RTDroid: A Design for Real-Time Android

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

Abstract
This paper presents our work on the inception of 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 the Android framework layer for real-time systems. Existing work only provides solutions to traditional problems, including adding support for real-time garbage collection at the virtual machine layer as well as 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, the Android programming model, libraries and core systems services. 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 Dalvik VM with a real-time VM, and leverages off-the-shelf real-time OSes. We demonstrate the feasibility and predictability of our solution on three different platforms. The evaluation results show that our design can successfully provide predictability to Android applications even under heavy loads.

Index Terms
Real-time Systems, Mobile Systems, Smartphones, Android

1 INTRODUCTION
There is a growing interest in adopting Android in embedded, real-time environments. The United States military has explored the use of Android in portable navigation systems [30], [31] and for the detection of wounded soldiers [22], [39]. The United Kingdom and NASA have started examining the use of Android devices as a control system in satellites [6]. In health care, much discussion is currently ongoing as to how the medical device industry can adopt Android [4], [7], [8], [1]. 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.

Surprisingly little work, however, 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 [27] and extensions to Android’s Java VM (Dalvik) that enable real-time garbage collection [24], [17]. 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 stock Android and identify its limitations. We then replace and redesign several internal components of Android to provide predictability in three aspects—time, space, and resource management. More specifically, we replace the Android kernel with a real-time kernel that provides system primitives for resource management, e.g., real-time task scheduling, shared resource allocation, and interruption handling; we also replace the Android Java VM with a real-time Java VM that provides real-time garbage collection for memory predictability; on top of these off-the-shelf building blocks, we redesign Android’s programming constructs and system services to provide time predictability. We recognize, however, that Android is a vast system with many components, and that it is difficult to evaluate all of the Android components. 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 with timing guarantees. As the rest of the paper shows, this goal alone poses many hard challenges. It is also a prerequisite to supporting multiple real-time applications (i.e., mixed criticality [21], [19]—the ability to execute multiple components with different criticality levels safely).

More concretely, this paper makes the following four contributions:

- 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.
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 timing guarantees.

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 [36] as our Java runtime and two real-time OSes (RTLinux [3], [20] 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 their solution.

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 Android’s version of Linux. In all three platforms, we show that our redesigns provide timing guarantees to applications even under heavily-loaded conditions. To validate the timing guarantees of our system, we have implemented micro-benchmark for RT Looper and Hanlder and RT-AlarmManager, and executed these micro-microbenchmarks on all three platforms. For thorough evaluation of the RT-SensorManager, we implement a fall detection algorithm and port a traditional real-time system benchmark, jPapaBench [23] as real-time applications. We have deployed these two applications on a Nexus S and LEON3, and measured the sensor data delivery latency. We show that even with hundreds of non-real-time “noise” generating threads, the application shows good timeliness and predictability. In addition, we present observed worst-case execution times for our redesigned components to empirically demonstrate the worst-case execution behavior of our system.

This paper combines our previous workshop and conference papers [43], [42], [41] and presents a comprehensive discussion of our current design. 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 10. Additional performance measures, experiments, raw data, plotting scripts, as well as the implementation are publicly available on our website: http://rtdroid.cse.buffalo.edu.

The rest of the paper is organized as follows. Section 2 presents the motivation for real-time Android. Section 3 provides an overview of our system, RTDroid. Sections 4, 5, and 6 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 7 reports our experience in replacing non-real-time components with real-time counterparts. Section 8 demonstrates experimental results of the three components with our micro-benchmarks and test applications. Finally, Section 9 discusses our related work and Section 10 discusses our future work and conclusions.

2 Motivation

This section presents two broad use cases as well as examples of their concrete deployments to motivate the design and utility of RTDroid. We envision RTDroid to be leveraged in two distinct ways: (1) to run a single real-time app on either specialized embedded hardware or a mobile device and (2) to run real-time apps along with existing non-real-time apps in a mixed-criticality environment. The former case is the primary contribution of this paper, but important design considerations must be made to support the latter in the same system. As such, we discuss salient implementation details for supporting multiple real-time apps along with non-real-time apps.

2.1 Single Real-Time App

As Android becomes increasingly popular, researchers have begun to explore its use as a platform for safety- and mission-critical apps. For example, the UK has built and launched a satellite equipped with a regular control system as well as a smartphone (Google Nexus One) [6]. The goal of the satellite is to experiment with transferring control from the standard control system to the mobile device itself.

The medical device industry has expended significant resources in exploring Android as a future platform [4], [7], [8], [1]. They report that Android is well suited for envisioned applications, such as remote patient monitoring devices including cardio monitors and glucose analyzers, because such applications require support for wireless connectivity as well as good user interface design. Other proposed applications include fall and gate monitoring for the elderly and patients undergoing rehabilitation.

In all of these scenarios, Android is used as a platform to run a single real-time app. The underlying hardware can be a traditional embedded board or a mobile device. The benefit of using Android in these scenarios is the rich APIs and libraries that exist on Android. It supports connectivity through Wi-Fi, Bluetooth, 3G, and 4G; it provides native support for various sensors such as GPS, accelerometer, camera, and gyroscope; and it fosters an intuitive user interface design through a touch screen and gesture recognition. Control and medical apps typically require these functionalities and their development can be streamlined as well as standardized through the Android APIs and libraries.
2.2 Mixed-Criticality

In addition to supporting a single real-time app, we envision allowing a mobile device to run multiple real-time apps along with regular apps through mixed-criticality support. For medical monitoring, the same mobile device that monitors its user’s medical conditions can be used as a traditional smartphone. This reduces the number of devices a user needs to carry. Similarly, if a user requires multiple medical monitoring applications, they can be executed on the same device.

Google reports that its Play Store currently has more than one million apps available for Android. The ability to install and leverage these apps will greatly simplify the development and maintenance of real-time apps. For example, a medical monitoring device may be expected to send a report to the patient’s doctor on a daily or weekly basis. Since there are many apps that already provide such a functionality (e.g., Gmail) that allows other apps to send emails through it, the monitoring app can simply leverage one of these apps; this reduces the complexity of development and maintenance of real-time apps.

We note that supporting such mixed-criticality scenarios requires significant engineering effort to support downloading and installation of real-time apps, validation of newly installed apps with regard to the schedulability of other apps present in the system, JITing DEX bytecode to specialize for the target platform, access to I/O and hardware sensors, and power management issues. Such challenges are out of the scope of this paper. Instead, we focus on the core Android constructs and system services that are used not only for single-app scenarios, but also for multi-app scenarios as those are essential to build Android compatibility as we describe in the next section.

3 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.

3.1 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 [24], [17] 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.

1. As of Oct, 2014 (https://play.google.com/about/apps/)
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 2 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 4, we discuss this problem in more detail and present our solution. Section 8 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 5 and 6, 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.

3.2 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 uni-process 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.

3.2.1 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) 18 support without scoped memory. This goal of providing Android compatibility makes our architecture unique and different from potential architectures discussed previously in the literature [27], where much of the focus is on the kernel and the JVM layers.

3.2.2 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 [36], 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 [37], 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. By constructs, we mean abstract Java classes provided by the Android application framework. Application developers can extend these abstract classes to leverage advanced functionalities.
3.2.3 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, which we detail in Section 4. 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 and 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.

3.2.4 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 preemptible RTLinux kernel. As we detail in Section 7, 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 application 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 8 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.

4 RTLooper and RTHandler

As discussed in Section 3, 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.

4.1 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.

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

3. With a fully preemptible kernel, all parts of the kernel become preemptible by a high priority thread.
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 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 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 come before finishing the processing of non-real-time messages, the real-time message will get delayed further by non-real-time messages.

4.2 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 the messages 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.

4.3 Worst-Case 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 $T_i$ to be the $i^{th}$ message
Fig. 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.

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):

$$S_i^j = \sum_{l=0}^{i} (h_i^j + deq(T_i^j)), \quad (1)$$

Where $h_i^j$ is the cost of time to handle $T_i^j$ and $deq(T_i^j)$ 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):

$$\text{phase}_0(T_i^j) = \sum_{p>j} S_{\text{last}}^p + S_i^j. \quad (2)$$

Where last is the last message in the message queue with priority $j$. Since the system also handles new incoming messages, which may have a priority greater than or equal 4 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):

$$\text{phase}_n(T_i^j) = \sum_{p>j} \sum_{i=0}^{\text{phase}_{n-1}(T_i^j) \cdot R_p} (h_i^p + deq(i) + enq(i)). \quad (3)$$

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 phase$_n$, the RHS represents the total time cost for handling all messages that arrive during 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 $R_p$ gives the number of messages currently in each priority based queue. The recursion ends when phase$_n$ is smaller than the time unit of the rate $R_p$. Thus, the summation of all phases is the actual worst-case execution time for handling message, $m$ as shown in Equation (4):

$$WCET(T_i^j) = \text{phase}_0(T_i^j) + \text{phase}_1(T_i^j) + ... + \text{phase}_{n-1}(T_i^j) + \text{phase}_n(T_i^j). \quad (4)$$

4. 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.
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, is less than phase−1.

5 RT Alarm Manager

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 PowerManager that controls power; 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 app, 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 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.

5.1 Background and Challenges

AlarmManager receives timer registration requests from applications and sends a “timer triggered” message to these applications when its timer fires. Since real-time applications frequently rely on aperiodic 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 and execution time, is made to the AlarmManager every time an application registers an alarm. When 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.

5.2 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

5. This message is associated with a callback for the application which gets executed when the message is delivered.
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) [18] specifies the interface for.

We have implemented two versions for AEH. The first is a per-thread AEH implementation used in our workshop paper [43], 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 [26]. 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 [13].

5.3 Worst-Case Execution Time

The worst-case execution scenario for AlarmManager is similar to that discussed for the Looper and Handler in Section 4.3. 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.

- \( T^j_i \) represents the \( i^{th} \) alarm registered by application with priority \( j \).
- \( S^j_i \) represents the time cost for handling the \( i^{th} \) alarm in priority level \( j \).
- \( h^j \) is the cost of time to execute the alarm \( T^j_i \).
- \( \text{last} \) is the last alarm in priority level \( j \).
- \( \text{enq}(T^j_i) \) is the cost of alarm registration.
- \( \text{deq}(T^j_i) \) is the cost of alarm delivery.

6 RT Sensor Architecture

Another system service we redesign in our system 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.

6.1 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 six hardware sensors and seven 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 the 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.

### 6.2 Redesign

We redesign the sensor architecture for RTDroid to address the two issues mentioned above. Our design is inspired by event processing architectures used for Web servers [33], [40]. 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 to 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.

6.3 Worst-Case Execution Time

The worst-case execution scenario for SensorManager is slightly different than what we have discussed in Section 4.3 and 5.3. The upper bound for delivery of the sensor event to a sensor listener, $I$, 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 $I$, (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):

$$phase_0(T^i_j) = \sum_{p \geq j} P_j(sensor_e) + \sum_{p > j} S_{\text{last}}^p + S_i^j$$

$$phase_n(T^i_j) = \sum_{p \geq j} \left( \sum_{i=0}^{\text{phase}_{n-1}(T^i_j) \ast R_p} (h_i^j + \text{deq}(T^i_j) + \text{enq}(T^i_j)) \right).$$

- $T^i_j$ represents the $i^{th}$ sensor listener in application with priority $j$.
- $S_i^j$ represents 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 $T^i_j$.
- $\text{deq}(T^i_j)$ is the cost of listener registration.
7 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 LEON3, and an ARM-based smartphone environment with Nexus S. 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.

7.1 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 machine to guarantee their timeliness or leverage off-the-self 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, we have designed and implemented an accelerometer daughter board as well as the associated RTEMS compliant driver.

7.2 Nexus S Smartphone

Unfortunately, the same approach is not adequate for execution 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.

7.2.1 Bionic

Android does not utilize glibc as the core C library, instead it uses its own library called Bionic [16]. Bionic is a significantly simplified, optimized, light-weight C library specifically designed 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.

7.2.2 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.

7.2.3 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 [14]. 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 not necessary, but a source of missed real-time task deadlines. Memory management is provided by Fiji VM’s Schism real-time, fragmentation tolerant GC [37]. 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 into consideration voltage scaling has been vetted for hardware architecture that provides predictable mechanisms for doing so [12].

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 [29] to validate the hardware and leverage this timing analysis to modify the kernel and VM schedulers appropriately.

8 Evaluation

To measure and validate our prototype of RTDroid, we have tested our implementation on three different machine configurations, each of which represents one of our target deployments outlined in Section 3.2.4. The first configuration utilizes an Intel Core 2 Duo 1.86 GHz 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 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 configuration running at 50Mhz. The development board has an 8MB flash PROM and 128MB of PC133 SDRAM.

We repeat each experiment multiple times, and present empirically-observed worst case execution time metrics as it is difficult to provide worst case latencies for the whole system without specialized timing hardware and whole system static analysis. 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 the worst-case execution time of each component individually presented in Sections 4.3, 5.3, and 6.3.

To enable testing of our RT SensorManager on LEON3, we have designed and developed a daughter board with interface circuitry based on the Freescale Semiconductor MMA8452Q triple axis accelerometer. We have developed an RTEMS driver for the accelerometer and integrated it into our RTEMS build. We have repeated the experiments multiple times, and present empirically-observed worst case execution time metrics as it is difficult to provide worst case latencies for the whole system without specialized timing hardware and whole system static analysis. 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 the worst-case execution time of each component individually presented in Sections 4.3, 5.3, and 6.3.

8.1 RT Looper and RT Handler Microbenchmarks

To measure the effectiveness of our prototype, we have conducted 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 message. To do this, we take a timestamp in the real-time thread prior to sending the message. This timestamp is the data encoded within the message. A 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 the 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 x86 and Nexus S, we have varied the number of low-priority threads in increments of 10 from 0 to 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 to 30 when running on LEON3.

Fig. 9 and Fig. 10 demonstrate the consistent latency of our RT Looper and RT Handler implementation. On x86, 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. Fig. 11a shows that the message latencies fluctuate from 0.04 ms to 0.5 ms on Nexus S with the ondemand governor. This is due to the periodic releases of each low-priority thread which vary the system load and trigger the governor module to adjust the frequency of CPU. The tests with the performance governor show a consistent latency in Fig. 11a, 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. 11c.

To quantify the empirical worst case behavior of our RT Looper and RT Handler implementation, we have run the microbenchmark 10 times on Nexus S and LEON3. Fig. 12 shows observed WCETs of message passing latency over 10 executions with increasing number of low-priority threads. Fig. 12a shows that the observed WCETs range from 0.1 ms to 0.7 ms, the standard deviations of the latencies are from 0.01 ms to 0.2 ms. These variances are caused by scheduling time cost on a non-fully-preemptible Linux RT kernel on Nexus S. We do not observe such variance on LEON3, show in Fig. 12b, since the RTEMS kernel provides more consistent scheduler for tasks dispatching. Thus we conclude that the variance of the observed WCETs on Nexus S are most likely caused by kernel jitter, not our implementation.
RTDroid: Per Thread

RTDroid: Thread Pool

Fig. 12: Observed WCETs of Message Passing Latency

Fig. 13: RT AlarmManager Per Thread vs Thread Pool on x86.

8.2 RT AlarmManager Microbenchmarks

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 (AEH 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 measures shows how long it takes for the system to deliver an alarm from the AlarmManager to the app. We run the experiment on all three platforms. These results show the timing and latencies 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 5, we have implemented two techniques for alarm management in RT AlarmManager—one with a per-thread AEH implementation used in our previous workshop paper [43] and another implemented with a thread pool. We show the predictability of RTDroid with each technique by using threads ranging from 5 to 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.

Fig. 13 shows the results of the per-thread AEH and the thread pool AEH experiments running on x86. The latency of the entire alarm delivery for per-thread AEH on the x86 is bounded from 0.22 ms to 0.33 ms. The asynchronous event fire latency is consistently around 0.11 ms. The per-thread implementation exhibits a slightly lower performance with the alarm delivery bounded from 0.26 ms to 0.36 ms. The results of the thread-poll implementation is 0.1 ms longer than the results of the per-thread implementation. Such variations are expected and caused by alarm queuing in the thread pool itself.

To evaluate the empirical worst case behavior of RT AlarmManager, we have repeated the same experimental scenario 10 times on Nexus S and LEON3. Fig. 14 shows the observed WCETs of alarm delivery latency and AEH firing latency as a function of the number of low-priority alarms on Nexus S. It shows a similar pattern as it does on x86 with slightly larger values. This is not surprising considering the different hardware architectures between x86 and Nexus S in terms of the type and frequency of their CPU and available memory. For per-thread scheduling, the observed WCETs of alarm delivery range from 0.44 ms to 0.77 ms with different number of low-priority alarms, the standard deviations of them are from 0.01 ms to 0.05 ms. Fig. 14a presents the AEH firing latency between 0.34 ms to 0.38 ms. It shows that the alarm delivery latency is attributed to AEH firing latency. For thread pool scheduling, Fig. 14b demonstrates the observed WCETs of alarm...
delivery from 0.19 ms to 0.22 ms with around 0.09 ms standard deviation. Since the number of tasks in the system are limited, the alarm delivery latency are more consistent.

Fig. 15 shows the same results of RT AlarmManager microbenchmark on LEON3. The overall performance is around 3 times slower than Nexus S, because of the board’s slower CPU frequency. However, due to the RTEMS kernel, the observed WCETs of alarm delivery latency are more consistent, around 2.5 ms with per-thread scheduling, and 1.1 ms with thread pool scheduling. The standard deviations are almost negligible with both scheduling methods. In general, we have observed that the alarm delivery latencies are dominant by two factors: 1) the number of schedulable objects in the system. 2) The cost of AEH firing. As we have discussed above, the alarm delivery latencies with the thread-pool scheduling mechanism are more consistent under a fully-preemptive kernel. For soft-real-time application, these techniques may be sufficient to provide a soft-real-time alarm with bounded latency. For hard-real-time usage that requires sub-millisecond latency, systems require more precise timing measurement for thread wake-up in order to trigger the AEH with lower cost. A dedicated real-time hardware clock needs to be integrated as an additional device on either the smartphone or embedded board to achieve such latencies. Indeed many embedded boards used in such systems have such hardware. We believe with a hardware real-time clock, the overall latency of alarm delivery will be just the difference of the alarm delivery latency and the AEH firing latency, approximately 0.3 ms according to our experimental results in Fig. 15a and Fig. 15b.

8.3 Real-Time Fall Detector

To validate the predictability of our sensor architectures in data delivery, we have created a soft real-time fall detection application that leverages our SensorManager outlined in Section 6. 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 and RTGCs are known to be able to minimize and in some cases eliminate fragmentation [37]. The computation intensive experiment creates low-priority, periodic threads with a period of 20 ms. Each thread executes a tight loop performing 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 system. After receiving events from the SensorManager as outlined in Section 6, 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.

Fig. 16 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. Fig. 16a, Fig. 16b, Fig. 16d and Fig. 16e 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 app with or without memory and computationally intensive threads. For comparison we provide Android performance numbers in Fig. 16c and Fig. 16f to show the effect of low-priority threads on sensor event delivery in stock Android.

Fig. 17 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 both Android and RTDroid running on Nexus S as is shown in Fig. 16.

Fig. 18 presents observed WCETs of sensor event delivery on Nexus S and LEON3. On Nexus S, the observed WCETs of sensor event delivery are from 24 ms to 28 ms with 0.2 ms standard deviation. The delivery latency mirrors the sensor polling rate of RT SensorManager framework, 25 ms. It is dominant by time difference between Linux kernel handle sensor interruption and RT SensorManager read the sensor data. On LEON3, the delivery latencies are reduced to around 5 ms, because we develop sensor driver to directly read device register through i2c bus on RTEMS, instead of buffering and polling from kernel buffer.

8.4 jPapaBench

To evaluate our design of the RT SensorManager in real-world context, we leveraged a port of jPapaBench [23] to execute on top of RTDroid. We first introduce jPapaBench and present the changes necessary to execute with RTDroid. We then present performance results using RTEMS running on LEON3 as well as RTLinux running on Nexus S.

8.4.1 jPapaBench Implementation in RTDroid

jPapaBench is designed as a Java real-time benchmark to evaluate Java real-time virtual machines. It mirrors the function of paparazzi [9], a UAV autopilot system written in C. The jPapaBench code is conceptually divided into three major modules: the autopilot, which controls UAV flight and is capable of automatic flight in the absence of other control; the fly-by-wire (FBW), which handles radio commands from a controlling station and passes information to the autopilot to be integrated into flight control; and the simulator, which collects information from each of the other modules, determines the UAV’s location, trajectory, and generates input from the environment (such as GPS data, servo feedback, etc.). Two of these modules, the autopilot and fly-by-wire (FBW), are housed in different microcontrollers on the conceptual hardware, and the jPapaBench code simulates a serial bus between them—they have no other direct communication path. The simulator is only loosely coupled to the FBW module, but shares a moderate amount of state with the autopilot. A high-level overview of the jPapaBench system is provided in Fig. 19.

As noted by Blanton et. al. [11], the simulator module updates the autopilot state with simulated sensor values and this provides a natural point for separating the simulation tasks from the main autopilot tasks. We integrate our RTDroid system into simulator module by delivering simulated data into the bottom-most layer of RTDroid, which in turn provides this data to the autopilot in jPapaBench. At a high-level, the simulation component of jPapaBench feeds simulated sensor data into an intermediate buffer that our polling thread pulls data from. This is used to model the kernel behavior over actual hardware sensors. The simulated sensor data is then processed by the RT SensorManager and delivered the control loops, which require data generated by a given simulated sensor. The control loops were modified slightly to subscribe to the RT SensorManager using traditional Android APIs.

8.4.2 Experimental Results

In all our results, we show end-to-end latency as well as the breakdown of latency. In Fig. 20 through Fig. 22, the circle points show the overall end-to-end latency from simulated sensor event generation till event delivery in jPapaBench. As stated earlier, we feed the simulated sensor data generated by jPapaBench’s simulator into an intermediate buffer first. This buffer emulates a typical kernel behavior. Then our polling thread pulls simulated sensor data out of it. Thus, the end-to-end latency measures the buffering delay in addition to the latency incurred purely in our architecture. The square points...
Simulations

- SimulatorFlightModelTaskHandler
- SimulatorGPSTaskHandler
- SimulatorIRTaskHandler

Fly-by-Wire (FBW)
- TestPPMTaskHandler
- sendDataToAutopolit
- CheckFailsafeTaskHandle
- CheckMega128ValuesTaskHandle

Autopilot
- RadioControlTaskHandler
- NavigationTaskHandler
- AltitudeControlTaskHandler
- ClimbControlTaskHandler
- StablizationTaskHandler
- LinkFBWSendTaskHandler
- ReportingTaskHandler

Fig. 19: jPapaBench Task Dependency

![Graph showing task dependency](image)

(a) Nexus S base line performance
(b) LEON3 base line performance

Fig. 20: RT SensorManager performance base line

show the buffering delay, and the cross points show the raw latency of the rest of our architecture below applications, i.e., the processing threads as well as RT Handler. The y-axis is latency given in millisecond and the x-axis is the time of release of the simulator task in jPapaBench. As shown in Fig. 20 through Fig. 22, since the sensors are periodically simulated, there is little difference between Nexus S and LEON3 and the data is generated at a rate ten times that of the hardware sensors’ capacity on Nexus S. Fig. 20 shows the baseline performance of the RT SensorManager on Nexus S and LEON3, respectively.

In addition, we run our experiments with three different configurations: memory, computation, and listener. The memory workload creates low priority noise making threads, each of which periodically allocates a 2MB byte array, then de-allocates every other element, and finally deallocates the array. This workload serves to create memory pressure in terms of total allocated memory as well as to fragment memory. The computation workload creates low priority noise making threads, each of which performs a numeric calculation in a tight loop. This workload serves to simulate additional computation tasks in the system, which are disjoint, in terms of dependencies, from the main workload. The listener workload creates low priority threads, each of which subscribes to receive sensor data from the RT SensorManager. This workload simulates low priority tasks, which periodically subscribe and consume sensor data from sensors that are utilized by the high priority real-time tasks in jPapaBench.

Fig. 21 and Fig. 22 show performance results obtained on Nexus S and LEON3, respectively. The figures illustrate two workload configurations for each system level configuration: 5 noise generating threads and 30 noise generating threads,
Fig. 21: RT SensorManager stress tests on Nexus S

respectively. Interested readers can view additional experimental results and raw data on our website \(^6\). The baselines for both hardware platforms are provided in Fig. 20a and Fig. 20b. Fig. 23 presents the observed WCETs of the end-to-end delivery latency over 10 executions with an increasing number of noisy threads.

The observed WCETs of sensor delivery range from 19.65 ms to 25.55 ms with 0.19 ms to 5.31 ms standard deviations on Nexus S. On LEON3, the results of LEON3 reflect the same trend with larger values, WCETs are from 27.20 ms to 35.40 ms with 3.7 ms to 7.47 ms standard deviations. Such variances are caused by the sensor data buffering, and the delays from other tasks with higher priorities in the application. It is reasonable that we obtain larger latencies on LEON3 than Nexus S due to the massive differences in computation capability and main memory in the architectures. In addition, we do not observe performance difference between three different noisy threads. This means that there are enough time slots in the system for each task to meet its deadline. The RTGC has enough slack time to keep up with memory allocations performed in the memory noise generating threads on both Nexus S and LEON3.

9 RELATED WORK

Recently, Google has released Android Wear and Android Auto that can run on various types of embedded devices. Android Auto provides easy interfaces for developing automotive Apps, while Android Wear focuses on notifications and sensing gear toward biometrics. However, there is no evidence that these systems are planning to support any real-time guarantee. In academia, researchers have started to look into the real-time capabilities of Android. Maia et al. evaluated Android for real-time and proposed the initial architectural models \(^{27}\). Their proposed models are depicted in Fig. 24. The first system model (Fig. 24a) is built around a clean separation between Android and real-time components, allowing for real-time applications to run directly on top of a real-time operating system (RTOS). Although viable, this model prevents the creation of real-time Android apps, instead opting for a system that can run both Android apps and separate real-time applications. In addition, real-time applications are prevented from leveraging the features offered by Android and cannot include any Android related services or libraries. The next model (Fig. 24b) is similar to the first, but instead of swapping the standard Linux kernel for an RTOS, it introduces a real-time hypervisor at the bottommost layer. In this model, Android runs as a guest operating system in one partition and real-time applications in another. Thus, this model suffers from the same deficiencies of the first. The last two models (Fig. 24c and Fig. 24d) permit the construction of real-time Android apps by adding a secondary VM with real-time capabilities or by extending DVM with real-time support (alternatively, replacing DVM with a real-time JVM) respectively. These two approaches provide the groundwork for predictability and determinism within the Android system by replacing the standard Linux kernel with an RTOS as well as introducing

\(^6\) Full results available: http://rtdroid.cse.buffalo.edu
real-time features at the VM level. Notably, these models support real-time Android apps, the use of Android features, in addition to Android services and libraries. The last two models, unfortunately, provide little or no insight on how Android features, services, and libraries can themselves be extended to support execution of real-time Android apps. Unlike these models, we advocate a clean-slate design.

The overall performance and predictability of DVM in a real-time setting was first characterized by Oh et al. [32]. Their findings mirror our general observation on Android that the internals of Android are not designed with predictability in mind. In the work of Perneel et al., they have also confirmed this observation by evaluating the performance of different components [34]. Similarly, Mongia et al. have also showed that deadlines were frequently missed in Android with delays ranging from 1 ms to 500 ms [38]. Although they have observed that Android provides reasonable performance in many other conditions, the core system does not provide any guarantees, and the worst case execution time is parameterized by other apps and components in the system. Thus, to provide real-time guarantees, we need to alter the core system constructs, the libraries, the framework, and the system services built from them.

Mauerer et al. [28] proposed an approach that builds a partitioned system with RTLinux kernel, and enabled communication channels between a real-time partition and an Android partition. The authors evaluated real-time native process behaviors and their interactions. However, their approach does not consider the internals of Android, such as Dalvik

Fig. 22: RT SensorManager stress tests on LEON3

Fig. 23: RT SensorManager Observed WCET of Sensor Data Delivery in jPapaBench Application
Applications
Application Framework
Core
Dalvik VM
Libraries
RTOS Kernel
RT
Apps

(a)

Applications
Application Framework
Core
Dalvik VM
Libraries
RT Hypervisor
RTOS Kernel
RT
Apps

(b)

Applications
Application Framework
Core
Dalvik VM
Libraries
RTOS Kernel
RT JVM

(c)

Applications
Application Framework
Core
RT JVM / Extend Dalvik
Libraries
RTOS Kernel

(d)

Fig. 24: Real-Time Android System Models Proposed by Mais et al. Shaded components represent additions or changes to the Android architecture.

VM, runtime libraries, and the communication mechanisms. Kalkov et al. [24] 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 apps 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 re-design of the whole Dalvik GC, relying on developers to achieve predictability adds a layer of complexity. Kalkov et al. also explored how different components within a single app (or across multiple apps) interact through the Intent message-passing mechanism on Android, and re-designed it for predictability [25]. This design is in-line with our approach that re-designs the core constructs and mechanisms of Android. In this sense, their paper presents an independent confirmation point that Android’s specialized programming model and runtime provide many design and implementation challenges for real-time.

RTDroid currently does not provide any real-time guarantee either with CPU frequency scaling or in a multi-core environment. There are some known techniques to deal with such scenarios [35], [15]; adopting and implementing these techniques in RTDroid will enable us to address the issues that arise in those environments.

10 CONCLUSIONS AND FUTURE WORK

In this paper we have shown that replacing DVM with a RT JVM and Linux with an RTOS is insufficient to run Android application with real-time guarantees. To address this shortcoming, we presented RTDroid, an initial design of a real-time Android system focusing on supporting a single real-time app. We have designed RTDroid to be VM and RTOS agnostic and with mixed-criticality in mind. We have validated our design and prototype, showing RTDroid has good observed predictability.

Our next step is to move toward a multi-app, mixed-criticality execution environment and expand our IPC and service support to provide predictable, cross partition usage of the Android constructs. In addition, we plan to explore the use of scoped memory for providing tighter memory guarantees within core Android constructs. This multi-app, mixed-criticality environment will allow us to study more interesting questions. For example, we can study the question of how to design a fully-functional smartphone that is capable of making calls and downloading apps, all with appropriate real-time guarantees. This requires further investigation into the phone sub-system since it can create jitter if the kernel does not treat it as a low priority task. In addition, enabling app downloads requires further investigation into real-time JIT, following the popular distribution model of performing JIT compilation for a downloaded app. These topics are part of our future work.

REFERENCES


Yin Yan received the B.Eng degree from Beijing Information Science and Technology University in 2009, and the MEng degree from University at Buffalo SUNY, NY, US in 2013. He is currently working toward to the PhD degree in the Department of Computer Science and Engineering at University at Buffalo SUNY. His research interest lies in real-time OS and embedded systems.

Shaun Cosgrove received the B.Eng in Digital and Software Systems Engineering from GMIT, Ireland in 2006 and the MEng degree in Computer Science and Engineering from University at Buffalo SUNY, NY, US in 2015.

Varun Anand received the MEng degree in Computer Science and Engineering from University at Buffalo in 2014. He currently work at Akamai Technologies as software engineer.

Amit Kulkarni received the MEng degree in Computer Science and Engineering from University at Buffalo in 2014.

Sree Harsha Konduri received the B.Tech in Electronics and communication engineering, Jawaharlal Nehru Technological University, and the MEng degree from University at Buffalo SUNY, NY, US. He currently works at Amazon.com, Inc as software engineer.
**Steven Y. Ko** Steve Ko is an assistant professor in the Department of Computer Science and Engineering at the University of Buffalo, The State University of New York. His research interest spans in systems and networking, with the current focus on mobile systems. He received his PhD from the University of Illinois at Urbana-Champaign in Computer Science.

**Lukasz Ziarek** received the B.S. degree in Computer Science from the University of Chicago and the Ph.D. degree in Computer Science from Purdue University. Lukasz Ziarek is an assistant professor in the Department of Computer Science and Engineering at the University of Buffalo, The State University of New York. His research interests are in programming languages and real-time systems.