RTDroid: A Design for Real-Time Android

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

ABSTRACT

There has been much recent interest in adding support for real-time capabilities to Android. Proposed architectures for doing so fall into four broad categories, but only two support real-time Android applications. These two proposals include a real-time garbage collector for real-time memory management and a real-time operating system for real-time scheduling and resource management. Although they provide the fundamental building blocks for real-time Android, unfortunately such proposals are incomplete. In this paper we examine the Android programming model, libraries, and core systems services in the context of the Fiji real-time VM coupled with the RT Linux and RTEMS real-time operating systems. We show that even with a real-time operating system as well as real-time memory management, the predictability of core Android constructs is poor. To address this limitation we propose a preliminary RTDroid design and show its applicability to real-time systems.

Categories and Subject Descriptors

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

1. INTRODUCTION

Android's open source nature has prompted its ubiquitous adoption in various embedded system domains. Instead of being built around the Java Virtual Machine (JVM), Android uses the Dalvik Virtual Machine (DVM) [12]. Unlike a JVM, DVM leverages register based [21] bytecode (called DEX [2]) instead of Java bytecode. The DVM supports just-in-time compilation (JIT) [10] to optimize for the target deployment device. Android, itself, runs on top of a modified Linux kernel and provides a framework layer for applications to leverage. The framework layer is built from Linux kernel functionality, third party libraries, as well as core Android mechanisms. Applications targeting the Android system are colloquially referred to as "apps."

JTRES'13, October 09 - 11 2013, Karlsruhe, Germany

http://dx.doi.org/10.1145/2512989.2512993.

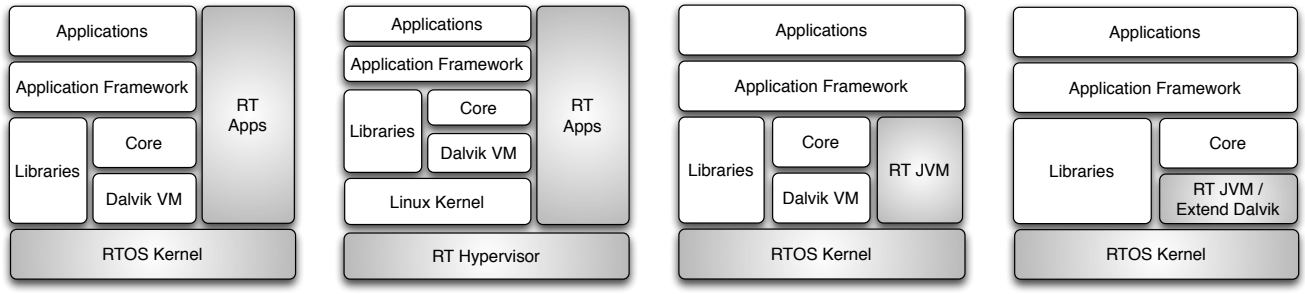
Since its inception, there has been much interest in a real-time Android variant; researchers have proposed four canonical system architectures [15, 17] for extending Android with real-time capabilities. These architectural models are illustrated in Fig. 1. The first proposed system architecture (Fig. 1a) 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 approach (Fig. 1b) 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, running Android as a guest operating system in one partition and real-time applications in another. This model suffers from the same deficiencies of the first.

The last two models (Fig. 1c and Fig. 1d) 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 ground work for predictability and determinism within the Android system by replacing the standard Linux kernel with an RTOS as well as introducing 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.

In this paper, we demonstrate a new approach to achieving real-time capabilities with the Android system. Our prototype, called RTDroid, focuses on extending an existing real-time JVM [1] with Android constructs and leveraging an off-the-shelf RTOS. We have identified critical changes to the Android framework and core mechanisms necessary to guarantee predictable runtime performance for apps that leverage Android mechanisms and services. Specifically, the contributions of this paper are:

- An investigation of the core Android components and their suitability for real-time.

[1]Extending an RT JVM for Android requires DEX support; we discuss this issue in Section 3.2.4.



(a) (b) (c) (d)

Figure 1: Proposed RT Android System Architectures. Shaded components represent additions or changes to the Android architecture.

- An initial system implementation of core Android constructs and services, which provides real-time guarantees for real-time threads. Our system is implemented on top of the Fiji VM, but is both VM and RTOS agnostic. Our current design focuses on supporting a single real-time app, but its design can be generalized to support multiple apps.
- A detailed evaluation study of our system leveraging two separate RTOSes, RT Linux and RTEMS. Our performance results indicate that previously proposed models, without any changes to the framework layer, cannot provide real-time guarantees if real-time threads and/or tasks leverage Android's core mechanisms and services.

The rest of paper is organized as follows: in Section 2, we provide two high-level motivating examples for RTDroid. In Section 3 we outline Android details, specifically focusing on the challenges of adding real-time capabilities. We detail our solution in Section 4. Evaluation and experiments are presented in Section 5. We discuss related work and conclusions in Section 6 and Section 7 respectively.

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 being 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 launched a satellite equipped with a regular control system as well as a smartphone (Google Nexus One) [7]. 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 [5, 8, 9, 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. As we detail in Section 3, 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 gestures. 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 the use of a mixed-criticality system. 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 700,000 apps available for Android [2]. The ability to install and leverage these apps will greatly simplify the development and maintenance of real-time apps. For example, any 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 those apps; this reduces the complexity of development and maintenance of real-time apps.

We note that supporting such mixed-criticality scenarios

[2]As of June, 2013 (`https://play.google.com/about/apps/`)

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 [10], 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. THE CASE FOR RTDROID

To support the scenarios discussed in the previous section, we advocate a clean-slate design for RTDroid. We start from the ground up, leveraging an established RTOS (*e.g.*, RT Linux [4, 14], RTEMS [6]) and an RT JVM (*e.g.*, Fiji VM [19]). Upon this foundation we build Android compatibility. RTDroid provides a faithful illusion to an existing Android app running on our platform that it is executing on Android. This necessitates providing the same set of Android APIs as well as preserving their semantics for both regular Android apps and real-time apps. For real-time apps, 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 RTSJ [13] support without scoped memory.

This approach, however, does not mean that we can simply "port" the existing Android code to run on an RTOS and an RT JVM. As a consequence, we do not have full freedom to re-architect the underlying implementation of all the APIs. The unique programming and runtime model of Android requires careful consideration as to how we can simultaneously support Android compatibility while providing real-time guarantees. In this section, we discuss why this is the case by first presenting the benefits of our design, then discussing the challenges in realizing our design within the context of a concrete system.

3.1 Benefits

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. For example, our RTDroid prototype uses Fiji VM [19], 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 [20], synchronization, threading, etc. and crucially provides mixed-criticality support necessary for executing multiple applications within a single VM. 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 each use case discussed in Section 2. Using an RTOS and an RT JVM provides the freedom to control the runtime model. For example, we can leverage the RTEMS [6] runtime model, where one process compiled together with the kernel with full utility of all the resources of the underlying hardware, for single app deployment. Using this runtime model is not currently possible with Android, as Android requires most system services to run as separate processes (Section 3.2.3 provides more details). Simply modifying DVM or the OS is not enough to augment Android's runtime model; the framework layer itself must be changed. Since the Fiji VM has mixed criticality support, we can execute system services either as threads within a single-app deployment, or as separate Java programs executing in a partition of a single multi-VM instance.

The third benefit of our architecture is the streamlining of real-time app development. Developers can leverage the rich APIs and libraries already implemented and 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. Thus, Android APIs make it easier to write a single app that can run on different hardware. In addition, Android APIs allow a developer to use all hardware components available on a mobile device, such as: a touch screen, Wi-Fi, GPS, Bluetooth, telephony, accelerometer, camera, etc. Thus, Android compatibility can reduce the complexity of real-time app development.

3.2 Challenges

Providing Android compatibility means that we provide a faithful illusion that an app is running on Android, *i.e.*, we should be able to take an existing Android app and run it on our platform. In addition, a developer should be able to use Android APIs in a real-time app. This is where our main challenge lies; due to the unique programming and runtime model of Android, it requires careful consideration as to how to preserve the semantics of the APIs while using an RT JVM and an RTOS. There are two main concerns that Android introduces though its programming and execution models: 1) Android leverages extensive usage of callbacks – callbacks registered by high priority threads will not necessarily be executed in the high priority thread itself resulting in computation expressed in a high priority task to potentially execute at lower priority and 2) core Android constructs are not priority aware, most use FIFO ordering for processing.

More specifically, we need to consider the following four aspects essential to Android to achieve our goal of providing Android compatibility: 1) supporting the four main components of Android that an app implements, 2) handling Android constructs that an app can use, 3) handling Android system services, and 4) supporting Android's bytecode format, DEX. Two of the challenges—challenges 1 and 4—are not necessarily research directions, but rather engineering tasks. The other two challenges require more careful consideration. In the rest of this section, we discuss these challenges at a high level. In Section 4, we take two specific examples that illustrate these challenges and discuss how we address them within the implementation of RTDroid.

3.2.1 Supporting Android's Four Main Components

The Android programming model provides four main components for constructing apps—`Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`—which are essentially abstract Java classes that define callbacks that an app implements. An Android app needs to extend and implement at least one of these four classes to run on Android. Thus, supporting these four main components exactly the way Android does is critical to providing Android compatibility.

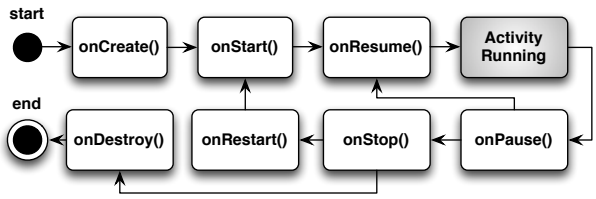


Figure 2: The State Transition for `Activity`.

Briefly speaking [3], an `Activity` class controls the UI of an app. A `Service` class implements tasks that run in the background, *e.g.*, playing background music for an app. A `BroadcastReceiver` class can receive and react to broadcast messages sent by other apps or the Android platform; for example, Android sends out a "low battery" broadcast message to alert apps that the mobile device is running low on battery. Lastly, an app can have a DB-like storage by implementing a `ContentProvider` class.

Each of these four main component classes defines callbacks that an app can implement. The Android platform invokes one of these callbacks at an appropriate time depending on what state the app is in. For example, an `Activity` class defines a number of callbacks such as `onCreate`, `onStart`, `onResume`, `onPause`, `onStop`, etc. When an app starts for the first time, the Android platform invokes `onCreate` and `onStart` in that order. When an app goes out of focus (*e.g.,* when the user pushes the home button or the app switcher button), the platform invokes `onPause`. When an app returns to the foreground, the platform invokes `onResume`. Fig. 2 shows a simplified state transition diagram for an `Activity`. Likewise, all other main component classes define a set of callbacks as entry points to an app. In order to provide Android compatibility, our platform needs to preserve the same execution flows of all the main components.

3.2.2 Supporting Android Constructs

Android provides a set of constructs that facilitate communication between different entities, *e.g.*, threads, main components, and processes. There are four such constructs—`Handler`, `Looper`, `Binder`, and `Messenger`. Since any typical Android app uses these constructs, it is critical for us to support these constructs properly in a real-time context.

We first briefly describe how these constructs work on Android as this is necessary to understand the challenges for supporting these constructs. First, `Looper` and `Handler` jointly enable inter-thread communication. If two threads need to communicate, the receiving thread should first implement `Handler`'s abstract callback method `handleMessage` and instantiate it. Then it should share this `Handler` object with the sending thread, which uses `Handler`'s `sendMessage` or other similar methods to send a message. Internally, `Looper` bridges a `sendMessage` call to its corresponding `handleMessage` call; it maintains a message queue that stores all messages sent by `sendMessage` and dispatches those messages to its `Handler` by calling `handleMessage`. Fig. 3 illustrates the use of `Looper` and `Handler`.

`Binder` is the IPC mechanism of Android that allows different components and processes to communicate. Its usage is similar to a typical IPC mechanism; a process can expose some or all of its methods through `Binder`, which takes care of all aspects of communication such as marshalling and unmarshalling. `Messenger` is another IPC mechanism easier to use than `Binder`; however, it is not a separate construct as the underlying implementation uses `Binder` and `Handler`.

[3]Android Developers website has more detailed information (`http://developer.android.com/guide/components/fundamentals.html`).

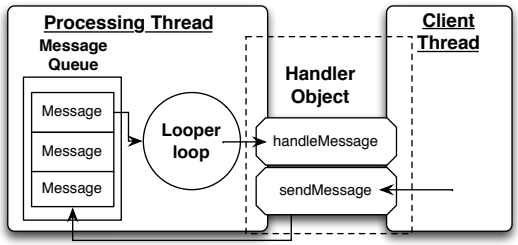


Figure 3: An Example of `Looper` and `Handler`. In the example, the client thread is sending a message to the processing thread. Both threads share the `Handler` object. The processing thread has a `Looper` and a message queue.

There are three reasons why a typical Android app uses these constructs. First, any Android app that implements more than one of four main components discussed in Section 3.2.1 needs to use either `Binder` or `Messenger` for inter-component communication. Second, Android does not allow any long-running operations to run on its `Activity` thread since the `Activity` thread manipulates the UI and long-running operations can render the UI unresponsive. Thus, a typical Android app creates multiple threads to handle long-running operations and leverages `Looper` and `Handler` for inter-thread communication. Third, Android's system services run as separate processes (as we detail in Section 3.2.3) and accessing these services also requires the use of `Binder`.

Among the constructs, the joint use of `Looper` and `Handler` is the ideal first choice to explore the design space in our platform; this is because these constructs are used in all three types of cross-entity communication, either directly (for inter-thread communication) or indirectly (for inter-component or inter-process communication through the use of `Messenger`).

Since `Looper` and `Handler` jointly handle messages, it raises a question for real-time apps when there are multiple threads with different priorities sending messages simultaneously. In Android, there are two ways that `Looper` and `Handler` process messages sent to their thread. By default, `Looper` and `Handler` 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 leveraging `Looper` and `Handler`. If a real-time thread sends a message through a `Handler`, its message will not be processed until the `Looper` dispatches every other message in the queue regardless of the sender's priority as seen in Fig. 4. 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

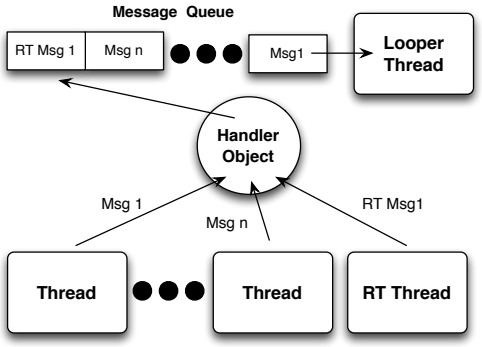


Figure 4: The thread in which the looper executes processes the messages sent through this handler object in the order in which they are received.

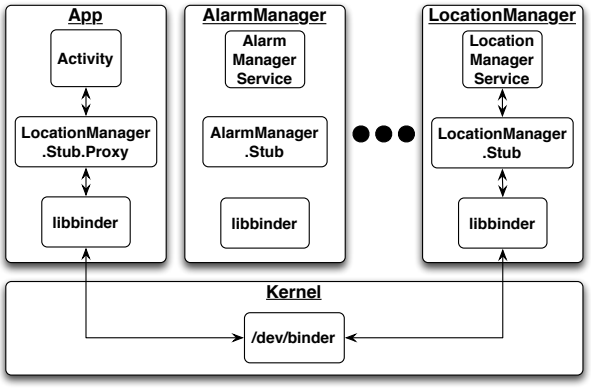


Figure 5: The example shows how an app accesses the `LocationManager` service through the `Binder` IPC.

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.

As a concrete example, consider a health monitoring app that communicates over Bluetooth with a nearby caretaker's device for alert information. Assume a real-time thread is relying on another thread for sending an emergency response request over Bluetooth in the instance it detects a fall. To communicate with this networking thread, a thread must access the networking thread's `Handler` object. However, if other non-real-time tasks also leverage the same networking thread, they can delay the sending of the emergency response request for an amount of time required to process the pending messages. Typically, to handle a message sent to the networking thread requires creation of a packet and the transmission of the packet.

3.2.3 Supporting System Services

Android mediates all access to its core system functionalities through a set of system services. Just to name a few, these services include `ConnectivityManager` that handles network connection management; `PowerManager` that controls power; `LocationManager` that controls location updates through either GPS or nearby cell tower information; and `AlarmManager` that provides a timer service. These system services run as separate processes and the APIs for accessing these services use the `Binder` IPC. Fig. 5 shows an example.

The presence and the access model of these system services raises two questions that our platform needs to answer. First, in our first use case 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 app and the system services as a single process to improve the overall efficiency of the system. Then the question is how to re-design the system service architecture in our platform in order to avoid creating separate processes while preserving the underlying behavior of Android.

The second question is how to mediate access to the system services when multiple threads or processes with different priorities access them simultaneously. Since it involves multiple threads or processes, the question arises not only in our first use case of a single real-time app but also in our second use case of mixed criticality. The default policy for Android is first-come, first-served, which does not consider priorities as a parameter, hence not suitable in real-time systems.

We explore the answers to the above two questions in Section 4 with `AlarmManager` as an illustrative example. `AlarmManager` provides a system timer that triggers at specific times; it receives timer registration requests from different apps and sends a "timer triggered" message to each app when its timer fires. Since it deals with both timing and message delivery, it is the ideal first choice to explore the solution space to answer the above two questions.

3.2.4 Supporting DEX

Android apps use DEX as their bytecode format, which is different from typical Java bytecode [2]. Thus, in order to run existing Android apps on our platform, we need to support DEX. Although this is not necessarily a research challenge, it is nevertheless a challenge to address. Since Fiji VM only supports Java bytecode, our current prototype assumes that the source code is available to re-compile. However, there are other potential solutions; we can either implement DEX support in Fiji VM or leverage existing tools that convert DEX bytecode to Java bytecode, *e.g.*, Dexpler [11] or dex2jar [3]. Exploring these options is our future work.

4. ILLUSTRATIVE EXAMPLES

To discuss our current approach to adding real-time support to Android, we present two examples in this section. The first example is the joint use of `Looper` and `Handler` to illustrate how we support Android constructs. The second example is `AlarmManager` to illustrate how we support system services.

4.1 Looper and Handler

As mentioned in Section 3.2.2, Android's implementation for `Looper` and `Handler` simply processes incoming messages either in the order of reception or at specified times. This can create problematic scenarios for high priority threads where their messages are delayed by messages sent by low priority threads.

To address this problem, we make two design decisions to support real-time. 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 has sent. We are exploring other types of policies in order to provide guarantees depending on the context in which the `Handler` and

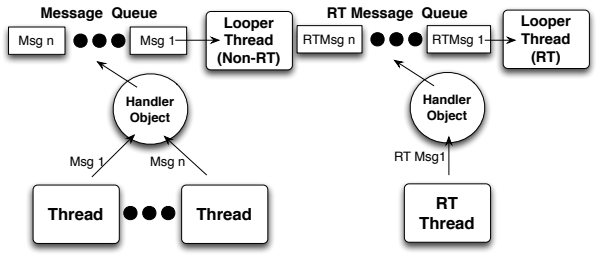


Figure 6: An Example of `Looper` and `Handler` on RTDroid. Each message has a priority and is stored in a priority queue. Processing of messages is also done by priority. The example shows one high-priority thread and multiple non-real-time threads.

`Looper` are leveraged. For instance, we are investigating an earliest deadline first processing scheme for soft-real-time UI updates. Fully exploring priority assignment policies is our future work.

Our second design decision is to create multiple priority queues to store incoming messages according to their priority. We then associate one `Looper` and `Handler` for each queue to process each message according to its priority. Fig. 6 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.

Although our conceptual model leverages one thread per priority level, the implementation of these constructs is backed by a thread pool. We adopt a similar approach to implementing efficient Asynchronous Event Handler (AEH) in RTSJ, proposed by Kim *et al.* [16]. To provide memory safety we limit the size of the queue. We are currently investigating which queue management schemes and size limitations are appropriate for which service.

4.2 AlarmManager

The `AlarmManager` is an ideal candidate to discuss how we address the following two questions for system services: 1) how to adapt Android's multi-process architecture when running a single real-time app, and 2) how to mediate access to a system service from multiple threads with different priorities. We first briefly overview how `AlarmManager` works and discuss our approach.

Android's implementation for `AlarmManager` involves alarm registration and alarm delivery as shown in Fig. 7. When an app registers an alarm, it makes an IPC call to `AlarmManager` with a message and a time. The message is associated with a callback of the app which gets executed when the message is delivered [4]. When the alarm triggers at the specified time, `AlarmManager` sends the message back to the app, and the callback gets executed. In both the registration and the delivery, Android provides no guarantee on when or in what order the message is delivered.

Thus, we re-design both registration and delivery of alarms to support predictability. For alarm registration, we use red-black trees to maintain alarms as shown in Fig. 8; 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; then we use these per-timestamp trees 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 ensure 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.

[4]This is done by using Android's `Intent`, though we do not discuss the details here due to space considerations.

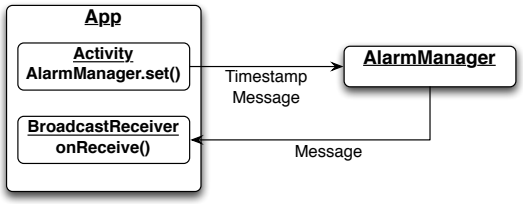


Figure 7: An Example Flow of `AlarmManager`. An app uses `AlarmManager.set()` to register an alarm. When the alarm triggers, the `AlarmManager` sends a message back to the app, and the app's callback (`BroadcastReceiver.onReceive()` in the example) gets executed.

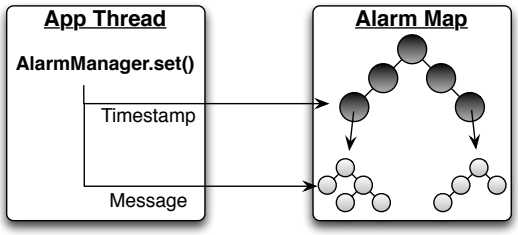
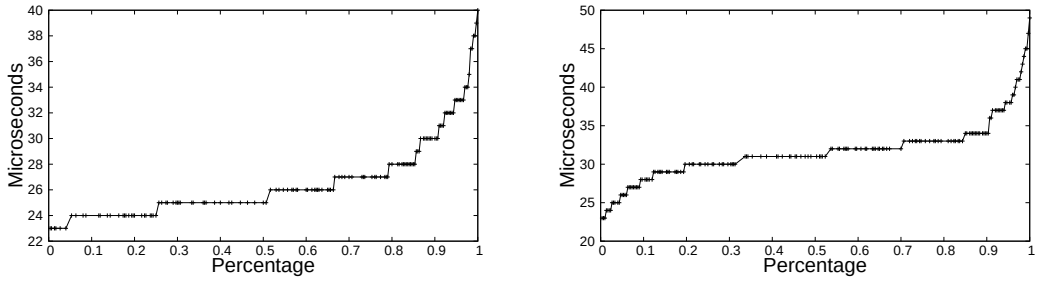


Figure 8: The Implementation of Alarm Registration on RTDroid. The tree colored black at the top maintains timestamps; the trees colored gray are per-timestamp trees maintaining actual alarm messages to be delivered.

For alarm delivery, we create an `AlarmManager` thread and assign the highest priority for timely delivery of alarm messages. With this thread, we replace the multi-process message passing architecture of the original `AlarmManager` with RTSJ's AEH. More specifically, the thread wakes up whenever an app inserts a new alarm to our red-black trees; then it schedules an AEH at the specified time for the alarm. We associate the app's callback for the alarm message with this AEH, so that we execute the callback exactly at the time specified in the alarm. In practice, most Android apps leverage only a few alarms. However, we are exploring an alternative approach where we create one AEH per priority level and leverage a thread pool.

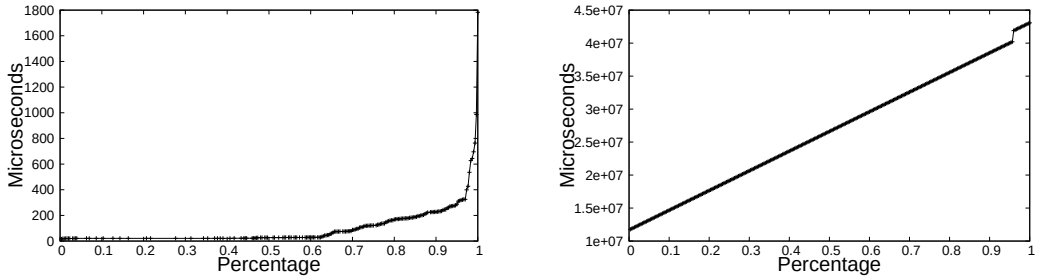
5. EVALUATION AND RESULTS

To measure and validate our prototype of RTDroid, we tested our implementation on two system level configurations. The first configuration utilizes an Intel Core 2 Duo 1.86GHz Processor with 2GB of RAM running Linux patched with RT Linux v.3.4.45. For precise timing measurements we disabled one of the cores prior to running the experi-



(a) With 30 low priority threads (b) With 300 low priority threads

Figure 9: The cumulative distribution of latency for RTDroid at 30 and 300 low priority threads configuration on Core 2 Duo running RT Linux.



(a) With 30 low priority threads (b) With 300 low priority threads

Figure 10: The cumulative distribution of latency for Android at 30 and 300 low priority threads configuration on Core 2 Duo running RT Linux.

ments.

For the second configuration we leveraged a stock LEON3 embedded board manufactured by Gaisler. The experiments were run on a GR-XC6S-LX75 LEON development board [5] running RTEMS version 4.9.2. The board's Xilinx Spartan 6 Family FPGA was flashed with a LEON3 configuration running at 50Mhz. The development board has an 8MB flash PROM and 128MB of PC133 SDRAM.

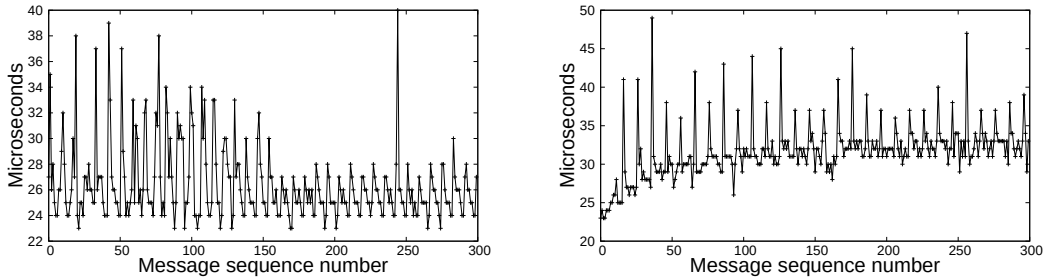
In order to evaluate the performance of our approach, we ran our experiments comparing two different configurations of `Looper` and `Handler`. The first configuration (Android) is a direct port of the stock Android Implementation and mirrors the previously proposed system architectures given in Fig. 1c and Fig. 1d. The second configuration (RTDroid) is our proposed extension.

5.1 Experiments

To measure the effectiveness of our prototype, we constructed an experiment that leveraged `Looper` and `Handler`. Since these are two of the core constructs for inter-thread, inter-component, and inter-process communication, showing the predictability of this construct is crucial. Our microbenchmark creates one real-time task with a 100ms period that sends a high priority message. To measure the predictability of the system, we calculate the latency of processing this 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 `Looper` responsible for processing this message after the message has been received and the appropriate callback invoked. The difference between the timestamps is the messages latency. In addition, the experiments include a number of low priority threads which also leverage the same resource though the same `Handler` object. These threads have a period of 10ms and send 10 messages during each period.

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 thread, and have a hard stop at 50 seconds. We measure latency only for the high priority messages. We 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. We have varied the number of low priority threads in increments of 10 from 10-100 and in increments of 100 from 100-300 when running the experiments on the Intel Core 2 Duo running RT Linux. 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. Saturation of the message queue occurs

[5]Additional board specification can be found at Gaisler's website: `www.gaisler.com`.



(a) (b)

Figure 11: The raw latency measurement for RTDroid at 30 and 300 low priority threads configuration on Core 2 Duo running RT Linux.

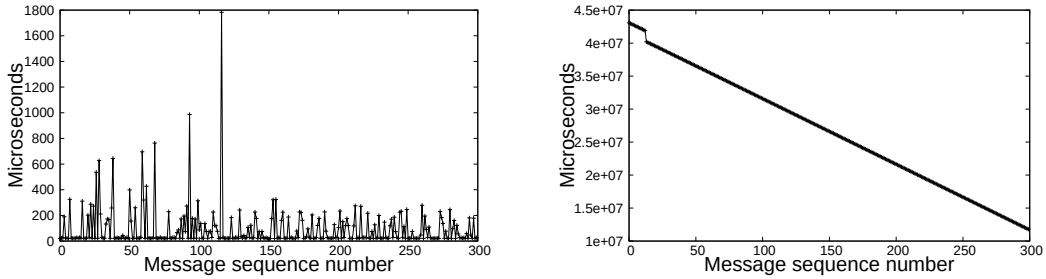


Figure 12: The raw latency measurement for Android at 30 and 300 low priority threads configuration on Core 2 Duo running RT Linux.

at around 15 low priority threads. Besides the measurement of high priority message, we also calculate the overall message throughput to test the broken point for the both of the Android and RTDroid in the same scenario.

The data is discussed both in aggregate as well as introspecting a given run on each hardware platform. We present three different types of plots: 1) the cumulative distribution of the message processing latency for each release of the high priority thread with differing numbers of low priority threads using the same `Handler`, 2) the raw latency observed in handling messages from high priority threads, and 3) the overall throughput of RTDroid compared to Android. Raw data gathered from the experiments as well as scripts to generate the graphs are available at: `http://RTDroid.cse.buffalo.edu`.

5.2 Result Overview

We first present a summary of the results and show a more detailed view in the following subsection. Fig. 9 and Fig. 10 show the overall cumulative latency distribution. We observe that the bounds of latency vary with increasing low priority threads in Android, but stays relatively constant for RTDroid; Fig. 9a and Fig. 9b show that all messages experience latency between 23μs and 49μs regardless of the number of threads on RTDroid. Moreover, the observed worst case latency increases only by 9μs even when the number of threads is increased by an order of magnitude from 30 to 300. This increase is attributed to the fact that there is a greater probability of context switching from a low priority thread to the high priority thread and also the increased contention on the scheduling queue with a higher number of threads. On the Android configuration, we observe an interesting phenomena. When the number of low priority threads is relatively small with 30 threads in Fig. 10a, 60% of the messages have similar latency to that of RTDroid, but there is a high spike in latency in the last 10% of messages. Further, the difference between the best latency and the worst latency is almost two orders of magnitude. Once the number of low priority threads increases to 300 in Fig. 10b, the latency distribution becomes almost linear with a very large difference between the best case and the worst case latency. This occurs because before the first release of the high priority thread, the queue is already saturated. We provide more details of the phenomena in the following subsection.

5.3 Raw Latency

In the latency plots for Android implementation, we observe that message processing delays vary arbitrarily, with increase in latency being greater than an order of magnitude in some cases (see Fig. 12). These plots clearly show the lack of real-time processing guarantees. The bound on latency is loose and there is no consistent or uniform latency distribution. The raw timing numbers do provide an explanation for the phenomena we observed in the Android case when examining the cumulative distribution plots. Notice that the initial messages have the greatest latency and this decreases over time. What is occurring is that the low priority threads fill the message queue faster than the messages

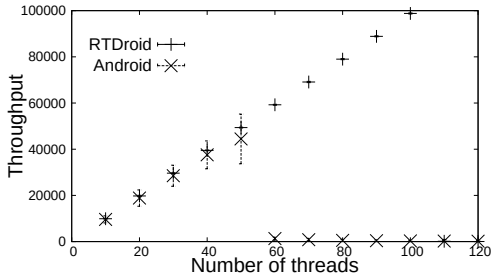


Figure 13: A throughput comparison of Android and RTDroid scaled by the number of low priority threads on Core 2 Duo running RT Linux.

can be processed, causing large initial delays. Eventually the processing thread catches up as the low priority threads complete their workloads.

Compared to the Android implementation, RTDroid shows significantly better consistency in terms of message delay bounds. More than 90% of the messages experience the latency between 22μs and 50μs with any number of threads, with variance of around 20μs from the lowest to the highest latency for those messages in any given run. Further, the largest latency variance is 26μs. This variance is attributed to context switch costs and scheduling queue contention. We have isolated these delays by profiling the execution times of the threads themselves. For Android roughly 6% of messages have similar latencies to RTDroid, however there are significant spikes in the remaining 40% of messages. The standard deviation for Android is 1600μs for a reasonably loaded system and $2 * 10^7$ μs for an overloaded system (additional plots are provided in the accompanying technical report [22]).

5.4 Throughput

To measure the cost associated with our solution we compared the throughput of the Android configuration to RTDroid given in Fig. 13. To calculate the throughput we measured the rate in which all messages, both from high and low priority threads, are processed. We observe that for the two configurations, the throughput is roughly equivalent with low numbers of threads. This result is not unexpected as the primary overhead in RTDroid is the use of a `Handler` object by a thread with a priority that has not communicated through this `Handler` before. In such situations, a new queue and `Looper` thread are created for this priority. We observe that the system can be pre-configured and these resources allocated at boot time, but the current implementation does this dynamically. We also note that at around 100 low priority threads, we create enough contention on the message queue and scheduling queue that throughput decreases dramatically. Note, however, that even in extreme cases of contention the latency of high priority thread's messages is only increased by the cost of preempting the low priority thread. For the Android configuration, this dramatic drop occurs before the RTDroid configuration at around 50 low priority threads. This happens because the high priority thread introduces enough additional preemption to prevent the low priority `Looper` from being starved by the low priority threads, which allows for additional throughput scalability. Based on our experiences with Android applications it is highly unusual for any app to leverage more than a few threads that interact with a given component. As such, these experiments serve to illustrate the maximum loads the system can realistically handle. We expect the throughput of the Android configuration to mirror that of RTDroid, if additional threads were present in the system that did not leverage the `Handler`, thereby increasing preemptions and reducing overall per unit time contention on the message queue.

5.5 LEON3

The LEON3 results are summarized in Fig. 14, for the default RTEMS configuration leveraging 10 threads. We note that the development board configuration can only report timing results at a granularity of 400 μs since it does not have a timing register and the hardware clock frequency cannot be adjusted. Based on the timing granularity, our LEON3 results mirror the results obtained on the Intel Core 2 Duo.

6. RELATED WORK

Recent work has performed prelimnary 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 [17]. 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 performace and predictability of DVM in a real-time setting was first characterized by Oh *et al.* [18]. Their findings mirror our general observations on Android. In many situations Android is quite performant. However, 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, and system services built from them.

Kalkov *et al.* [15] 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. Although, Kalkov *et al.* proposed more modest changes to the Android system, focusing on modifying DVM's GC to provide real-time capabilities, they have not yet explored how different components within a single app (or across multiple apps) 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. Indeed, Android's specialized programming model and execution provide many design and implementation challenges. We believe our implementation is synergistic to such proposals and can be leveraged to provide predictability when apps leverage services, IPC, or the core Android constructs.

7. CONCLUSION AND FUTURE WORK

In this paper we have shown that replacing DVM with

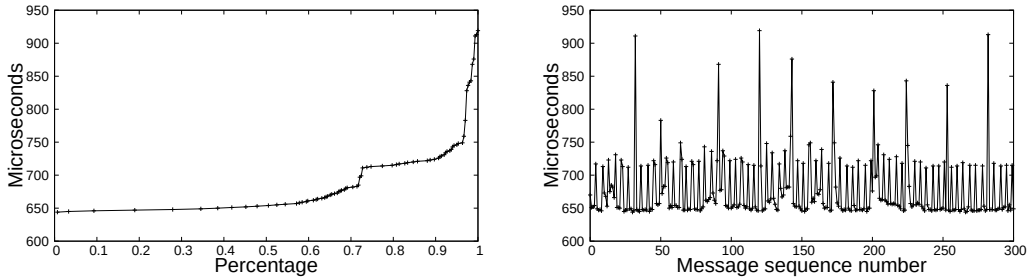


Figure 14: The cumulative distribution and raw measurement of latency for RTDroid at 10 low priority threads configuration on LEON3 running RTEMS.

a RT JVM and Linux with an RTOS is insufficient to run Android application with real-time guarantees. To address this shortcoming of the proposed real-time Android models, 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 about to adapt the solutions that we mentions in Section 4.2 to make the `AlarmManager` more efficient. Then we will move toward a 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.

8. 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] .dex — Dalvik Executable Format. `http://source.android.com/tech/dalvik/dex-format.html`.

[3] dex2jar. `http://code.google.com/p/dex2jar/`.

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

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

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

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

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

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

[10] B. Buzbee B. Cheng. A JIT Compiler for Android's Dalvik VM. `http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html`.

[11] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android Dalvik bytecode to Jimple for static analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM.

[12] D. Bornstein. Dalvik VM internals. `http://sites.google.com/site/io/dalvik-vm-internals`.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4):2:1–2:36, January 2008.

[22] Yin Yan, Sree Harsha Konduri, Amit Kulkarni, Varun Anand, Steven Y. Ko, and Lukasz Ziarek. [technical report] RTDroid: A Design for Real-Time Android. `http://rtdroid.cse.buffalo.edu/techreport/rtdroid.pdf`.