Real-Time Android with RTDroid

Yin Yan, Shaun Cosgrove, Varun Anand, Amit Kulkarni,
Sree Harsha Konduri, Steven Y. Ko, Lukasz Ziarek
Department of Computer Science and Engineering
University at Buffalo, The State University of New York
{yinyan, shaunger, varunana, amitshri, sreehars, stevko, lziarek}@buffalo.edu

ABSTRACT

This paper presents RTDroid, a variant of Android that provides predictability to Android applications. Although there has been much interest in adopting Android in real-time contexts, surprisingly little work has been done to examine the suitability of Android for real-time systems. Existing work only provides solutions to traditional problems, including real-time garbage collection at the virtual machine layer and kernel-level real-time scheduling and resource management. While it is critical to address these issues, it is by no means sufficient. After all, Android is a vast system that is more than a Java virtual machine and a kernel.

Thus, this paper goes beyond existing work and examines the internals of Android. We discuss the implications and challenges of adapting Android constructs and core system services for real-time and present a solution for each. Our system is unique in that it redesigns Android's internal components, replaces Android's Java VM (Dalvik) with a real-time VM, and leverages off-the-shelf real-time OSes. We demonstrate the feasibility and predictability of our solution by evaluating it on three different platforms—an x86 PC, a LEON3 embedded board, and a Nexus S smartphone. The evaluation results show that our design can successfully provide predictability to Android applications, even under heavy load.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Design, Measurement, Experimentation, Performance

Keywords

Real-time Systems, Mobile Systems, Smartphones, Android

1. INTRODUCTION

There is a growing interest in adopting Android in embedded, real-time environments. A DARPA project utilizing Android is currently in development, which creates a plug and play navigation and sensor network that can scale from personal devices up to aircraft navigators [23, 24]. The UK has recently launched a satellite equipped with an Android smartphone to explore the possibility of using a smartphone as a control system [30]. In health care, much discussion is currently ongoing as to how the medical device industry can adopt Android [1, 4, 6, 7]. In these domains, the benefits are numerous; developers can leverage Android's rich set of APIs to utilize new types of hardware such as sensors and touch screens; Android's well-supported, open-source development environment eases application development; and many applications published in online application stores give an opportunity to incorporate creative functionalities with less effort.

However, surprisingly little work has been done in actually adding real-time capabilities in Android. The current literature only provides a short overview of potential high-level system models [21] and extensions to Android's Java VM (Dalvik) that enable real-time garbage collection [13, 19]. The fundamental question of how to add real-time support to Android *as a whole system* has not been explored.

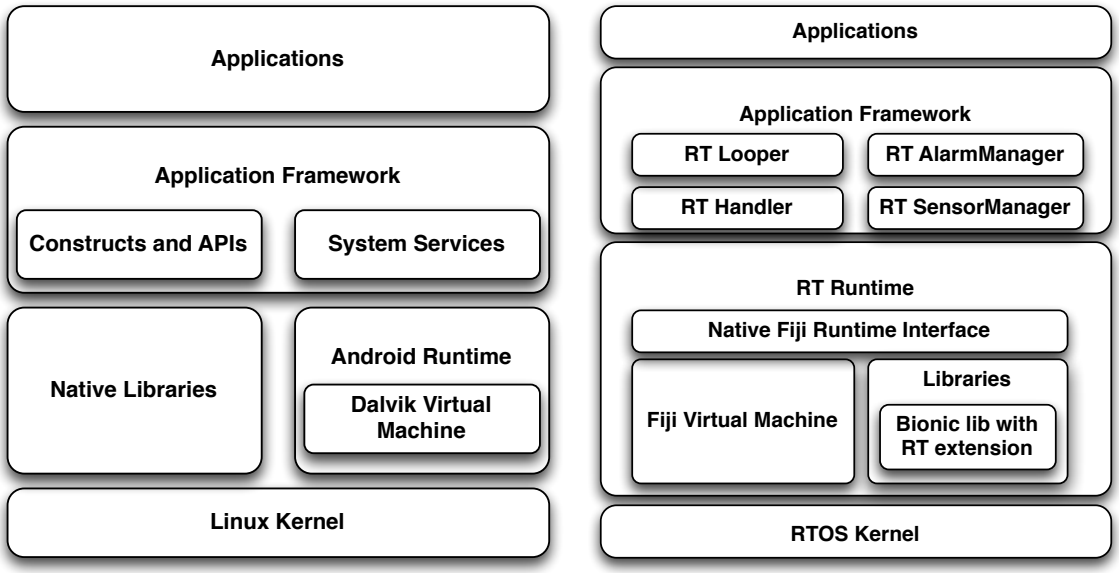
This paper presents our first step to answering that question. We analyze the real-time capabilities of Android and identify limitations. We then propose and implement redesigns of several internal components of Android to provide real-time support. We recognize, however, that Android is a vast system with many components, and that it is difficult to evaluate every aspect of Android. Thus, our goal for this paper is to identify and redesign core components central to Android, in order to support the execution of a *single* real-time application. As the rest of the paper shows, this goal alone has many hard challenges associated and still has broad applicability in utilizing smartphones in real-time domains such as control, medical, and military devices. It is also a prerequisite to supporting multiple real-time applications (*i.e.*, mixed criticality [15, 17]—the ability to execute multiple components with different criticality levels safely).

More concretely, this paper makes the following four contributions. First, we analyze the real-time capabilities of Android and present the result. In addition to the kernel and JVM layers, we examine Android's application framework, which provides programming constructs and system services to applications. We show that Android, due to its heavy reliance on unpredictable message passing mechanisms, does not provide predictable timing guarantees. We also show that system services (understandably) were not designed to support real-time.

Second, we provide an implementation that addresses the limitations discovered in our analysis. We redesign three of the core components in the application framework—a message-passing mechanism (`Looper-Handler`), the timer service (`AlarmManager`), and the sensor architecture (`SensorManager`)—to provide predictable

MobiSys'14, June 16–19, 2014, Bretton Woods, New Hampshire, USA.

http://dx.doi.org/10.1145/2594368.2594381 .



(a) Simplified Android Architecture

(b) RTDroid Architecture

Figure 1: Comparison of Simplified Android and RTDroid Architectures

timing guarantees. We have chosen these components in order to support a class of applications that perform real-time sensing such as fall detection ("man down") applications [18, 29], medical monitors, and control applications.

Third, we report our experience in replacing non-real-time building blocks (Dalvik and Linux) with real-time building blocks. We utilize the Fiji real-time VM [27] as our Java runtime and two real-time OSes (RTLinux [3, 16] and RTEMS [5]) as our kernel options. These building blocks give us a good starting point with sound lower-layer real-time guarantees. In replacing these components, we have encountered practical challenges in modifying both the JVM as well as the kernel layers. We discuss these challenges and our solutions.

Fourth, we demonstrate the real-time capabilities of our redesigns on three different platforms with varying degrees of guarantees: (1) hard real-time on a LEON3 embedded board with the RTEMS RTOS, (2) soft real-time on an x86 PC with RTLinux, and (3) soft real-time on a Nexus S smartphone with the RTLinux patch applied on an Android's version of Linux. In all three platforms, we show that our redesigns provide predictability to applications even under heavily-loaded conditions. As part of this effort, we have implemented, executed, and measured the first real-time Android application: an automatic fall detector deployed on a Nexus S smartphone and on a LEON3 development board. We show that even with hundreds of non-real-time "noise" generating threads, the application shows good timeliness and predictability.

This paper is a continuation of our previous workshop paper [32], which focused on preliminary redesigns of `Looper-Handler` and `AlarmManager`. In this paper, we explore a more comprehensive set of challenges including updated versions of `Looper-Handler` and `AlarmManager`, a redesigned real-time `SensorManager`, and a fall detector that runs on both a smartphone and a LEON3 embedded board. Our prototype presented in this paper enables the execution of a single application while providing real-time guarantees. It serves as a base system on which we will explore multi-application execution in a mixed criticality environment, which is part of our future work discussed in Section 9. Additional performance measures, experiments, raw data, and plotting scripts are publicly available on our website: `http://rtdroid.cse.buffalo.edu`.

The rest of the paper is organized as follows. Section 2 provides an overview of our system, RTDroid. Sections 3, 4, and 5 show the limitations of three components of Android, presents our redesigns and provide a Worst-Case Execution Time (WCET) formalization for each component to algorithmically quantify an upper bound for each individual component. Section 6 reports our experience in replacing non-real-time components with real-time counterparts. Finally, Section 8 discusses our related work and Section 9 discusses our future work and conclusions.

2. OVERVIEW

In this section, we first examine the architecture of Android and discuss the existing work for real-time Android. We then explore the question of how to add real-time capabilities to Android focusing on time, memory, and resource predictability. Lastly, we present an overview of our system, RTDroid.

2.1 Background

Fig. 1a shows a simplified version of the Android architecture. The purpose of the figure is *not* to give a detailed view of Android; instead, we highlight only those components relevant to our discussion in this section.

As Fig. 1a depicts, we can divide Android into roughly three layers below the application layer: (1) the application framework layer, (2) the runtime and libraries layer, and (3) the kernel layer. Android leverages a modified Linux kernel, which does not provide any real-time features such as priority-based preemption of threads, priority inversion avoidance protocols, and priority based resource management. Previous work [13, 19] has also shown that Android's runtime and libraries provide no real-time guarantees and Dalvik's garbage collector can arbitrarily stall application threads regardless of priority, resulting in non-deterministic behavior. Thus, it

is currently well-understood that the bottom layers need real-time support in order to provide a predictable platform.

However, we show in this paper that even with the proper real-time features at the kernel and VM layers, Android cannot provide real-time guarantees. This is due to the fact that the application framework layer does not provide predictability for its core constructs, allowing for arbitrary priority inversion.

Broadly speaking, the application framework layer poses two problems for real-time applications, one rooted in each of its two categories shown in Fig. 1a. The first problem lies in the category shown on the left, *constructs and APIs*, which provides programming constructs [1] and APIs that application developers can use such as `Looper`, `Handler`, and `AsyncTask`. This category poses a problem for real-time applications since the constructs do not provide any time or memory predictability as well as priority awareness. The main issue is that the latency of message delivery in these mechanisms is unpredictable; lower priority threads can unnecessarily prevent higher priority threads from making progress. In Section 3, we discuss this problem in more detail and present our solution. Section 7 demonstrates the problem experimentally.

The second problem occurs in the category shown on the right, *system services*, which provides essential system services. For example, `SensorManager` mediates access to sensors and `AlarmManager` provides system timers. The issue with these system services is that the implementation of the services does not consider real-time guarantees as a requirement. In Sections 4 and 5, we show how two core system services necessary to run a single sensing application, `AlarmManager`, and `SensorManager`, exhibit this general issue and discuss how we redesign these services for real-time support.

2.2 Overview of RTDroid

Our system, RTDroid, aims to add real-time support in Android *as a whole system*, thereby providing the ability to execute a single real-time application that leverages the built-in system services, such as `AlarmManager` and `SensorManager`. This necessitates that our system is predictable in time and memory usage as well as resource management. Our current system design targets a uniprocess environment where only a single user-level process (*i.e.*, the application process) executes. However, we believe that the design is extensible to multi-core and mixed-criticality systems. Such extensions are our future work.

2.2.1 RTDroid Architecture

In order to provide real-time support in all three layers depicted in Fig. 1a, we advocate a clean-slate redesign of Android in Fig 1b. Our redesign starts from the ground up, leveraging an established RTOS (*e.g.*, RT Linux or RTEMS) and an RT JVM (*e.g.*, Fiji VM). Upon this foundation we build *Android compatibility*. In other words, our design provides a faithful illusion to an existing Android application running on our platform that it is executing on Android. This entails providing the same set of Android APIs as well as preserving their semantics for both regular Android applications and real-time applications. For real-time applications, Android compatibility means that developers can use standard Android APIs in addition to a small number of additional APIs our platform provides to support real-time features. These additional APIs provide limited Real-Time Specification for Java (RTSJ) [14] support without scoped memory. This goal of providing Android compatibility makes our architecture unique and different from potential architectures discussed previously in the literature [**?**], where much of the focus is on the kernel and the JVM layers.

[1] By constructs, we mean abstract Java classes provided by the Android application framework. Application developers can extend these abstract classes to leverage advanced functionalities.

2.2.2 Benefits of RTDroid

There are three major benefits of our clean-slate design. First, by using an RTOS and an RT JVM, we can rely on the sound design decisions already made and implemented to support real-time capabilities in these systems. Our RTDroid prototype uses Fiji VM [27], which is designed to support real-time Java programs from the ground up. Fiji VM already provides real-time functionality through static compiler checks, real-time garbage collection [28], synchronization, threading, *etc*. We note, however, that RTDroid's design is VM independent.

The second benefit of our architecture is the flexibility of adjusting the runtime model for different use cases. This is because using an RTOS and an RT JVM provides the freedom to control the runtime model. For example, we can leverage the RTEMS [5] runtime model, where one process is compiled together with the kernel for single application deployment. With this model, an application can fully utilize all the resources of the underlying hardware.Using this runtime model is not currently possible with Android, as Android runs most system services as separate processes. Simply modifying Dalvik or the OS is not enough to augment Android's runtime model; the framework layer itself must be changed.

The third benefit of our architecture is the streamlining of real-time application development. Developers can leverage the rich APIs and libraries that are already implemented and have support for various hardware components. Unlike other mobile OSes, Android excels in supporting a wide variety of hardware with different CPUs, memory capacities, screen sizes, and sensors. Android APIs make it easier to write a single application that can run on different types of hardware. Thus, Android compatibility can reduce the complexity of real-time application development.

2.2.3 Current Scope of Implementation

Our current RTDroid prototype redesigns three core Android components, `Looper` and `Handler`, `AlarmManager` and `SensorManager`. We have chosen these components due to their extensive use in existing Android applications as well as in our target applications. For example, `Handler` and `Looper` are essential to Android applications as they are used implicitly by every application, as we detail in Section 3. `AlarmManager` provides a timer service used by any application that runs periodic tasks; many real-time applications need to run periodic tasks and rely on such a service to trigger their tasks. `SensorManager` provides sensing APIs in Android, which are necessary for our target real-time sensing applications such as fall detectors as well as health monitors.

In addition to redesigning the above three components, we have also ported a subset of other Android programming components necessary to run an application, such as `Service`, `Context`, *etc*. These components do not require a redesign and RTDroid is able to leverage them wholesale. As part of our future work, we plan to increase our coverage to create a more comprehensive system.

2.2.4 Deployment Profiles

RTDroid supports three different types of deployment profiles with varying degrees of guarantees provided by the underlying platform and RTOS kernel. Not all of the deployment profiles currently support hard real-time guarantees due to their use of the RTLinux kernel and closed source drivers as we explain below.

- **Soft Real-time Smartphone:** This profile provides the loosest guarantees due to its reliance on unverified closed source

drivers and a partially preemptible RTLinux kernel as opposed to a *fully* preemtible RTLinux kernel. [2] As we detail in Section 6, the Android patch to Linux is incompatible with the RTLinux patch, which prevents us from putting the kernel into a fully-preemptible mode. As such, it is only suited for soft real-time tasks. However, most applications domains, such as medical device monitoring are soft real-time systems. In this profile, task deadlines can be missed due to jitter from the kernel or blocking from the drivers. Nevertheless, we demonstrate in Section 7 that we can still provide tight latency bounds and predictability even on this profile with RTDroid.

- **Soft Real-time Desktop:** This profile provides stricter guarantees than that of the smartphone as it leverages a fully preemptible RTLinux kernel. In this profile, we can leverage verified-and-certified drivers. However, RTLinux, even in the fully preemptible kernel is not typically used in hard real-time systems. Based on current best practices, this deployment should only be used for soft real-time systems. In this profile, deadlines can be missed due to jitter from the kernel.
- **Hard Real-time Embedded:** By moving away from RTLinux and using a certified RTOS such as RTEMS as well as a development board with certified drivers for its hardware sensors, much stricter guarantees can be provided. No deadlines will be missed due to jitter from the kernel or the drivers.

3. RT LOOPER AND RT HANDLER

In this section as well as the next two sections, we discuss how we add real-time support in the application framework layer of Android. As discussed in Section 2, the first issue that the application framework poses lies in its message-passing constructs. These constructs do not provide any predictability or priority-awareness. We detail this issue in this section and discuss how we address it in RTDroid.

3.1 Background and Challenges

Android provides a set of constructs that facilitate communication between different entities, *e.g.*, threads and processes. There are four such constructs—`Handler`, `Looper`, `Binder`, and `Messenger`. Since any typical Android application uses these constructs, we need to support these constructs properly in a real-time context.

Among these four constructs, `Looper` and `Handler` are the most critical constructs for our target scenario of running a single real-time sensing application. This is because `Binder` and `Messenger` are inter-process communication constructs, while `Looper` and `Handler` are inter-thread communication constructs. Further, `Looper` and `Handler` are used not only explicitly by an application, but also *implicitly* by *all* applications. This is due to the fact that Android's application container, `ActivityThread`, uses `Looper` and `Handler` to control the execution of an application. When an application needs to make transitions between its execution states (*e.g.*, start, stop, resume, *etc.*), `ActivityThread` uses `Looper` and `Handler` to signal necessary actions.

Fig. 2 shows how `Looper` and `Handler` work. `Looper` is a per-thread message loop that Android's application framework implements. Its job is to maintain a message queue and dispatch each message to the corresponding `Handler` that can process the message. The developer of the application provides the processing logic for a message by implementing `Handler`'s `handleMessage()`. A `Handler` instance is shared between two threads to send and receive messages.

[2]With a fully preemptible kernel, all parts of the kernel become preemptible by a high priority thread.

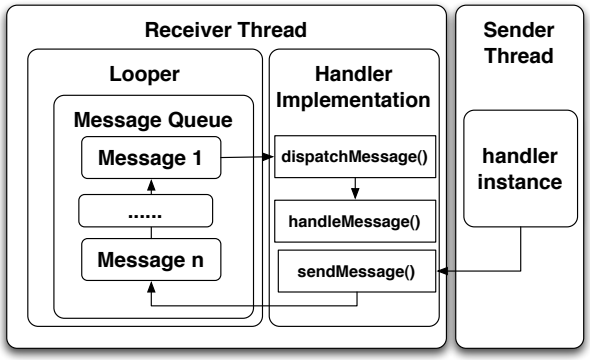


Figure 2: The Use of `Looper` and `Handler`

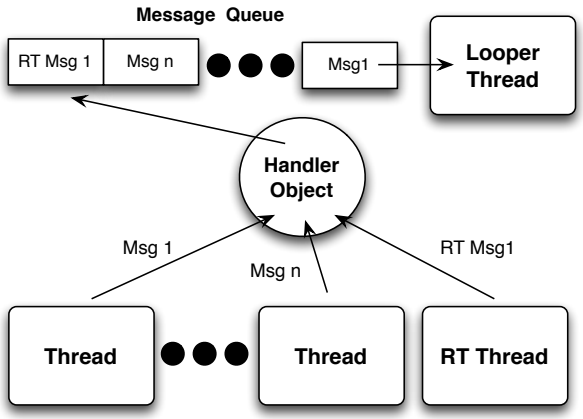


Figure 3: The thread in which the looper executes processes the messages sent through this handler object in the order in which they are received.

The `Looper` and `Handler` mechanism raises a question for real-time applications when there are multiple threads with different priorities sending messages simultaneously. In Android, there are two ways that `Looper` and `Handler` process messages. By default, they process messages in the order in which they were received. Additionally, a sending thread can specify a message processing time, in which case `Looper` and `Handler` will process the message at the specified time. In both cases, however, the processing of a message is done regardless of the priority of the sending thread or the receiving thread. Consider if multiple user-defined threads send messages to another thread. If a real-time (*i.e.*, high-priority) thread sends a message through a `Handler`, its message will not be processed until the `Looper` dispatches every other message prior to its message in the queue regardless of the sender's priority as seen in Fig. 3. The situation is exacerbated by the fact that Android can re-arrange messages in a message queue if there are messages with specific processing times. For example, suppose that there are a number of messages sent by non-real-time (*i.e.*, low-priority) threads in a queue received before a message sent by a real-time thread. While processing those messages, any number of low-priority threads can send messages with specific times. If those times fall within the

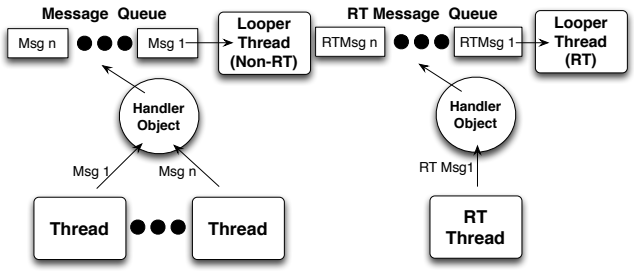


Figure 4: An Example of `Looper` and `Handler` in RTDroid. Each message has a priority and is stored in a priority queue. Processing of messages is also done by priority. The example shows one high-priority thread and multiple non-real-time threads.

processing time window for non-real-time messages, the real-time message will get delayed further by non-real-time messages.

3.2 Redesign

To mitigate the issues mentioned, we redesign `Looper` and `Handler` in two ways. First, we assign a priority to each message sent by a thread. We currently support two policies for priority assignment. These policies are *priority inheritance*, where a message inherits its sender's priority, and *priority inheritance + specified* where a sender can specify the message's priority in relation to other messages it sends.

Second, we create multiple priority queues to store incoming messages according to their priorities. We then associate one `Looper` and `Handler` for each queue to process each message according to its priority. Fig. 4 shows our new implementation for `Looper` and `Handler`. Since we now process each message according to its sender's priority, messages sent by lower priority threads do not delay the messages sent by higher priority threads. For memory predictability, queues can be statically configured in size.

3.3 Worst-Case Execution Time

We now formally show how our new design provides predictability. The worst-case execution time is the best metric for this purpose as it gives the upper bound on execution time. To understand the worst-case execution characteristics of the Real-time `Looper` and `Handler` we must reason about how the constructs process a series of messages and execute each message's callback function. We define T_i^j to be the i^{th} message issued by the application from a thread with priority *j*. The messages are passed into a real-time `Looper` that has the same priority as the messages and then they are enqueued in a `MessageQueue`. The time cost for handling the i^{th} message in priority level `j` is shown as S_i^j in Equation (1):

$$S_i^j = \sum_{l=0}^{i} (h_l^j + deq(T_l^j)), \tag{1}$$

Where h_l^j is the cost of time to handle T_l^j and deq(T_l^j) is the cost of dequeuing from the message queue.

To reason about the worst-case execution time for a message *m*, we must first calculate the processing time for all messages that have priorities greater than or equal to the priority of message `m`, shown in Equation (2):

$$phase_0(T_i^j) = \sum_{p>j} S_{last}^p + S_i^j. \tag{2}$$

Where *last* is the last message in the message queue with priority *p* that is greater than *j*. Since the system also handles new incoming messages, which may have a priority greater than or equal [3] to that of *m*, we must also define the system in terms of a message arrival rate *R* for a given priority *p*.

We divide the amount of time for the system to handle *m* into a number of phases. During $phase_0$, the system handles all of the messages in the priority queue which are greater than or equal to the priority of *m* as shown in Equation 2. While handling the message in the current phase, new messages arrive at a given rate per priority level, the system must then handle each of the new messages with priority greater than or equal to *m* before handling message *m*.

In order to quantify the number of messages in each priority queue, we define a sending rate for each group of clients with priority *p*, R_p. When n $\geq$ 1, then worst-case handling time is integrating all of the handling times for messages that are greater than or equal to the priority of message `m`, as shown in Equation (3):

$$phase_n(T_i^j) = \sum_{p \geq j} \sum_{i=0}^{phase_{n-1}(T_i^j) * R_p} \big(h_i^p + deq(T_i^j) + enq(T_i^j)\big). \tag{3}$$

Where enq(T_i^j) is the cost of enqueuing in the message queue.

The LHS represents the upper bound of the time cost for message handling for $\texttt{phase}_n$, the RHS represents the total time cost for handling all messages that arrive during $\texttt{phase}_{n-1}$; The outer summation is the time to handle each priority level and the inner summation is the integration of the time to handle all of the same priority messages that have arrived in the phase_{n-1}. $phase_{n-1}(T_i^j)$ represents the time spent in previous phase, and when multiplied by the rate R_p gives the number of messages currently in each priority based queue. The recursion ends when $phase_n$ is smaller than the time unit of R_p. Thus, the summation of all phases is the actual worst-case execution time for handling message, *m* as shown in Equation (4):

$$\begin{aligned} WCET(T_i^j) = {} & phase_0(T_i^j) + phase_1(T_i^j) \\ & + ... + phase_{n-1}(T_i^j) + phase_n(T_i^j). \end{aligned} \tag{4}$$

Notice, the system is only well defined (*i.e.* able to process messages with real-time guarantees) if the worst-case execution time for each message is less than the deadline for processing that message relative to its arrival time and if phase_n is less than phase_{n-1}.

4. RT ALARM MANAGER

As mentioned in Section 2, the second issue that Android's application framework layer poses for real-time support is that system services do not provide real-time guarantees. Since Android mediates all access to its core system functionalities through a set of system services, it is critical to provide real-time guarantees in the system services. Just to name a few, these services include `SensorManager` that mediates all sensor access and data acquisition; and `AlarmManager` that provides a timer service.

The presence of these system services raises two questions. First, in our target scenario of running a single real-time application, there is no need to run system services as separate processes; rather it is more favorable to run the application and the system services as a single process to improve the overall efficiency of the system. Then the question is how to redesign the system service architecture in our platform in order to avoid creating separate processes

[3] Although our `Looper` and `Handler` uses a FIFO priority queue, we are abstracting the complexities of the data-structure algorithm, such as queuing and dequeuing costs, in the calculation and thus creating a generalized equation applicable to all our RT redesigns.

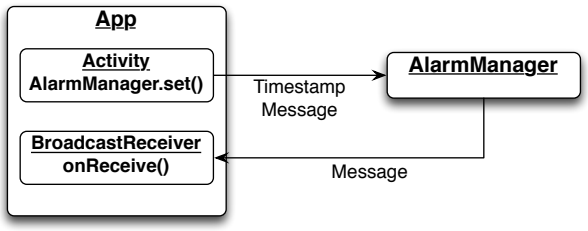


Figure 5: An Example Flow of `AlarmManager`. An application uses `AlarmManager.set()` to register an alarm. When the alarm triggers, the `AlarmManager` sends a message back to the application, and the application's callback (`BroadcastReceiver.onReceive()` in the example) gets executed.

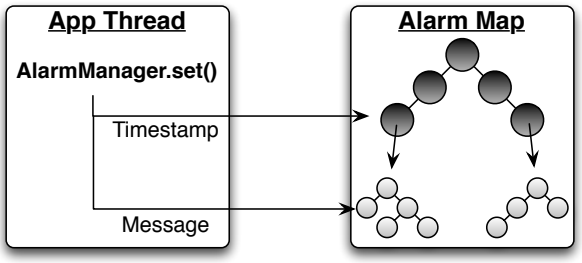


Figure 6: The Implementation of Alarm Execution on RTDroid. The tree colored black at the top maintains timestamps. The trees colored gray are per-timestamp trees maintaining actual alarm messages to be delivered.

while preserving the underlying behavior of Android. Second, as we show in this section and the next section, the internals of these system services do not consider real-time support as a design requirement.

To answer these two questions, we redesign two of the system services—`AlarmManager` and `SensorManager`. In this section we first show how we redesign `AlarmManager` to provide real-time guarantees. In the next section, we discuss our `SensorManager` redesign.

4.1 Background and Challenges

`AlarmManager` receives timer registration requests from applications and sends "timer triggered" messages to these applications when its timer fires. Since real-time applications frequently rely on periodic and sporadic tasks, it is important to provide real-time guarantees in `AlarmManager`.

Fig. 5 shows how `AlarmManager` works, including alarm registration and alarm delivery. An IPC call, with a message [4] and its execution time, is made to the `AlarmManager` every time an application registers an alarm. When the the alarm triggers at the specified time, the `AlarmManager` sends a message back to the application, and the associated callback is executed. The issue with `AlarmManager` is that it provides no guarantee on when or in what order alarm messages are delivered, hence does not provide any timing guarantee or priority-awareness.

4.2 Redesign

We redesign both alarm registration and delivery mechanisms to support predictable alarm delivery. For alarm registration, we use red-black trees to maintain alarms as shown in Fig. 6. This means that we can make the registration process predictable based on the complexity of red-black tree operations, *i.e.*, the longest path of a tree is no longer than twice the shortest path of the tree. We use one red-black tree for storing timestamps and pointers to per-timestamp red-black trees. Per-timestamp trees are leveraged to order alarms with the same timestamp by their sender's priority. Thus, our alarm registration process is essentially one insert operation to the timestamp tree and another insert operation to a per-timestamp tree. By organizing the alarms based on senders' priorities, we guarantee that an alarm message for a low priority thread does not delay an alarm message for a high priority thread. Expired alarms are discarded. Note that this ensures that low priority threads whose alarm registration rate exceeds the alarm delivery capacity of the system cannot prevent a high priority alarm from being triggered.

For alarm delivery, we create an `AlarmManager` thread and assign the highest priority for timely delivery of alarm messages. This thread replaces the original multi-process message passing architecture of Android. It wakes up whenever an application inserts a new alarm into our red-black trees, then it schedules a new thread at the specified time for the alarm. We associate the application's callback for the alarm message with this new thread. For precise execution timing of this callback thread, we implement Asynchronous Event Handlers (AEH) that Real-Time Specification for Java (RTSJ) [14] specifies the interface for.

We have implemented two versions for AEH. The first is a per-thread AEH implementation used in our workshop paper [32], which creates one thread per handler to process a given event type. This simple mechanism is efficient in handling low numbers of events, but can create memory and processing pressure due to large number of handling threads if a large number of events occur within the same time period. Although most Android applications do not register alarms at a frequency that would cause problems, our system must be resilient to such behavior nonetheless.

The second mechanism leverages a thread pool with a statically configured number of threads, which reduces the number of threads that we need to create. Our implementation is based on Kim *et al.*'s proposed model [20] and is similar to how the jRate [10] implements RTSJ's AEH. The benefit of this implementation is a hard, statically known limit on the number of threads to handle asynchronous events. There is lower memory usage due to less threads being created and the output is deterministic with a well-known, predictable behavior [10].

4.3 Worst-Case Execution Time

The worst-case execution scenario for `AlarmManager` is similar to that discussed for the `Looper` and `Handler` in Section. 3.3. The upper bound of delivery and execution of an alarm *a* consists of 1) the delivery and execution of all alarms that have been registered with priority greater or equal to that of *a*, 2) the delivery and execution of all newly registered alarms with priority greater or equal to *a* based on a per priority rate of alarm delivery and registration. The equation of WCET for `AlarmManager` is the same pattern as shown in Equation (1), (2), (3), (4), but couched in terms of alarm processing instead of message delivery.

- T_i^j represents the i^{th} alarm registered by application with priority *j*.
- S_i^j represents the time cost for handling the i^{th} alarm in priority level *j*.

[4] This message is associated with a callback for the application which gets executed when the message is delivered.

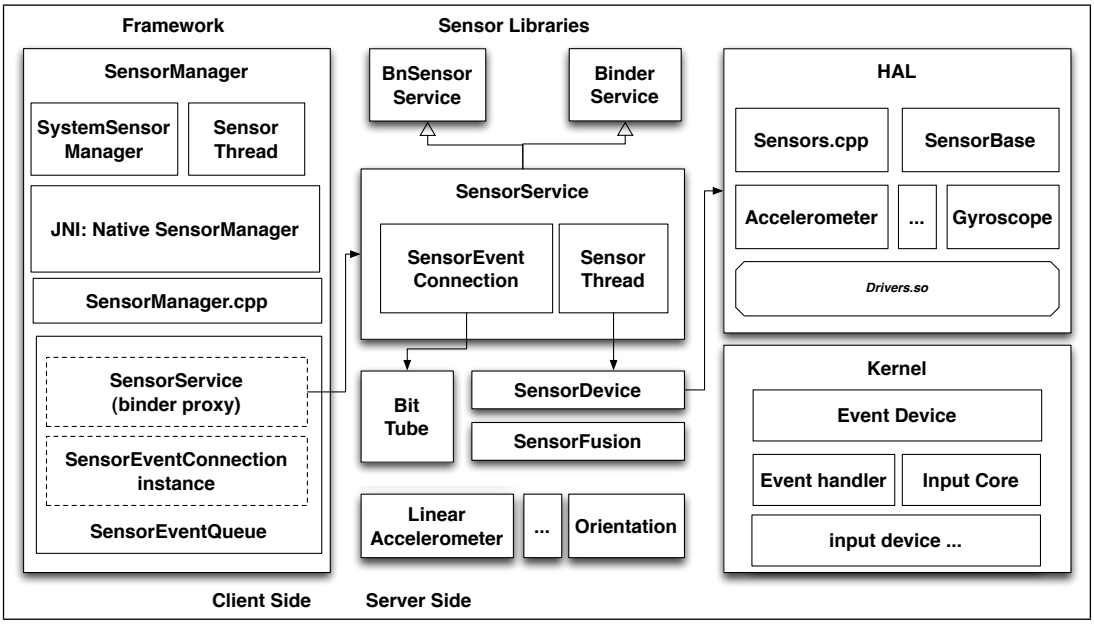


Figure 7: Android Sensor Architecture

- h_i^j is the cost of time to execute the alarm T_i^j.
- *last* is the last alarm in priority p that is greater than *j*.
- enq(T_i^j) is the cost of alarm registration.
- deq(T_i^j) is the cost of alarm delivery.

5. RT SENSOR ARCHITECTURE

Another system service we redesign in RTDroid is `SensorManager`. Modern mobile devices are equipped with many sensors such as accelerometers, gyroscopes, *etc*. Android, mainly through its `SensorManager`, provides a set of APIs to acquire sensor data. This section examines the current sensor architecture of Android and presents our new design for real-time support.

5.1 Background and Challenges

On Android, sensors are broadly classified into two categories. The first category is *hardware* sensors, which are the sensors that have a corresponding hardware device. For example, accelerometer and gyroscope belong to this category. The second category is *software* sensors, which are "virtual" sensors that exist purely in software. Android fuses different hardware sensor events to provide software sensor events. For example, Android provides an orientation sensor in software. On Nexus S, Android 4.2 has 6 hardware sensors and 7 software sensors.

These sensors are available to applications through `SensorManager` APIs. An application registers sensor event listeners through the provided APIs. These listeners provide the application's callbacks that the Android framework calls whenever there is any requested sensor event available. When registering a listener, an application can also specify its desired delivery rate. The Android framework uses this as a hint when delivering sensor events.

Internally, there are four layers involved in the overall sensor architecture—the kernel, HAL, `SensorService`, and `SensorManager`. Fig. 7 shows a simplified architecture.

1. **Kernel:** The main job of the kernel layer is to pull hardware sensor events and populate the Linux `/dev` file system to make the events accessible from the user space. Each sensor hooks to the circuit board through an I^2C bus and registers itself as an *input device*.

2. **HAL:** The HAL layer provides sensor hardware abstractions by defining a common interface for each hardware sensor type. Hardware vendors provide actual implementations underneath.

3. `SensorService`**:** `SensorService` converts raw sensor data to more meaningful data using application-friendly data structures. This involves three steps. First, `SensorService` polls the Linux `/dev` file system to read raw sensor input events. Second, it composites both hardware and software sensor events from the raw sensor input events. For hardware sensors, it just reformats the data; for software sensors, it combines different sources to calculate software sensor events via sensor fusion. Finally, it writes each sensor event to the `SensorEventQueue` via `SensorEventConnection`.

4. **Framework Layer:** `SensorManager` delivers the sensor events by reading the data from `SensorEventQueue` and invoking the registered application listeners to deliver sensor events.

There are two issues that the current architecture has in providing predictable sensing. First, there is no priority support in the sensor event delivery mechanism since all sensor events go through the same `SensorEventQueue`. When there are multiple threads with different priorities, the event delivery of lower-priority threads can delay the event delivery of higher-priority threads. Second, the primary event delivery mechanisms poll and buffer at the boundary of different layers (*e.g.*, between the kernel and `SensorService` and between `SensorService` and `SensorManager`) by use of message passing constructs. Android does not provide any guarantee on how long it takes to deliver events through these mechanisms.

5.2 Redesign

We redesign the sensor architecture for RTDroid to address the two issues mentioned above. Our design is inspired by event pro-

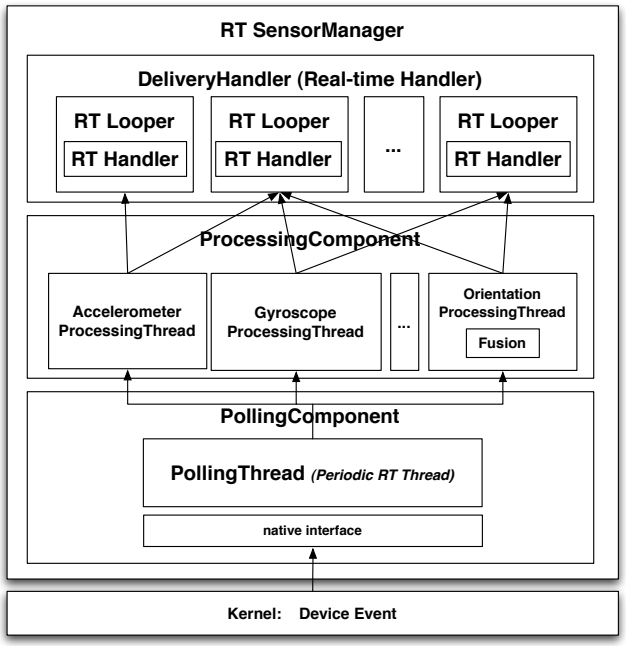


Figure 8: RTDroid Sensor Architecture

cessing architectures used for Web servers [26, 31]. We first describe the architecture and discuss how we address the two problems with our new architecture.

As shown in Fig. 8, there are multiple threads specialized for different tasks. At the bottom, there is a *polling thread* that periodically reads raw sensor data out of the kernel. This polling thread communicates with multiple *processing threads*. We allocate one thread per sensor type as shown in Fig. 8, *e.g.*, one thread for accelerometer, one thread for gyroscope, and one thread for the orientation sensor. The main job of these processing threads is to perform raw sensor data processing for each sensor type. For example, a processing thread for a hardware sensor reformats raw sensor data using an application-friendly format, and a processing thread for a software sensor performs sensor fusion. Once the raw sensor data is properly processed, each processing thread notifies the *delivery thread* whose job is to create a new thread that executes the sensor event listener callback registered by an application thread. To provide predictable delivery, we use notification, not polling, for our event delivery except in the boundary between the kernel and the polling thread. We provide additional predictability through our priority inheritance mechanism described next.

We address the two issues mentioned earlier by priority inheritance. When an application thread of priority p registers a listener for a sensor, say, gyroscope, then the processing thread for gyroscope inherits the same priority p. If there are multiple application threads that register for the same gyroscope, then the gyroscope processing thread inherits the priority of the highest-priority application thread. In addition, when the delivery thread creates a new thread that executes a sensor event listener callback, this new thread also inherits the original priority p of the application thread. We assign the highest priority available in the system to the polling thread to ensure precise timing for data pulling.

This combined use of event-based processing threads and priority inheritance has two implications. First, when an application thread registers a listener for a sensor, we effectively create a new, isolated event delivery path from the polling thread to the listener. Second, this newly created path inherits the priority of the original application thread. This means that we assign the priority of the application thread to the *whole event delivery path*.

5.3 Worst-Case Execution Time

The worst-case execution scenario for `SensorManager` is slightly different than what we have discussed in Section. 3.3 and 4.3. The upper bound for delivery of the sensor event to a sensor listener, *l*, consists of three parts: (1) the time cost of the system delivery the sensor event to all sensor listeners that registered a listener that are greater or equal to the priority of *l*, (2) recursively integrate the time cost for register and deliver of the sensor data for the new higher-priority listener arriving at a per priority rate, and (3) the time cost for polling the data from each sensor kernel module. The WCET equation for `SensorManager` is in the same fashion as previously defined in Equation (1), (2), (3), (4), and includes the sensor data polling cost as shown in in Equation. 5, 6:

$$phase_0(T_i^j) = \sum_{p \geq j} P_j(sensor_e) + \sum_{p > j} S_{last}^p + S_i^j \tag{5}$$

$$phase_n(T_i^j) = \sum_{p \geq j} \sum_{i=0}^{phase_{n-1}(T_i^j) * R_p} \big(h_i^p + deq(T_i^j) + enq(T_i^j)\big). \tag{6}$$

- T_i^j represents the i^{th} sensor listener in application with priority *j*.
- S_i^j represent the time cost to execute the i^{th} callback of sensor listener in priority level *j* .
- h_i^j is the amount of time to execute the callback of sensor listener of T_i^j.
- deq(T_i^j) is the cost of listener registration.
- *last* is the sensor listener in priority *p* that is greater than *j*.
- $P_j(sensor_e)$ is the cost of sensor data polling.

6. REAL-TIME BUILDING BLOCKS

In this section, we report our experience in replacing non-real-time building blocks (Dalvik and Linux) with off-the-shelf real-time counterparts (Fiji VM and RTOSes). As mentioned earlier, we support three deployment profiles, an x86 PC environment, an embedded environment with a LEON3 development board, and an ARM-based smartphone environment with a Nexus S smartphone. The x86 and the LEON3 environments do not require any more than replacing the non-real-time kernel with either real-time Linux kernel (by applying an RT-Preempt patch, *i.e.*, RTLinux) or the real-time RTEMS kernel. The same strategy, however, does not work for the smartphone environment because Android has introduced extensive changes in the kernel that are not compatible with RTLinux patches. Thus, we first briefly describe our x86 and LEON3 environments. We then report our experience with the smartphone environment in detail.

6.1 x86 PC and LEON3

For the x86 environment, we apply an RTLinux patch (patch-3.4.45-rt60) to Linux 3.4.45, and use Fiji as the real-time VM. Fiji already runs on RTLinux, thus it did not require any additional effort. This configuration represents our soft real-time deployment. Tighter bounds are provided as RTLinux makes the kernel fully preemptible. Similarly, we can introspect the drivers on the ma-

chine to guarantee their timeliness or leverage off-the-shelf drivers that have already been vetted.

To create the LEON3 environment, we use a LEON3 embedded board, GR-XC6S-LX75, manufactured by Gaisler. We then use RTEMS as the real-time kernel and Fiji as the real-time VM. RTEMS has native support for LEON3 and Fiji already supports RTEMS. This configuration represents our hard real-time embedded board deployment, avoiding the issues that plague RTLinux and closed source drivers. The LEON3 manufacturers provide drivers that have previously been certified for automotive, aerospace, and civilian aviation.

In order to test the `SensorManager` on the LEON3 system, we have designed and implemented an accelerometer daughter board as well as the associated RTEMS compliant driver.

6.2 Nexus S Smartphone

Unfortunately, the same approach is not adequate for executing real-time applications on an Android phone. This is mainly due to the incompatibilities between Android and the real-time building blocks in the kernel layer as well as in Android's C library, Bionic. The following are the main challenges to integration.

6.2.1 Bionic

Android does not utilize glibc as the core C library, instead it uses its own library called Bionic [12]. Bionic is a significantly simplified, optimized, light-weight C library specifically design for resource constrained devices with low frequency CPUs and limited main memory. Its architectural targets are only ARM and x86.

Bionic becomes a problem when replacing Dalvik with Fiji; this is because it does not support the real-time extensions for Pthreads and mutexes, which are required by Fiji (or any other real-time Java VM). In addition, it is not POSIX-compliant. Thus, we have modified Bionic to include all necessary POSIX compliant real-time interfaces. This includes all the real-time extensions for Pthreads and mutexes.

6.2.2 Incompatible Kernel Patches

Android has introduced a significant amount of changes specializing the Linux kernel for Android, *e.g.*, low memory killer, wakelock, binder, logger, *etc*. Due to these changes, automatic patching of an Android kernel with an RTLinux patch is not possible, requiring a manually applied RTLinux patch.

Even after manual patching, however, we have discovered that we are still not able to get a fully-preemptible kernel which can provide tighter latency bounds. The reason is simply that Android's changes are not designed with full preemption in mind. We are currently investigating this issue and it is likely that this is an engineering task. Nevertheless, we are not aware of any report of a fully-preemptible Android kernel.

6.2.3 Non-Real-Time Kernel Features

During our initial testing and experimentation, we have discovered that there are two kernel features that are not real-time friendly. They are the *out of memory killer* (OOM killer) [2] and *CPUFreq governors* [11]. The OOM killer is triggered when there is not enough space for memory allocation. It scans all pages for each process to verify if the system is truly out of memory. It then selects one process and kills it. We have found out that this causes other threads and processes to stop for an arbitrary long time, creating unpredictable spikes in latency. For our target scenario of running a single real-time application, the OOM killer is not only unnecessary, but a source of missed real-time task deadlines. Memory management is provided by Fiji VM's Schism, which is a real-time, fragmentation tolerant GC [28]. It is therefore, critical to disable OOM killer.

CPUFreq governors offer dynamic CPU frequency scaling by changing the frequency scaling policies. Android uses this to balance between phone performance and battery usage. The problem is that when a CPUFreq governor changes the frequency, it affects the execution time of all running threads, again introducing jitter in the system. Moreover, frequency scaling is not taken into consideration when scheduling threads. The result is missed task deadlines and unpredictable spikes in latency. Although not the focus of our experiments, we note that real-time scheduling that takes voltage scaling into consideration has been vetted for hardware architectures with specialized mechanisms for predictability [9].

In our experiments, we show the behavior of RTDroid with two governors—the "ondemand" governor, which dynamically changes the CPU frequency depending on the current usage, and the "performance" governor, which sets the CPU frequency to the highest frequency possible. We leave it as our future work to handle dynamic frequency scaling. For example, we can apply an existing method for worse case execution time analysis [22] to validate the hardware and leverage this timing analysis to modify the kernel and VM schedulers appropriately.

7. EXPERIMENTAL RESULTS

To measure and validate our prototype of RTDroid, we tested our implementation on three system level configurations, each of which represents one of our target deployments discussed in Section 2.2.4. The first configuration utilizes an Intel Core 2 Duo 1.86 GHz Processor with 2GB of RAM. For precise timing measurements, we disabled one of the cores prior to running the experiments. The second configuration is a Nexus S phone equipped with a 1 GHz Cortex-A8 and 512 MB RAM along with 16GB of internal storage and an accelerometer, gyro, proximity, and compass sensors running Android OS v4.1.2 (Jelly Bean) patched with RT Linux v.3.0.50. For the third configuration we leveraged a GR-XC6S-LX75 LEON3 development board running RTEMS version 4.9.6. The board's Xilinx Spartan 6 Family FPGA was flashed with a modified LEON3 [5] configuration running at 50Mhz. The development board has an 8MB flash PROM and 128MB of PC133 SDRAM. We present observed, end-to-end worst-case execution times as it is difficult to provide the latency breakdown for the whole system without specialized timing hardware. We therefore focus on showing the timeliness of our system on a series of stress tests. We couple the worst observed latency/processing time for each experiment with the algorithmic characterization of each component, individually presented in Sections 3.3, 4.3, and 5.3.

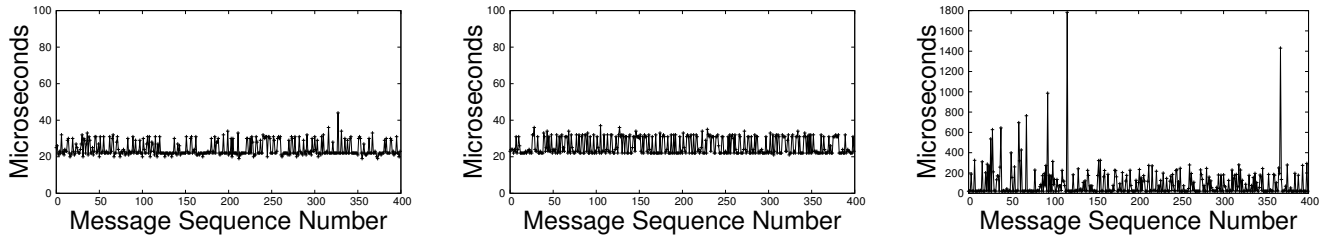
We have designed and developed a daughter board with interface circuitry based on an MMA8452Q triple axis accelerometer. We have developed an RTEMS driver for the accelerometer and integrated it into our RTEMS build.

Due to space constraints, we only show a subset of our experimental results. All of our results are available through our website: `http://rtdroid.cse.buffalo.edu`.

7.1 RT Looper and RT Handler

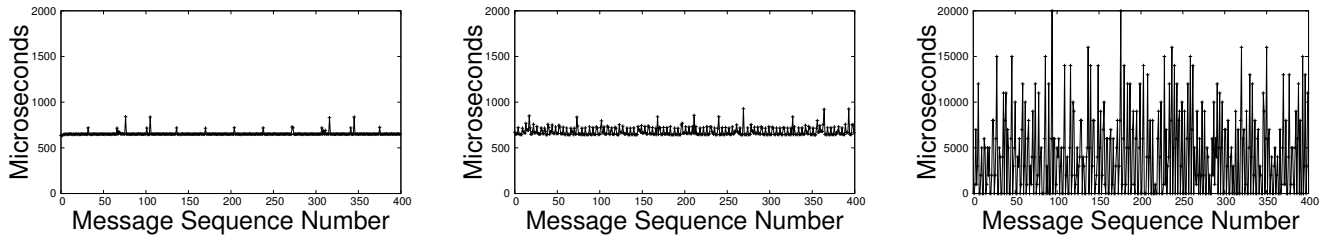
To measure the effectiveness of our prototype, we have constructed an experiment that leveraged `RT Looper` and `RT Handler`. Our microbenchmark creates one real-time task with a 100 ms period that sends a high-priority message. To measure the predictability of the system, we calculate the latency of processing the

[5]The LEON3 core was reconfigured to allow access to a specific I^2C bus so that the accelerometer could be connected to the board.



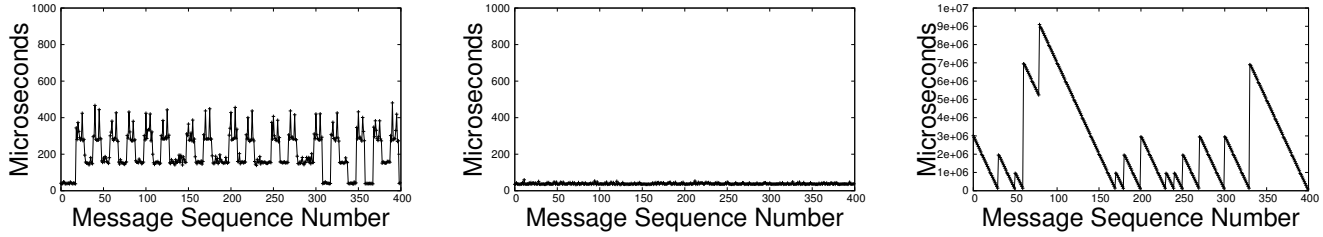
(a) RTDroid: 30 low-priority threads (b) RTDroid: 300 low-priority threads (c) Android: 30 low-priority threads

Figure 9: The observed raw latency of `Looper` and `Handler` on x86. Please note graph (c) has a different Y-axis.



(a) RTDroid: 5 low-priority threads (b) RTDroid: 30 low-priority threads (c) Android: 5 low-priority threads

Figure 10: The observed raw latency of `Looper` and `Handler` on LEON3 Please note graph (c) has a different Y-axis.



(a) RTDroid: With Ondemand Governor (b) RTDroid: With Performance Governor (c) Android: Base System

Figure 11: The observed raw latency of `Looper` and `Handler` on Nexus S. Please note graph (c) has a different Y-axis

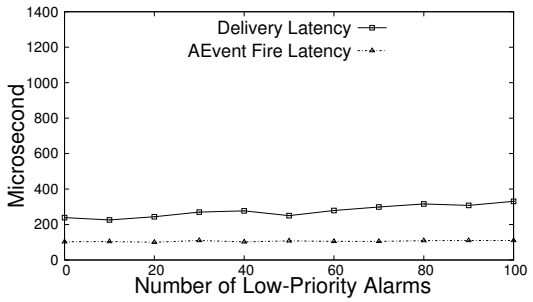
message, determined by two timestamps. The first timestamp is taken in the real-time thread prior to sending the message. This timestamp is the data encoded within the message. The second timestamp is taken within the `RT Handler` responsible for processing this message after the message has been received and the appropriate callback invoked. The difference between these two timestamps is the message's latency.

In addition, the experiments include a number of low-priority threads which also leverage `RT Looper` and `RT Handler`. These threads have a period of 10 ms and send 10 messages during each period. To compare the `Looper` and `Handler` designs between RTDroid and Android, we have ported the relevant portion of Android's application framework, including `Looper` and `Handler`, so we can compile and run our benchmark application on x86. Thus, on Android, all threads, regardless of their priorities, use the same `Looper` and `Handler`—this is the default behavior. On RTDroid, each thread uses a different pair of `RT Looper` and `RT Handler` according to its priority—this is opaque to the application developer and handled automatically by the system.

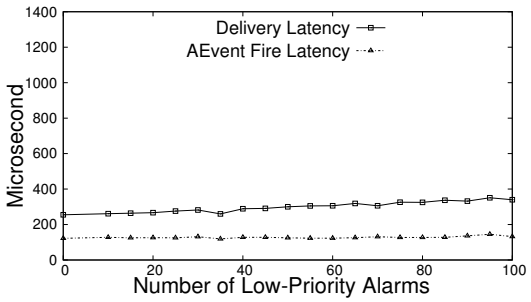
To measure the predictability of our constructs under a loaded system, we increase the number of low-priority threads. We have executed each experiment for 40 seconds, corresponding to 400 releases of the high-priority message, and have a hard stop at 50 seconds. We measure latency only for the high-priority messages and scale the number of low-priority threads up to the point where the total number of messages sent by the low-priority threads exceeds the ability to process those messages within the 40 second execution window. On both Intel Core 2 Duo and Nexus S, we have varied the number of low-priority threads in increments of 10 from 0-300. Considering memory and other limitations of our resource constrained embedded board, we have run the experiments increasing the low priority threads in increments of 5 from 5-30 when running on the LEON3 board.

Fig. 9 and Fig. 10 demonstrates the consistent latency of our `RT Looper` and `RT Handler` implementation. On the desktop, we observe most of the latency for messaging is between 22 μs and 50 μs with any number of threads, and the variance is around 20 μs from the lowest to the highest latency in any given run. The worst observed latency variance is 26 μs. This degree of variance on the system is attributed to context switch costs and scheduling queue contention. On the LEON3 development board, the result shows a similar pattern. In contrast, the huge variance of Android on both platforms clearly indicate its inability to provide real-time guarantees.

Fig. 11 shows the results on Nexus S. We run two series of experiments, one with the ondemand governor and the other one with the performance governor. It was observed that the latency of the tests with the ondemand governor decreases with an increasing number

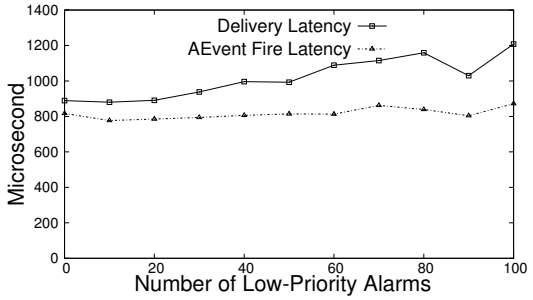


(a) RTDroid: Per Thread AEH

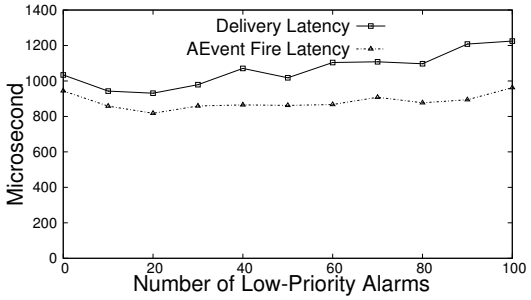


(b) RTDroid: Thread Pool AEH

Figure 12: RTSJ's AEH implementation - Per Thread vs Thread Pool on x86.



(a) RTDroid: Per Thread AEH



(b) RTDroid: Thread Pool AEH

Figure 13: RTSJ's AEH implementation - Per Thread vs Thread Pool on Nexus S.

of low-priority threads. This is due to the extra load on the system, which results in the CPU's frequency being increased by the ondemand governor. The tests with the performance governor show a consistent latency in any given run, since the CPU frequency does not change. On the other hand, the latency variation from Android is several orders of magnitude greater than that of RTDroid as shown in Fig. 11a.

7.2 RT AlarmManager

Measuring the performance of the `RT AlarmManager` was done with an experiment consisting of scheduling of a single high-priority alarm at the current system time + 40 ms, while increasing the number of low-priority alarms scheduled at the exact same time. We measure two types of latency for the experiment: 1) the entire latency of the alarm delivery (*Delivery latency*), which is the difference between the scheduled time and actual execution time of the high-priority alarm, and 2) the latency of the asynchronous event fire (*AEvent fire latency*), which is the difference between the scheduled time and the actual firing time by the `AlarmManager`. The difference between the two types of latency shows how long it takes for the system to deliver an alarm from the `AlarmManager` to the application. We run the experiment on all three platforms. These results show the timing and latency of the alarm execution process and indicate that the `RT AlarmManager` is efficient at prioritizing high-priority alarms and scheduling them at their specified time.

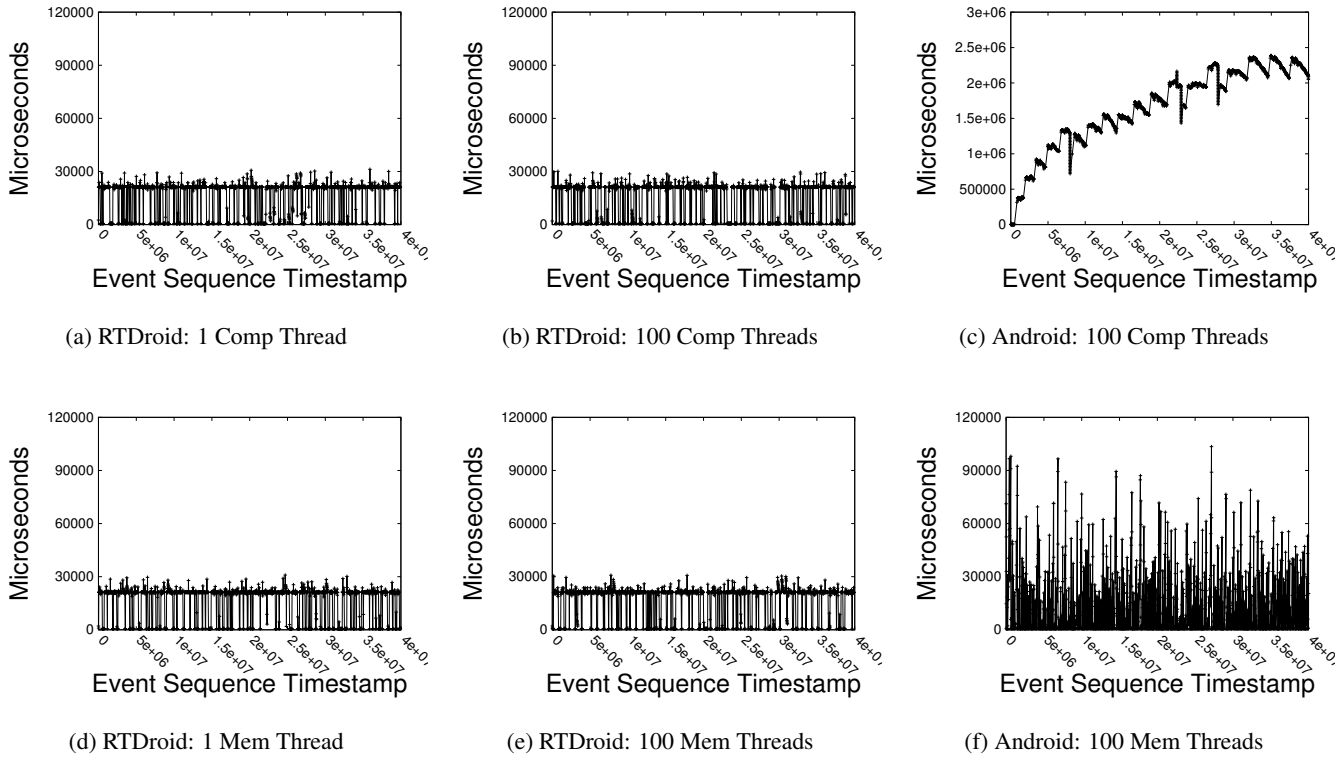
As mentioned in Section 4, we have implemented two techniques for alarm management in `RT AlarmManager`—one with a per-thread AEH implementation used in our previous workshop paper [32] and another implemented with a thread pool. We show the predictability of RTDroid with each technique by using threads ranging from 5-100 and a thread granularity of 10. To induce queueing in the thread pool implementation, only 3 worker threads are allocated for the thread pool.

7.2.1 *x86 Desktop*

Fig. 12 shows the results of the per-thread AEH and the thread pool AEH experiments running on RTLinux. The latency of the entire alarm delivery for per-thread AEH on the x86 is bounded from 220 μs to 331 μs with a 32 μs standard deviation. The *Asynchronous event fire* latency is consistently around 105 μs. The thread pool implementation exhibits a slightly slower performance with the alarm delivery bounded from 255 μs to 355 μs with a 29 μs standard deviation. This small performance drop is expected and caused by alarm queuing in the thread pool itself.

7.2.2 *Nexus S Smartphone*

Fig. 13 demonstrates the results with the same experimental scenario on the Nexus S smartphone. It shows the same pattern as the x86 does, but with a lager value. This is not surprising considering the hardware difference between X86 and smartphone in terms of the type and frequency of their CPU and available memory. On av-



(a) RTDroid: 1 Comp Thread

(b) RTDroid: 100 Comp Threads

(c) Android: 100 Comp Threads

(d) RTDroid: 1 Mem Thread

(e) RTDroid: 100 Mem Threads

(f) Android: 100 Mem Threads

Figure 14: Memory and Computation stress test for the Fall Detection Application on Nexus S.

erage, the thread pool shows less than 1 ms latency and the worst observed case is 1.2 ms for both of the the per-thread AEH and thread pool implementations, with 108 μs and 92 μs deviation.

7.2.3 LEON3 Embedded Board

We have conducted a similar set of experiments on LEON3. Our results, however, show little difference from our x86 and Nexus S results. Due to this reason and space considerations, we do not include the graphs. Overall, for both AEH implementations across platforms, our experiments show that high-priority threads execute in a deterministic fashion and with tight bounds. This is irrespective of the number of low-priority threads that exist in the system.

7.3 Real-Time Fall Detector

To validate the predictability of our sensor architecture in data delivery, we have created a soft real-time fall detection application that leverages our `SensorManager` outlined in Section 5. We designed two experiments with two different types of workloads: (1) a memory intensive load and (2) a computation intensive load. The memory intensive experiment creates a varying number of non-real-time priority threads that each allocate a 2.5 MB integer array storing integer objects. The thread then assigns every other entry in the array to null. The effect of this operation is to fragment memory and create memory pressure. The extent of fragmentation is dependent on the VM and underlying GC. RTGCs can minimize and in some cases eliminate fragmentation [28]. The computation intensive experiment creates low-priority, periodic threads with a period of 20 ms. Each thread executes a tight loop performing a floating point multiplication for 1,000 iterations.

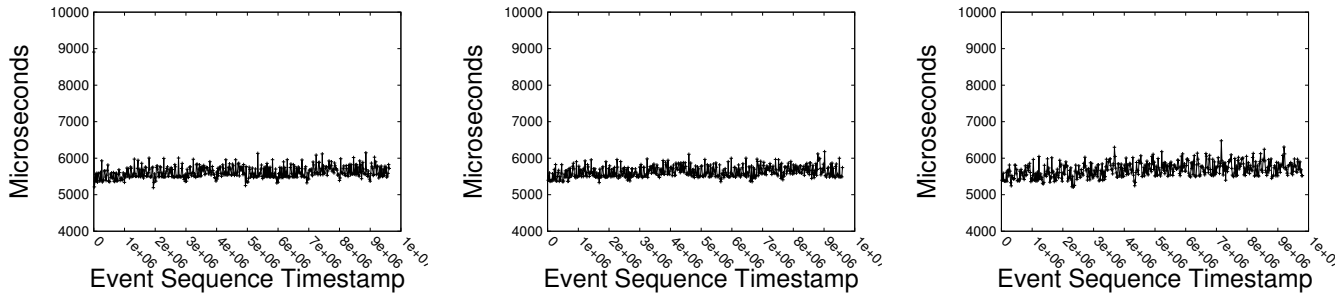
The fall detection application is registered as a `SensorEventListener` with `SensorManager` and executed with the highest priority in the system. After receiving events from the `SensorManager` as outlined in Section 5, the application consumes the `SensorEvent` with the value of x, y, and z coordinates and computes the fall detection algorithm. If a fall is detected the application notifies a server through a direct socket connection using Wi-Fi. Since network does not provide any real-time guarantees, we measure data-passing latency between the time of the sensor raw data detected in the kernel and the time that the sensor event is delivered by `SensorManager` to the fall detection application.

7.3.1 Nexus S Smartphone

Fig. 14 illustrates the observed latency of the sensor event delivery for the fall detection application. To stress the predictability of our `SensorManager` implementation, we have injected memory and computationally intensive threads into the application itself that run alongside of the fall detecting thread. We set these additional threads to a low priority. The Fig. 14a, Fig. 14b, Fig. 14d, and Fig. 14e show the latency of sensor event delivery with one low-priority thread and 100 low priority threads. The upper bound of these four runs was always around 30 ms, and there is no perceivable difference between executing the application with or without memory and computationally intensive threads. For comparison we provide Android performance numbers in Fig. 14c and Fig. 14f to show the effect of low-priority threads on sensor event delivery in stock Android.

7.3.2 LEON3

Fig. 15 lists the results of running the system unloaded, with 30 computational threads and with 30 memory intensive threads. The typical latency is 5.5 ms with a very low standard deviation. The memory intensive test shows a greater variability in the sensor event delivery times but they still fall under 6.5 ms and are also typically 5.5 ms also. RTDroid deployed on this platform creates a very stable system, especially when compared to the results of



(a) RTDroid: No Low-Priority Thread (b) RTDroid: 30 Comp Threads (c) RTDroid: 30 Mem Threads

Figure 15: Memory and Computation stress test for the Fall Detection Application on LEON3.

both Android and RTDroid running on the Nexus S as is shown in Fig. 14.

8. RELATED WORK

Recent work has performed preliminary studies on the real-time capabilities of Android. Maia *et al.* evaluated Android for real-time and proposed the initial models for a high-level architecture [21]. The study did not explore the Android framework, services, IPC, nor core library implementations for their suitability in a real-time context. We believe our work further refines the proposed models.

The overall performance and predictability of DVM in a real-time setting was first characterized by Oh *et al.* [25]. Their findings mirror our general observations on Android. In general, Android performs well in many operational conditions. However, the core system does not provide any guarantees, and the worst-case execution time is parameterized by other applications and components in the system. Thus, to provide real-time guarantees, we need to alter the core system constructs, the libraries, and system services built from them.

Kalkov *et al.* [19] outline how to extend DVM to support real-time; they observed that DVM's garbage collection mechanism suspends all threads until it finishes garbage collection. This design is obviously problematic for applications that need predictability. The suggested solution is to introduce new APIs that allow developers to free objects explicitly. While this design decision does not require a redesign of the whole Dalvik GC, relying on developers to achieve predictability adds a layer of complexity. In addition, their work does not explore how different components within a single application (or across multiple applications) interact through Android's core constructs. We have observed, that the structure of many of Android's core mechanisms, from which many services and libraries are constructed, need to be augmented to provide real-time guarantees. Thus, we believe our implementation is synergistic to such proposals and can be leveraged to provide predictability when applications leverage services, IPC, or the core Android constructs.

9. CONCLUSIONS AND FUTURE WORK

This paper has presented RTDroid, a variation of Android that aims to provide real-time capabilities to Android as a whole system. We have shown that replacing DVM with an RT JVM and Linux with an RTOS is insufficient to run an Android application with real-time guarantees. To address this shortcoming, we have redesigned Android's core constructs and system services to provide tight latency bounds to real-time applications. Our experiments with three platforms—an x86 PC, a LEON3 embedded board, and a Nexus S smartphone, show that RTDroid has good observed predictability on several microbenchmarks as well as a real-time application across three distinct deployment profiles.

Our future work includes the development of Android specific real-time APIs and the design of new programming constructs that naturally support real-time applications on RTDroid. We also plan to extend the current Android's application manifest in order to enable the static definition of real-time features. In parallel, we are working on supporting multi-application execution using Fiji's mixed-criticality support [8, 33] and just-in-time compilation of real-time applications.

Acknowledgements: We thank our shepherd Feng Qian and the MobiSys 2014 program committee for their constructive feedback. We are also grateful to Ethan Blanton, Karthik Dantu, Kyungho Jeon, Taeyeon Ki, Feng Shen, and Jan Vitek for their insightful comments. This work is supported in part by a National Science Foundation Award, CNS-1205656.

References

[1] Android and RTOS together: The dynamic duo for today's medical devices. `http://embedded-computing.com/articles/android-rtos-duo-todays-medical-devices/`.

[2] Linux kernel memory management: Out of memory killer. `http://linux-mm.org/OOM_Killer`.

[3] Real-Time Linux Wiki. `https://rt.wiki.kernel.org/index.php/Main_Page`.

[4] Roving reporter: Medical Device Manufacturers Improve Their Bedside Manner with Android. `http://goo.gl/d2JF3`.

[5] RTEMS. `http://www.rtems.org/`.

[6] What OS Is Best for a Medical Device? `http://www.summitdata.com/blog/?p=68`.

[7] Why Android will be the biggest selling medical devices in the world by the end of 2012. `http://goo.gl/G5UXq`.

[8] Ethan Blanton and Lukasz Ziarek. Non-blocking inter-partition communication with wait-free pair transactions. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, pages 58–67, New York, NY, USA, 2013. ACM.

[9] Jian-Jia Chen and Chin-Fu Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 28–38, 2007.

[10] Angelo Corsaro and DouglasC. Schmidt. The design and performance of the jrate real-time java implementation. 2519:900–921, 2002.

[11] Cpu frequency and voltage scaling code in the linux(tm) kernel. `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`.

[12] Android Developers. Bionic c library overview. `http://www.kandroid.org/ndk/docs/system/libc/OVERVIEW.html`.

[13] Thomas Gerlitz, Igor Kalkov, John Schommer, Dominik Franke, and Stefan Kowalewski. Non-blocking garbage collection for real-time android. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, 2013.

[14] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] Dip Goswami, Martin Lukasiewycz, Reinhard Schneider, and Samarjit Chakraborty. Time-triggered implementations of mixed-criticality automotive software. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 1227–1232, San Jose, CA, USA, 2012. EDA Consortium.

[16] D. Hart, J. Stultz, and T. Ts'o. Real-time linux in real time. *IBM Syst. J.*, 47(2):207–220, April 2008.

[17] Mike G. Hill and Thomas W. Lake. Non-interference analysis for mixed criticality code in avionics systems. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, ASE '00, pages 257–, Washington, DC, USA, 2000. IEEE Computer Society.

[18] iOmniscient. Fall and man down detection. `http://iomniscient.com/index.php?option=com_content&view=article&id=155&Itemid=53`.

[19] Igor Kalkov, Dominik Franke, John F. Schommer, and Stefan Kowalewski. A real-time extension to the Android platform. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 105–114, New York, NY, USA, 2012. ACM.

[20] MinSeong Kim and Andy Wellings. An efficient and predictable implementation of asynchronous event handling in the RTSJ. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, JTRES '08, pages 48–57, New York, NY, USA, 2008. ACM.

[21] Cláudio Maia, Luís Nogueira, and Luis Miguel Pinho. Evaluating Android OS for embedded real-time systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels, Belgium*, OSPERT '10, pages 63–70, 2010.

[22] Sibin Mohan, Frank Mueller, Michael Root, William Hawkins, Christopher Healy, David Whalley, and Emilio Vivancos. Parametric timing analysis and its application to dynamic voltage scaling. *ACM Trans. Embed. Comput. Syst.*, 10(2):25:1–25:34, January 2011.

[23] Yolanda Murphy. Northrop grumman news release: DARPA ASPN project article. `http://www.irconnect.com/noc/press/pages/news_releases.html?d=10029353`.

[24] Northrop to demo darpa navigation system on android. `http://goo.gl/bgRggD`.

[25] Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi, and Soo-Mook Moon. Evaluation of Android Dalvik virtual machine. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 115–124, New York, NY, USA, 2012. ACM.

[26] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the 1999 USENIX Annual Technical Conference*, USENIX ATC'99, 1999.

[27] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 69–82, New York, NY, USA, 2010. ACM.

[28] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 146–159, New York, NY, USA, 2010. ACM.

[29] Military Embedded Systems. Rugged handheld computers suit up with android on the battlefield. `http://mil-embedded.com/articles/rugged-suit-with-android-the-battlefield/#`.

[30] Strand-1 satellite launches Google Nexus One smartphone into orbit. `http://www.wired.co.uk/news/archive/2013-02/25/strand-1-phone-satellite`.

[31] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.

[32] Yin Yan, Sree Harsha Konduri, Amit Kulkarni, Varun Anand, Steven Y. Ko, and Lukasz Ziarek. Rtdroid: A design for real-time android. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, 2013.

[33] Lukasz Ziarek. Prp: Priority rollback protocol – a pip extension for mixed criticality systems: Short paper. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 82–84, New York, NY, USA, 2010. ACM.