornstr. 55
52074 Aachen, Germany
franke@embedded.rwth-aachen.de

John F. Schommer
Embedded Software Laboratory
Ahornstr. 55
52074 Aachen, Germany
schommer@embedded.rwth-aachen.de

Stefan Kowalewski
Embedded Software Laboratory
Ahornstr. 55
52074 Aachen, Germany
kowalewski@embedded.rwth-aachen.de

ABSTRACT

Android belongs to the leading operating systems for mobile devices, e.g. smartphones or tablets. The availability of Android's source code under general public license allows interesting developments and useful modifications of the platform for third parties, like the integration of real-time support. This paper presents an extension of Android improving its real-time capabilities, without loss of original Android functionality and compatibility to existing applications. In our approach we apply the `RT_PREEMPT` patch to the Linux kernel, modify essential Android components like the Dalvik virtual machine and introduce a new real-time interface for Android developers. The resulting Android system supports applications with real-time requirements, which can be implemented in the same way as non-real-time applications.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

Keywords

Android, real-time, Dalvik VM, garbage collection

1. INTRODUCTION

Android is today one of the leading platforms for mobile devices. The fact that Android is open source distinguishes it from most other mobile platforms like iOS and Windows Phone 7. The large Android community uses this fact to port the original Android implementation to different devices, e.g. TVs, E-Readers or cameras, or to connect it to different kinds of peripheral hardware. We make use of the fact that Android is built upon a Linux kernel to extend it with real-time capabilities without losing any Android-specific functionality, while preserving backward compatibility. An Android system with the ability to serve real-time requests enlarges the field of application for Android devices to time-critical domains. For instance, real-time Android could then be used as an in-field monitoring device in industrial plants or as a control platform for home automation. The real-time extensions could also improve the core functionalities of Android itself by assuring the quality of speech processing or video playback.

In the following we present an approach to improve real-time capabilities of Android 2.2 without losing any of its original features. Further, we provide required interfaces for accessing real-time functionality on Java level, which is the common way of developing Android applications. To improve the reliability of the lower system level, we apply the `RT_PREEMPT` patch to the Linux kernel, which is the core part of the Android architecture (see Figure 2). Since Android applications are typically written in Java and executed inside *Dalvik virtual machine* (DVM) [13], we introduce necessary modifications to the DVM and the memory management system as well. Garbage collection becomes an issue in real-time systems if its invocation interrupts a process with real-time requirements or causes violations of predefined deadlines. In our approach, we allow explicit freeing of memory at the Java programming level, without triggering the automatic garbage collection. Real-time related functionality is encapsulated in one system class and provided to the developer through corresponding application programming interfaces, just like it is done for the development of conventional, non-real-time Android applications. Further, we do not inhibit any of the original features, which makes Android one of the leading mobile platforms, like connectivity or real multitasking. With our approach, real-time Android developers can access all non-real-time Android features, plus additional real-time features, in usual Android

JTRES 2012 October 24-26, 2012, Copenhagen, Denmark

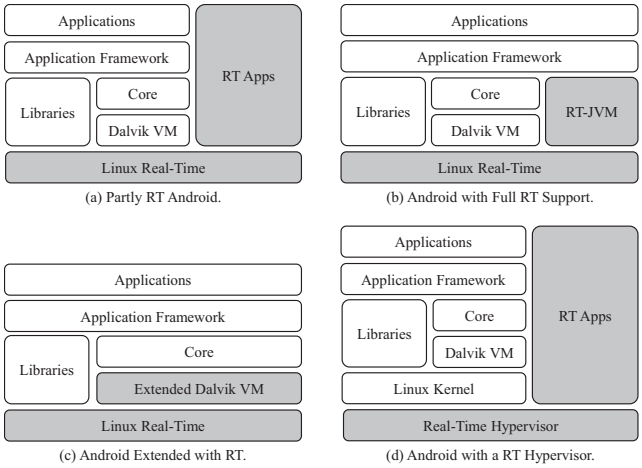

Figure 1: Possible Solutions for the Integration of RT into Android [4]

development way. For instance, starting a background service with real-time requirements is simply done by extending the class `android.app.ServiceRT`, instead of Android's native class `android.app.Service`.

This paper is structured as follows: An overview of related work is given in Section 2. Section 3 briefly presents the fundamentals on Android OS, DVM, memory management in Android and Linux `RT_PREEMPT` patch. In Section 4 the conceptual work behind our approach is explained in three parts: modifications of the Linux kernel, of the Android system core and required extensions to the Android application framework. The results of the evaluation of the real-time Android framework are presented in Section 5. Finally, Section 6 concludes this paper.

2. RELATED WORK

A detailed analysis of potential real-time capabilities in Android was done by Maia et. al [4]. They analyzed relevant parts of the Android system separately and identified its weak components. As explained in the work of Maia et al., the mainline Android implementation does not provide any support for real-time applications because of the internal design of the process and memory management. They also describe four possible ways of improving Android's architecture for real-time capability. These four ways are sketched in Figure 1 and discussed in the next two paragraphs in detail. Mongia and Madisetti provide similar conclusions based on the results of their experimental measurements [14]. Their empirical tests include responsiveness and latency measurements under three different levels of possible system load: no load, normal load and heavy load. The authors were able to show persistent violation of expected deadlines in the range of 1 ms up to 0.5 sec. As concluded in the work of Mongia and Madisetti, the analyzed version of the original Android OS was not suitable for reliable embedded applications with real-time requirements. Except for Figure 1d, where the introduction of a real-time hypervisor separates the Android platform from running real-time applications, approaches 1a to 1c imply the replacement of the shipped Linux OS by the real-time version of Linux. This step extends the properties of the original Android implementation with predictability and determinism, and allows the execution of Linux processes with real-time priorities. The simplest way to utilize a real-time capable Linux kernel is shown in Figure 1a, where all real-time applications can be written in C programming language and executed directly in a native environment on the top of the operating system. But there is no way for non-real-time Android applications to benefit from this setup, as no additional extensions of the sandboxing mechanism and the top level interfaces are included. This shortcoming is partly fixed in Figure 1b, where an additional real-time capable VM (e.g. RT-JVM) is integrated into the platform. But while this setup allows executing Java programs with real-time support, similar to 1a, it lacks on the possibility to adopt it for original Android applications. It should also be mentioned, that the integration of a second VM can lead to significant implementation overhead for providing required interfaces on the top level of the architecture and establishing an adequate infrastructure of the running system. For this reason the approach shown in Figure 1c is considered to be the most appropriate for early research, as control of the Linux real-time extensions can be propagated into the Android runtime environment basing on the slightest modifications of the original system. In this scenario, every Android application can be easily extended with required real-time properties without the need of C programming language or sophisticated combinations of multiple virtual machines.

As already mentioned above, another solution is based on the utilization of a real-time hypervisor (see Figure 1d), which is able to run the original Android platform in parallel with real-time applications. The real-time solutions RTLinux[1] and RTAI[2] use this concept [4]. Furthermore, the introduction of a real-time hypervisor seems to be advantageous on true parallel systems with multiple CPUs available. Such multi-core designs were already taken in consideration by Colin Walls in [6] and Joachim Hampp in [9] for having the Android OS and a specific RTOS running on separate cores of a single mobile device. The main drawback of this approach arises from the fact, that real-time applications are running on the top of the hypervisor or other low level interfaces. This means they do not depend on the actual Android platform and cannot use its internal components or services [4]. Also the number of potential devices is restricted to those with multiple CPUs. Although multi-core devices provide a promising concept for the future, they are used in only 20% of smartphones today[3].

3. BACKGROUND

The first part of this section gives a short overview of the Android platform. As virtual machines and garbage collection are always an issue in real-time systems [7], the second part of this section gives insights into these components on the Android platform. The third part introduces the `RT_PREEMPT` patch, which is used in this work for adding real-time capabilities to the underlying Linux kernel.

3.1 Android OS

There are various reasons why Android is so successful on mobile devices, like supporting shared data, inter-process

[1]http://www.rtlinuxfree.com
[2]https://www.rtai.org
[3]http://www.strategyanalytics.com/default.aspx?mod=reportabstractviewer&a0=7274

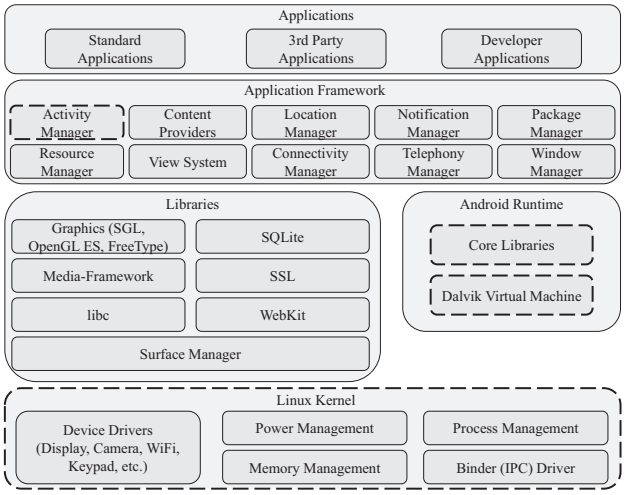

Figure 2: Android Architecture [1]

communication and real multitasking [13]. Another important reason is the open source nature of Android. This allows the community to customize the platform for own needs, port it to various hardware and include major changes in the original Android, like integrating real-time concepts. Android has a component-based architecture shown in Figure 2. The dashed frames mark parts of the architecture, which are modified in our approach to improve Android's real-time capabilities. We will go into more detail on this in Section 4. In this work we focus on Android version 2.2 since it is the most widespread Android version on the market while writing this paper. Yet, for other versions of Android, the main software architecture is not a subject of major changes. The presented work is basically applicable to all recent Android versions.

As illustrated in Figure 2, the Architecture of Android consists of four layers. The bottom layer represents the Linux kernel, which handles main services like hardware drivers, memory, power and process management [13]. On top of the kernel Android provides various C/C++ libraries for components like SSL, media playback, database and display management. Android runtime is the core part in the Android architecture. It encapsulates Java libraries and provides their functionality to the Android applications, usually written in Java. The Dalvik virtual machine is an optimized virtual machine (see Section 3.2). The application framework layer has access to both, C/C++ libraries and Android runtime. It provides various classes to create and run Android applications and serves as an abstraction level for hardware and resource access. Finally, the application layer contains installed system and user applications, which utilize the underlying framework to access hardware and make use of all functionalities that the Android platform provides. The benefits of Android's component-based design remain unchanged in the real-time version of Android presented in this paper.

This component-based architecture allows porting Android with low effort to a wide range of devices, without losing much (if any) Android-specific functionality. A real-time capable Android device could thus be deployed on other hardware platforms too, keeping all of its advantageous features.

3.2 Dalvik VM & Garbage Collection

The Dalvik virtual machine and Android's internal memory management are large areas, which are not completely of interest for this work. Due to page limitations, we therefore give a short overview of the most important aspects regarding our work and point out related references for more details.

The Dalvik virtual machine (DVM) shown in Figure 2 differs from the Java virtual machine (JVM). In contrast to JVM, DVM is not stack but register-based, works with 16 bit instructions and its input file format is .dex (Dalvik Executable) [2]. The register-based software design can lead to a performance boost, if it is adjusted to specific register-based hardware like ARM-CPUs, which many Android smartphones use today. The DVM is optimized in many different ways to run on resource constrained systems. For instance, it uses different types of verification and pre-verification during compile time to reduce the amount of just-in-time compilation during execution [15]. Just-in-time compilation is available on Android from version 2.2 on. Another enhancement is the *dexopt* tool, which is executed during compilation. It focuses on three improvements: to reduce the amount of dereferences to virtual methods, to align all opcodes and data to 16 bit and to replace certain opcodes by more efficient ones [5].

In general, each Android application is encapsulated in a separate Linux process, running inside a dedicated Dalvik VM. The process called *Zygote* plays a crucial role in Android. Zygote is started during system boot time, where it preloads and preinitializes core library classes. During the system runtime Zygote is responsible for spawning new DVM instances by using the Linux `fork()` command. With this concept Zygote provides a shared memory area for all running applications (e.g. for shared libraries), which reduces the memory footprint and startup time of new DVMs.

There are different memory areas available on Android [2] for each process:

- *Shared* memory area is available to all processes. The management of memory in this area is mainly done by Zygote, which uses it, e.g. for storing common class objects or loaded dex files. This data is used by different applications.

- *Private* memory is only available to its own process. This memory contains the application-specific dex files and the actual application heap, which is typically used for storing temporary data during the execution.

The main strategy of Android's memory management is to keep as much common information shared as possible. Each application is started in a separate instance of a DVM, having its own garbage collector. Different application-specific garbage collectors can run concurrently in their own memory segments. The garbage collector does not free shared objects while they are still referenced by at least one active process. In contrast to private data, shared data is collected separately by the Zygote garbage collector. When the Zygote garbage collector is about to start cleaning up shared memory, for reasons of consistency it makes sure all other garbage collectors are stopped [2], which is a critical issue for all running real-time applications.

The DVM garbage collector uses the *mark and sweep algorithm*, which works in two phases [16]. In the mark phase

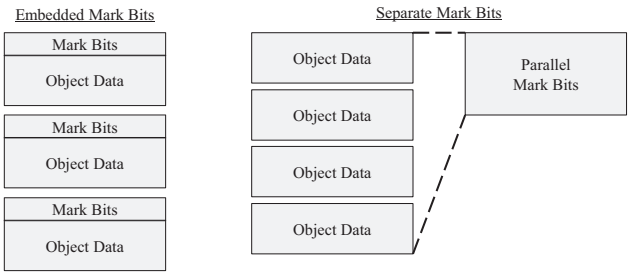

Figure 3: Two Garbage Collector Mark Bit Approaches [2]

every referenced object is marked as *being in use* by manipulating corresponding bits. In the following sweep phase GC frees all unmarked objects.

There are two different approaches to mark objects in memory for garbage collection, with embedded or separate mark bits. Figure 3 illustrates the difference. Embedded mark bits are set within an objects' memory area, while separate mark bits for different objects are kept together in a separate memory area. Android uses the separated mark bits approach [2].

3.3 RT-Preempt Patch

The `RT_PREEMPT`[4] patch, also known as `CONFIG_PREEMPT_RT` patch, is a kernel patch introducing a preemtible Linux kernel with improved real-time capabilities. It was developed by Ingo Molnar and is available open source. Figure 4 presents an overview of the different preemption classes which are available on a patched Linux.

The hard interrupt service routine (ISR) layer includes hardware interrupts. If this type of interrupt is enabled, such routines can preempt every other process on the system. The real-time patch also converts interrupt handlers into preemptible Linux kernel threads [11]. These so called virtual ISRs are scheduled amongst real-time tasks (RT-Tasks) depending on their priority. A task with a higher priority can preempt tasks with a lower priority. In case a low priority task holds a resource, which a higher priority task needs to access, priority inheritance for in-kernel spinlocks and semaphores is implemented, too [8]. Conventional processes with no real-time requirements are scheduled according to the Linux scheduler[5] class `SCHED_NORMAL` (also called `SCHED_OTHER`) [3]. There are further enhancements introduced by the real-time patch, like making in-kernel locking-primitives preemptible or converting the Linux timer API into high-resolution kernel timers which are POSIX-conformant [12], but these enhancements are not in the focus of our work.

4. BRINGING REAL-TIME TO ANDROID

This section explains required modifications and extensions of the Android system in order to improve its real-time capabilities. First, the Linux kernel modifications are explained. Second, platform-related modifications in the DVM and memory management are presented. The third subsection gives insight in how the real-time functionalities are integrated into the existing Android application framework and propagated to the application developer.

[4]See http://www.kernel.org/pub/linux/kernel/projects/rt .
[5]See https://lkml.org/lkml/2007/4/13/180 .

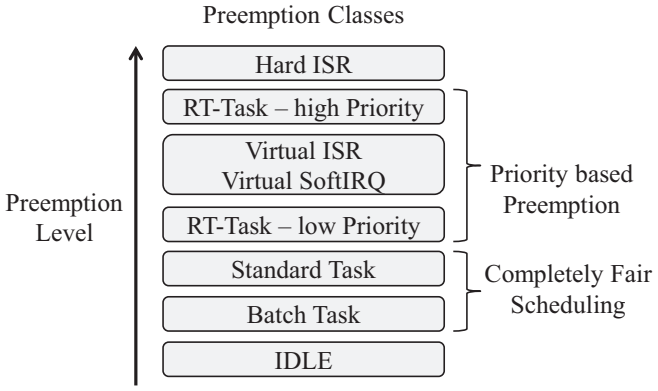

Figure 4: Preemption Classes on a RT-Preempt patched Linux Kernel

4.1 Linux Kernel Modifications

The integration of real-time into the Android platform requires a real-time capable version of the underlying Linux kernel. The kernel included in the standard Android 2.2 distribution is based on the mainline Linux kernel version 2.6.29. A full replacement of the Android's standard kernel with any of existent real-time solutions is considered time-consuming. The new RTOS would require major modifications for preserving compatibility with existing Android components. For this reason we choose the `RT_PREEMPT` patch for introducing necessary modifications towards a fully preemptible kernel with real-time capabilities.

Top-level Android components are executed on the top of a specific kernel variant, which depends on the current hardware. There exist different versions of the Linux kernel customized and optimized for different mobile devices, but all of them are based on the generic mainline kernel. In this work we focus on two important variants: the Goldfish kernel, which is used in the Android Emulator, and the kernel designed for real devices, like the device used in Section 5 for evaluation purposes.

The Android Emulator runs a virtual CPU called Goldfish. It can execute ARM926T (ARMv5) instruction set commands and provides additional hooks for input and output, e.g. for reading key presses or displaying video. These specific features are extensions to the generic kernel, implemented in separated source files. Many features of the Goldfish distribution not included into the standard Linux kernel have to be adjusted and modified manually in addition to the `RT_PREEMPT` patch. For instance, one such Goldfish-specific modification is required for internal timers, where the architectural implementation of a timer spinlock has to be adjusted according to the correct annotation. Other important patch adaptations have to be done for mutexes, IRQ handling and other system components.

Similar to the Goldfish kernel, Android's support for real devices based on Qualcomm's Mobile Station Modem (MSM) chipsets requires a set of specific interfaces in addition to the generic Linux kernel. For adapting this kernel, we choose the same approach, as for adapting the Goldfish kernel. Single patches of the `RT_PREEMPT` set require minor adjustments with respect to the characteristics of the MSM kernel. Modi-

fied version of `RT_PREEMPT` leads to a preemptible kernel with support for reliable scheduling of native Linux processes.

4.2 Android Platform Modifications

The successfully patched Linux kernel is already able to provide fundamental support for native Linux real-time processes. To allow Android applications to make use of the real-time functionality, further adjustments in Android core libraries, virtual machine and memory management are necessary.

4.2.1 Core Libraries

Real-time applications require root privileges in order to change the current Linux scheduling class. But in the original Android implementation the permission to gain root privileges is only allowed for the system server and shell of the debugging bridge. This restriction needs to be relaxed without overriding internal security mechanisms of the operating system. Allowing unrestricted access to the `su` command, which grants the user root privileges on Linux, is not acceptable. Instead, each privilege elevation has to be done in a controlled manner, such that the active user is notified and asked for his explicit permission for every particular action. This procedure assures that the user is notified each time an application is about to gain root privileges. Additionally the user can inhibit each root access. This kind of access control can be covered by *Superuser*[6] - a widespread, open source Android application for controlling privileged access on rooted Android devices. Superuser is implemented to intercept all internal calls to the `su` command by replacing its binary in the system directory. After this replacement, every request for root privileges from a running application is redirected to the device user as a GUI dialog. As the core part of the Superuser source code is available in text format, it can be easily represented as a system patch by discarding minor binary resources.

Selecting real-time priority values as well as changing current scheduling classes can be realized by calling `sched_setscheduler()` function of the patched kernel. But included in a bundle of native C libraries, this function is not available to top-level Android applications without prior forwarding. One solution is to extend the set of available core libraries with the functionality to handle top-level user requests for scheduling class or priority level changes. This additional native library could be used on demand in the application framework and would internally call the function `sched_setscheduler()` in the context of the calling process. A variant of such a library is already shipped with the original Android distribution as a tool called `renice`, which is part of Android's system toolbox. This tool can be used for gathering process information or influencing internal process scheduling. It also allows changing process priority values and re-assignment of processes to different scheduling classes. The Linux scheduler uses priorities within the range [0..99] for handling real-time tasks and the range [100..139] for all other tasks. A new non-real-time process gets the priority value 120, by default [10]. Android's original `renice` tool implicitly selects the `SCHED_RR` (Round-Robin) scheduling policy every time a real-time priority value is specified. This approach might cause unexpected system behavior if several real-time processes are running simultaneously and the developer assumes a FIFO scheduling order (`SCHED_FIFO`). This issue is fixed by extending the `renice` tool with the possibility to explicitly declare the desired scheduling policy in conjunction with the value for real-time priority. Further, real-time processes can be easily switched back to the `SCHED_OTHER` scheduling class, if they give up their real-time requirements. With these modifications the original `renice` tool becomes more flexible.

[6]See http://code.google.com/p/superuser .

4.2.2 Dalvik VM

The Dalvik VM (DVM) is one of the main components of the Android system. The DVM was designed with a focus on executing complex applications on mobile devices with limited resources. It is optimized for running with low memory requirements, where the main part of the actual memory management is done by the underlying Linux kernel. However, Android introduces additional facilities for monitoring memory allocations and ensuring memory usage in the most efficient way.

4.2.3 Memory Management

Android's *low memory killer* handles distinct application classes, which represent the importance of user processes. Processes with foreground activities are classified as more important than processes without user interactions. If the system is running out of memory, low memory killer iteratively terminates running applications starting with the most unimportant one. To additionally distinguish between processes of the same class, Android assigns every running process p_i a specific *Out Of Memory* (OOM) Adjustment Value $oom\_adj(p_i)$. This value depends on the corresponding application class, but may be dynamically changed during the process' lifetime. One of the main advantages of this approach is the ability of the system to reflect relevant priority changes during runtime. For instance, the importance of a background process can be temporarily increased (by decreasing oom_adj), if this process sends a visual notification to the user. Similar, transitions from foreground to background are handled by increasing the application's oom_adj value, which lowers the importance of this process.

Android also defines memory threshold values m_thr_l, which indicate six different levels of available system resources. For every moment in time, the set of killable applications depends on the amount of the currently available memory m_{free} as

$$\{p_i \mid m_{free} < m\_thr_l \wedge adj\_thr_l < oom\_adj(p_i)\}$$

for the corresponding level $l \in [0, ..., 5]$. Priority-specific threshold values adj_thr_l, as well as different memory levels m_thres_l are defined during the system startup for all possible $l \in [0, ..., 5]$. For example, values $mem\_thr_3 = 5120$ pages[7] and $adj\_thr_3 = 7$ will cause low memory killer to start terminating processes with $oom\_adj > 7$ as soon, as the amount of free memory gets lower than 20 MB[8].

Operating systems with support for reliable real-time applications must ensure continuous execution of processes with real-time requirements. In a safety-critical context, undesired killing of a real-time application might lead to destructive effects. For this reason, Android applications with real-time requirements must get the lowest possible oom_adj values in order to avoid enforced termination. Thus in our ap-

[7]Memory pages with 4096 bytes per page (BPP)

[8]5120 pages × 4096 BPP = 20971520 bytes

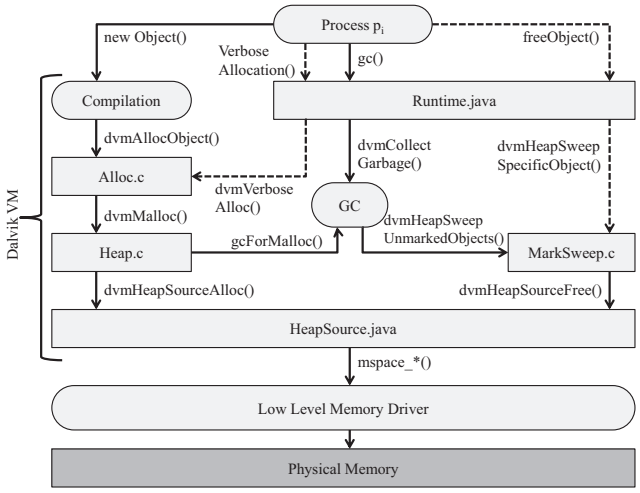

Figure 5: Extended Memory Management

proach the class `ProcessRecord` of Android's *Activity Manager* package `com.android.server.am` is extended by the flag *isRealtime* marking applications with real-time requirements. This flag can be internally accessed by the system, e.g. during the assignment of *oom_adj* values. Additionally, the Activity Manager itself and the corresponding Activity Manager System Service are modified in order to allow preferential treatment of real-time processes in the internal process management.

4.2.4 Garbage Collection

As explained in Section 3, the deployed mark-and-sweep algorithm for garbage collection might have a negative impact on determinism and predictability of the process behavior: Although its implementation as a part of the Dalvik VM results in process-independent GC and does not affect other running applications, the mark-and-sweep algorithm requires exclusive memory access for keeping the memory state consistent. This restriction leads to an enforced suspension of all threads of the active application each time automatic GC is started. The lack of predictability as well as pausing program execution up to 500 ms during the GC invocation [14], show that applications with real-time requirements need a different solution for memory management.

Disabling automatic GC completely is not reasonable in the context of Android applications. Every Android process uses its dedicated memory block, where the garbage collector is invoked only if there is no free memory available and a memory allocation request would fail otherwise. Disabling automatic GC in such setups leads to dangerous OOM situations, which can cause unpredictable system behavior. A more promising solution for avoiding GC-related process suspensions is to allow the developer to free memory explicitly. Giving developers the possibility to manually free unused objects increases the degree of active memory recycling and prevents unnecessary invocations of GC. Figure 5 is derived from source code analysis and reflects the core of Android's own memory management facilities. The transitions represent system events, like method calls, and show the corresponding source files. Rounded rectangles encapsulate abstract entities and complex system components. Extensions to the original system, which are introduced in this work, are highlighted with dashed lines. During compilation of an Android application, each object instantiation with the keyword `new` is translated into `dvmAllocObject()` call. This method gets a reference to the class of the requested object as an argument. Among other properties like the name of the instantiated class, this reference contains the size of the desired object. The size is then passed to the `dvmMalloc()` function for allocating the necessary amount of memory. In general, `dvmMalloc()` is responsible for error handling and initialization of common object properties, while the actual memory allocation is done in `dvmHeapSourceAlloc()`, which uses the low-level memory driver[9] and returns the result without any validation. A successfully allocated memory block is converted to the requested object and program execution continues. But a call to `dvmHeapSourceAlloc()` might also fail, returning a null-pointer and indicating an OOM situation. In this case, the function `dvmMalloc()` tries to solve this problem by an explicit invocation of the automatic GC. Afterwards either memory allocation is repeated or an OOM exception is thrown. Figure 5 illustrates this step by the `gcForMalloc()` transition.

Objects which remain unmarked after the mark phase of the GC algorithm are considered unreachable and will be destroyed in the sweep phase. The list of identified dispensable objects is passed to the function `dvmHeapSweepUnmarkedObjects()`, which releases corresponding memory by calling `dvmHeapSourceFree()` for every single object reference.

The standard Android implementation already provides a possibility to influence current memory load by calling the `Runtime.gc()` method manually. Explicit calls to `gc()` have the same negative effects on running applications, as implicit invocations of GC from `dvmMalloc()`. The resulting suspension of running threads is not acceptable in time-critical systems. Thus we strongly discourage using `gc()` for memory deallocations in the context of real-time applications.

Our approach follows the idea of freeing allocated memory blocks explicitly without the `gc()` method. Therefore we focus on the method `dvmHeapSourceFree()` (see Figure 5). It provides an interface for destroying unused objects without explicit activation of GC. We introduce a new method `dvmHeapSweepSpecificObject()` inside the DVM, which internally makes use of `dvmHeapSourceFree()`. Additionally, we extend the class `Runtime` by the method `freeObject()` in order to make the new functionality available inside Android applications. As an argument `freeObject()` receives an object to free. It calculates the pointer to a memory block containing this object and passes it to `dvmHeapSweepSpecificObject()`, where the actual freeing is done using `dvmHeapSourceFree()`. After memory is released this way, it can be immediately reused for upcoming allocations.

A disadvantage of this approach is that the method `freeObject()` can only be used for objects which the developer is aware of. But during a process' lifetime, memory can also be filled with objects from temporary allocations. For example, Android may instantiate objects for local variables or as intermediate step for complex operations. Iterative execution of this code will reduce the amount of available memory and eventually result in GC activation. One solution is the extension of the `Runtime` class by adding a method `verboseAllocations()`, which we introduce for pro-

[9]Memory driver is a part of the `libc` library.

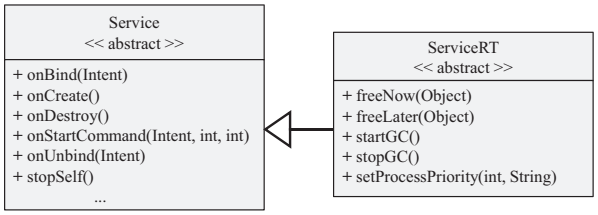

Figure 6: Extension of the Android Service Class

viding better transparency for background allocations. After a call to this method, all created objects inside the process' own memory segment will be logged in the system log with their class name, size and the current memory address. This method can be used during software development for optimization of the application's memory usage.

4.3 Application Framework Extensions

With the modifications introduced in previous sections and an elaborated use of the proposed real-time functionality, Android applications can be executed in real-time prioritized Linux processes without being killed by the OS due to low memory. Furthermore dynamic changes of the process' priority with the `renice` tool and additional methods for manual memory management are implemented. All these features complement each other and can be used in combination for applications with real-time requirements. Introduced extensions are distributed over different system components and mainly affect the internals of the OS. For providing the functionalities to the developer in a more convenient way, we include a corresponding programming interface on an application framework level. The idea is to provide the real-time capabilities encapsulated in a well-known and integrated environment to the developer. For this reason we extend the standard Android distribution by the `android.app.ServiceRT` class. As illustrated in Figure 6, this class utilizes all real-time features and provides the real-time functionality to the Android developer in a consistent way. The `ServiceRT` class encapsulates the real-time functionality and controls the use of relevant tools. For instance, it provides a method `setProcessPriority()`, which changes the current execution mode by selecting a real-time scheduling class. Depending on the passed arguments, `setProcessPriority()` executes `renice` with corresponding arguments. Furthermore, this method handles the preparation of the required `su` calls and validation of the return value.

The possibility to free unused objects without suspension of the whole process, allows implementations of manual memory management. Therefore a queue of objects and an internal thread are integrated into the `ServiceRT` class. Objects that are not used anymore can be freed immediately with `freeNow()` or can be inserted into the queue using the `freeLater()` method, for a delayed destruction (see Figure 6). When the queue is filled, the internal thread wakes up and iteratively frees enqueued objects by calling `freeObject()`, concurrently to the execution of the main program (see Figure 7). To ensure that the main application thread is not disturbed, the priority of the internal thread can be lowered by already existing Java methods like `Thread.setPriority()`. Aditionally, `ServiceRT` offers two methods `startGC()` and `stopGC()` which can enforce execution and termination of the concurrent GC thread.

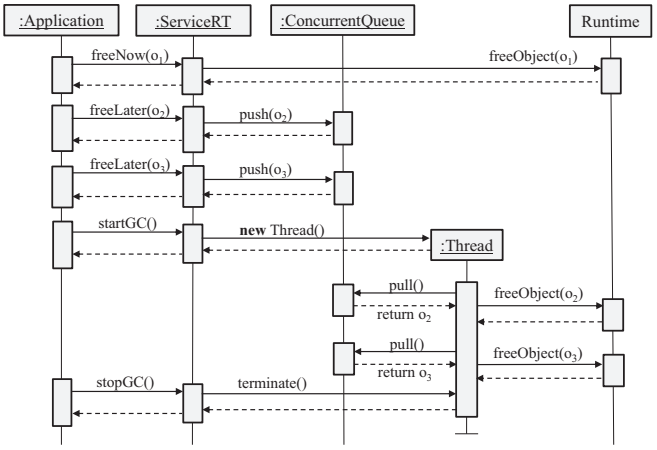

Figure 7: Concurrent GC inside the ServiceRT Class

In comparison to the basic approach, the only difference for implementation of reliable services for Android developers is the choice of the base class (`android.app.ServiceRT` instead of `android.app.Service`) and the usage of introduced methods in a reasonable way. Deriving from `ServiceRT` assures that the *oom_adj* value for the current process is automatically adjusted in the system core during the process startup, which prevents the created application from being killed by the OS in critical OOM situations. This approach is more convenient, than changing the *oom_adj* value manually via an additional method. It also seems to be beneficial for the overall behavior predictability, as the related process cannot be unexpectedly terminated before the corresponding method is called.

Figure 8 shows the startup of a real-time service in a sequence chart. After spawning a new Linux process, Android instantiates a new service object, registers it in the system and resets the *oom_adj* value according to the object's properties. After initialization finishes, the service is executed with standard application priority, like a non-real-time service. In the scenario of Figure 8 the service internally calls `setProcessPriority()` to acquire real-time priority. As described above, this method encapsulates invocation of the `renice` tool, which changes the priority value and scheduling class for the underlying Linux process. The corresponding service is then running with real-time priority. From this moment on, it can be used for execution of time-critical tasks.

5. EVALUATION

Evaluation of the resulting real-time Android system is done in two parts. The first part of this section presents the evaluation of the introduced memory management. The second and third parts show the time behavior of the process during memory allocation and scheduling of tasks.

All tests are executed on the Android smartphone *HTC Dream*. This device has a 528 MHz MSM7201A ARM11 processor and 192 MB of RAM. Furthermore, it supports up to 16 GB external storage, a number of different sensors and common connectivity interfaces like Wi-Fi (802.11b/g) and Bluetooth.

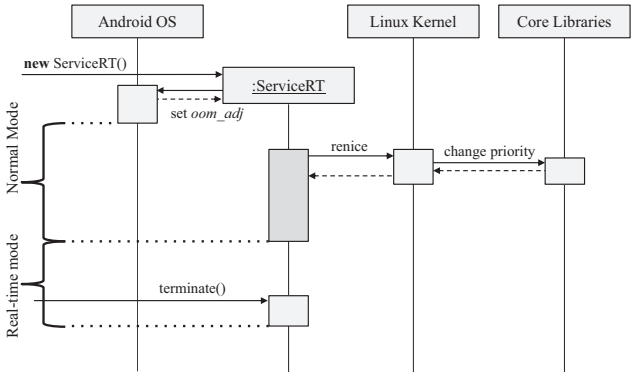

Figure 8: Starting a Real-time Service

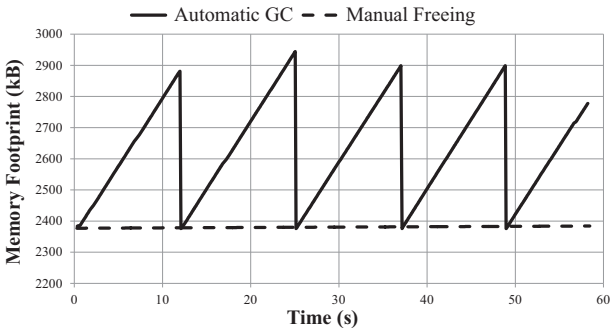

Figure 9: Application Footprints

5.1 Garbage Collector Test

Unpredictable or often triggered garbage collection can have a bad impact on time behavior of an application (see Section 3.2). In this test the application receives 2,000 data packets with a size of 1,210 bytes each, at maximum rate over a 54 Mbit Wi-Fi connection. These data packets include information which the mobile device has to present in a graphical user interface. After extracting the required information the reserved memory of the allocated packet object can be released.

The test is executed twice on our real-time Android system. Both times the application is started in a non-real-time process - the default setting for Android applications. During the first test run, memory is freed on demand by the automatic invocation of the garbage collection. During the second test run memory is freed explicitly by the process using the introduced `freeObject()` method. Memory footprint of the process is measured each time after receiving a new packet. Figure 9 presents the results of the test execution. Memory footprint of the process with automatic GC depends on the system. As soon as the amount of the used memory reaches a certain threshold, which seems to be at around 2,900 kB, the DVM triggers automatic GC for this process. During each GC run the process and all of its threads are suspended (see Section 3.2). The footprint of the process with manual object freeing does not visibly change. After receiving all 2,000 data packets the footprint only increased by 7.1 kB. During the whole process execution it was not necessary to trigger GC, since the size of allocated memory remains the same.

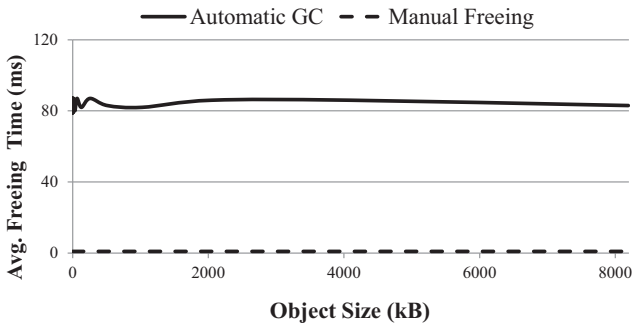

Figure 10: Duration of Deallocation

5.2 Deallocation Test

Predictability is a very important property of real-time systems. This holds also for allocating and deallocating memory. Since we introduced an improved way of freeing objects, this test shall determine the time behavior of our approach.

Therefore we start a real-time Android service (instance of `ServiceRT`), which periodically allocates an object with a certain size and afterwards frees it again. We measure the time to free each object. This procedure is repeated 10 times for each object size from 1 byte to 8 megabytes (1 byte, 2 bytes, 4 bytes, 8 bytes, ..., 8 megabytes). In the first test case the service uses the explicit `gc()` call to free memory, in the second case the method `freeObject()` is used. Figure 10 presents the results of this test. The average freeing time using GC seems to be independent of the object size. It remains between 80 – 90 ms. The reason is that GC needs to scan the whole memory block of the process each time it is executed. Freeing memory using our `freeObject()` method seems to be constant as well. For each measured object size the average freeing time is less than 1 ms.

The average memory freeing time seems to be deterministic in both cases. But on Android an application might get a different amount of memory, each time it is started. This depends on the current memory situation: If there is not much free memory available on the system, a freshly started application will get a corresponding low amount of available memory. Thus the execution time of the automatic GC may vary, if an application is restarted. While GC is suspending the executed application for around 85 ms each time it is executed, even if there is only one object with a size of 1 Byte to free, `freeObject()` needs less than 1 ms. Further, the execution time of GC is dependent on the amount of available memory for an application. If the size of the memory block increases, automatic GC has to check more memory and thus needs more time. With our approach, we notably reduce the amount of required GC invocations during the application runtime.

5.3 Timer Task Test

Meeting specified deadlines is an essential requirement on real-time systems, where the execution of scheduled task is often triggered by timers. This test checks if real-time Android applications are able to hold deadlines and measures actual latencies. For this area we did quite a lot of tests. Due to page limitations we will present only two of them, but refer to results from other tests, where appropriate.

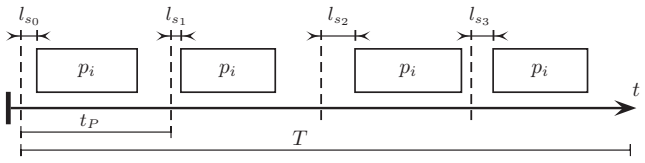

Figure 11: Computing the Latency

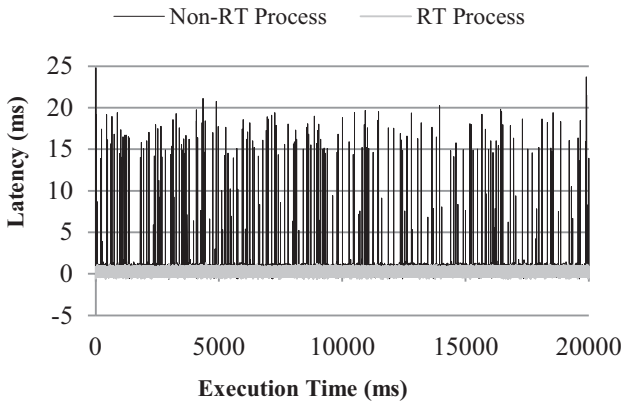

Figure 12: Latencies on Idle System

In this test the Android application has to execute a task periodically each $t_P = 5$ ms, for a fixed period of $T = 20$ s. We compute the latency l_s as the time difference between the expected and measured execution time. This setup is illustrated in Figure 11. In one test case the application is executed as a real-time application with process priority of 40 and in another test case as a standard Android application, running with the default system priority of 120. Both test cases are executed on the patched real-time Android platform. In the first test no additional applications are started, except for the default system applications and services, which are part of the standard Android boot process. In the second test we use the same setup, but start an additional user application to cause a high system load. This additional application is recursively computing Fibonacci numbers running in a process with the default process priority.

Figure 12 presents the results of the first test with no additional application running in the background. The non-real-time test application has latencies between -0.63 ms and 24.7 ms, although no additional CPU load is caused by a concurrent thread. The same test application, being executed with real-time priority, has latencies only between -0.63 ms and 1.25 ms. This is not only the case for the task execution interval $t_P = 5$ ms, but also for $t_P = 1$ ms, $t_P = 10$ ms, $t_P = 100$ ms and $t_P = 1$ s. Different values for t_P and T seem not to make a difference, as further tests show. The resulting latencies for the real-time test application stay within the same ranges.

When putting some load on the system by challenging the CPU with an additional non-real-time process, the latencies for the real-time test application stay within similar ranges: between -0.63 ms and 1.53 ms (see Figure 13). The process gets the required resources to execute its task in time. The non-real-time test application has many more problems

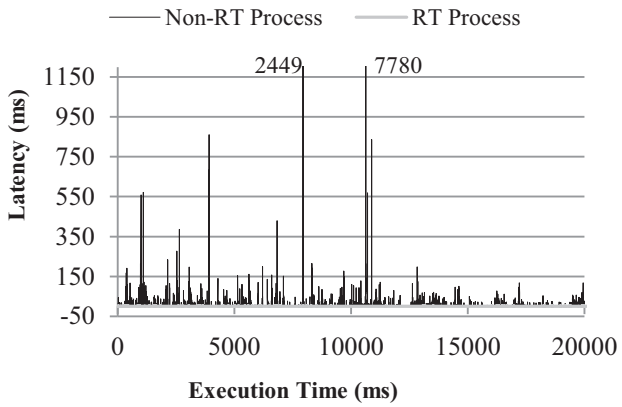

Figure 13: Latencies on System under Load

to get the required resources, like CPU and RAM. In the worst case, latency increases from 24.7 to 7,780.82 ms when another background process causes additional system load. Such large delays have massive impact on the application execution: video and audio playback would perceptibly halt and a 5 ms deadline would be missed around 1550-times.

6. CONCLUSION

This paper presents an approach to extend the mobile platform Android with real-time capabilities. The approach uses the `RT_PREEMPT` patch to equip Android's Linux kernel with basic real-time support. Further modifications in other Android components allow manual memory freeing on Android application level and propagate real-time core functionalities to the application layer. The developer can use the fully integrated features in the standard Android manner, e.g. simply by extending a different class in case of real-time service.

We evaluate the resulting system in three tests. The garbage collector test shows that manual freeing of objects during runtime works well. By using corresponding methods, object freeing and memory load can be controlled by the developer. Usage of explicit memory management can lead to better time behavior of the application by avoiding invocations of the automatic GC. Furthermore, this approach provides an improved, fine-grained control of memory allocations inside an Android application with the typical comfort of Java programming. As the deallocation tests show, the time for object freeing remains, even for objects with a size of 8 MB, less than 1 ms. An appropriate handling of objects can reasonably reduce the number of GC runs and thus the risk of a real-time application being disturbed by GC. Selecting real-time priorities inside of Android applications additionally reduces disruption by concurrent processes, as for instance the timer task test shows. With the right priority set, the execution of a real-time application is delayed during the test by at most 1.53 ms, even if applications with a lower priority put a high load on the CPU. The execution latency stays deterministic within the range of -0.63 – 1.53 ms. For Android applications using the default system priority the latency is not deterministic, as the timer task test shows: The application under test reaches a worst case delay of 7,780.82 ms during the test run, if a concurrent

application puts a high load on the CPU.

The test results highlight that the original Android implementation without any modifications is not capable of serving real-time requests. Our approach shows how this mobile platform can be improved towards its real-time capability. But there still remain open issues. For instance, during allocation and freeing of objects in the garbage collector test, the memory footprint of the test application grew by 7.1 kB. This memory is allocated by inner Android classes, e.g. in methods for updating the user interface. In the current version of our approach the developer cannot free objects that he does not allocate by himself. This means the developer has to invoke garbage collection explicitly, depending on which Android methods he uses. Else GC will be eventually triggered automatically in case of OOM. We investigate suitable approaches for nonblocking real-time GC to avoid OOM situations effectively. Another part of future work results from the chosen approach for the modification of the Linux kernel. Scheduling latencies and deterministic process execution on the Linux level depend on suitability and quality of the `RT_PREEMPT` patch.

7. ACKNOWLEDGMENTS

This work was supported by the UMIC Research Centre, RWTH Aachen University, Germany.

8. REFERENCES

[1] A. Becker and M. Pant. *Android 2*. dpunkt.verlag, Heidelberg, Germany, 2010.

[2] D. Bornstein. *Dalvik VM Internals*. IO-Google Conference, San Francisco, USA, 2008.

[3] D. Bovet and M. Cesatí. *Understanding the Linux Kernel*. O'Reilly Series. O'Reilly, 2006.

[4] C. Maia, L. 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*, OSPERT 2010, pages 63–70, Brussels, Belgium, 2010.

[5] C.-W. Chang, C.-Y. Lin, C.-T. King, Y.-F. Chung, and S.-Y. Tseng. Implementation of JVM tool interface on Dalvik virtual machine. In *Proceedings on VLSI-DAT 2010*, pages 143–146. IEEE, April 2010.

[6] Colin Walls. Android, Linux and Real-time Development for Embedded Systems. Mentor Graphics. Slides, 2010.

[7] W. Foote. Theory versus practice in real-time computing with the java(tm) platform. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '99, pages 105–, Washington, DC, USA, 1999. IEEE Computer Society.

[8] D. Hart, J. Stultz, and T. Ts'o. Real-time linux in real time. *IBM Systems Journal*, 47(2):207–220, April 2010.

[9] Joachim Hampp. Android not just a OS for mobile phones. Wind River GmbH. Slides, 2010.

[10] J. Kobus and R. Szklarski. Completely Fair Scheduler and its tuning. http://www.fizyka.umk.pl/ jkob/pracemag/cfs-tuning.pdf, 2009.

[11] S. Kume, Y. Kanamiya, and D. Sato. Towards an open-source integrated development and real-time control platform for robots. In *Proceedings on ROBIO 2008*, pages 204–209. International Conference on Robotics and Biomimetics, IEEE, May 2009.

[12] M. D. Marieska, P. G. Hariyanto, M. F. Fauzan, A. I. Kistijantoro, and A. Manaf. On performance of kernel based and embedded real-time operating system: Benchmarking and analysis. In *Proceedings on ICACSIS 2011*, pages 401–406. International Conference on Advanced Computer Science and Information System, IEEE, December 2011.

[13] R. Meier. *Professional Android 2 Application Development*. Wrox Press Ltd., Birmingham, UK, 1st edition, 2010.

[14] B. S. Mongia and V. K. Madisetti. Reliable Real-Time Applications on Android OS. Whitepaper, 2010.

[15] C. U. Nicola. *Einblick in die Dalvik Virtual Machine*. University of Applied Sciences and Arts Northwestern Switzerland, Brugg-Windisch, Switzerland, 2009.

[16] F. Wei and C. Hauser. Modeling real-time garbage collection cost. In *Proceedings on RTCSA 2007*, pages 217–225. IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE, September 2007.