RTDROID: A REAL-TIME SOLUTION ON ANDROID

by

Yin Yan

September 1, 2018

A dissertation submitted to the Faculty of the Graduate School of the University at Buffalo, State University of New York in fulfilment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering

The thesis of Yin Yan was reviewed by the following:

Dr. Lukasz Ziarek, *lziarek@buffalo.edu*
Assistant Professor at Department of Computer Science and Engineering
University at Buffalo, State University of New York
Major Advisor,

Dr. Steven Y. Ko, *stevko@buffalo.edu*
Associate Professor at Department of Computer Science and Engineering
University at Buffalo, State University of New York
Committee Member

Dr. Karthik Dantu, *kdantu@buffalo.edu*
Assistant Professor at Department of Computer Science and Engineering
University at Buffalo, State University of New York
Committee Member

Acknowledgments

My sincere thanks to Steven Y. Ko introduces me to Luke. Both of them helped me to start the RTDroid project and guided me throughout my PhD completion. I would also like to thank Karthik Dantu, the other committee member, for his support and patient, especially the help on my academic writing. From Luke, Steve and Karthik I learned how to be a researcher with critical thinking, responsible and detail-oriented towards research results.

My sincere thanks to Steven Y. Ko introduces me to Luke. Both of them helped me to start the RTDroid project and guided me throughout my PhD completion. I would also like to thank Karthik Dantu, the other committee member, for his support and patient, especially the help on my academic writing. From Luke, Steve and Karthik I learned how to be a researcher with critical thinking, responsible and detail-oriented towards research results.

I wish to thank Prof. Jan Vitek and Dr Ethan Blanton who provided external technical guidance and support to the RTDroid project, as well as Prof. Antony Hosking, my mentors during my internships at Data61, gave me a chance to work at a world-class research lab.

Extra special thanks to the other students who were working with me together on my research projects, including Shaun Cosgrove, Varun Anand, Amit Kulkarni, Sree Harsha Konduri, Chunyu Chen, Ji Zhang, Adam Czerniejewski, Sai Tummala, Manish Jain. It was a very good time for the cooperation.

I am grateful to my parents and family for always encouraging me to pursue my ideas and desires and for their unfailing support and love throughout this long journey that now seems to have passed too quickly.

Lastly, I am eternally grateful for the love and support of my wife, Lan. She helped me endure the many frustrations associated with the completion of my PhD study. But most of all, she makes me very happy.

Table of Contents

List of Tables

List of Figures

Abstract

Since the introduction of the smartphone, mobile computing has become pervasive in our society. Meanwhile, Mobile devices have evolved far beyond the stereotypical personal devices. Smartphones and tablets and have been employed in various traditional real-time embedded domains. Of the currently available mobile systems, Android has seen the most widespread deployment outside of the consumer electronics market. Its open source nature has prompted its ubiquitous adoption in sensing, medical, robotics, and autopilot applications. However, it is not surprising that Android does not provide any real-time guarantees, since it is designed as a mobile system and optimized for mobility, user experience, and energy efficiency.

Although there has been much interest in adopting Android in real-time contexts, surprisingly little work has been done to examine the suitability of Android for real-time systems. Existing work only provides solutions to traditional problems, including real-time garbage collection at the virtual machine layer and real-time OS 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.

This proposal goes beyond existing work and examines the internals of Android. We discuss the implications and challenges of adapting Android constructs and core system services for real-time and present a solution for each layer of the stock Android. Our system is unique in that it redesigns Android's internal components, replaces Androids Dalvik/ART with a real-time VM, and leverages off-the-shelf real-time OSes. Additionally, RTDroid also provides an event-driven programming model for real-time Android development, including four major components: 1) real-time constructs to express application logic, 2) real-time extension to Android's application manifest to specify real-time configuration, 3) real-time communication channels to enable constructs interactions with real-time semantics, 4) pause-less memory management for mem-

ory guarantees with scoped memory.

However, the use of scoped memory introduces extra design complexity and performance overhead for the management of scoped memory areas. One of our ongoing work is to define RTDroid specific scope definition with precise scope lifetime. Likewise, since RTDroid's programming model hides the scoped memory areas in its framework implementation, we are exploring the possibility of simplifying the scope management at runtime with runtime techniques. Meanwhile, to test the portability of our system design, we are planning to enable RTDroid's application on a micro virtual machine.

Chapter 1

Introduction

1.1 Real-Time Application in Mobile Computing

As one of the most popular mobile system, Android has been evolved far beyond a mobile system, a number of real-time embedded applications have been developed as Android applications (apps). For example, the medical industry has attempted to use Android devices as a potential platform that facilitates the development and the deployment of various medical applications, ranging from the critical care for vital signs of patient monitoring [24, 15, 6, 1], the handicap assistant [8, 2]. The defense industry is also exploring Android for a variety of soldier support roles, including portable navigation systems [57, 58] and wounded soldier detection [36, 78]. Similarly, the aerospace industry has experimented with Android as a control system for satellites [81].

There are numerous benefits of developing these applications in Android. From user perspective, the personalized device nature of Android smartphone encourages users to keep their device with their body, while deploying the applications on a personal device reduces the number of user carry-on devices. From the programming perspective, developers can leverage Android's rich set of APIs to utilize new types of hardware such as cameras, sensors and touch screens. Additionally, Android's well-supported development environment eases the burdens of the application development, and many applications published in the Google's online store give an opportunity to incorporate creative functionalities with less effort. However, all of these advantages co-exist with the fact that Android does not provide substantial dedicated support for real-time features or safety-critical concerns.

Many researchers have evaluated the real-time support on the stock Android [59, 62, 65]. It is not surprising that Android doesn't provide any real-time guarantees since it is designed as a mobile operating system which aims for smooth user experience, longer battery lifetime *etc*. To our best knowledge, Maia *et al.* are the first ones that propose system architectures for real-time Android. They have suggested to utilizing a real-time system kernel and a real-time Java Virtual Machine (JVM) for real-time tasks scheduling and predictable garbage collection (GC), which are fundamental to make Android real-time [51]. A real-time extension of Android's DalikVM has been developed by Kalkov *et al.* that reduces GC pauses duration the execution of time-constrained constructs [40]. However, the fundamental question of how to add real-time support to Android has never been explored systematically.

This dissertation address this question *as a whole system* and presents the solution as a real-time extention on Android, named RTDroid. The design of RTDroid aims to provide bounded response times for the real-time task requires strict timing guarantees, while still developers to use existing functionalities on Android for non-real-time requirement in real-time applications.

1.2 Overview of Contributions

This dissertation makes three major contributions. First of all, RTDroid proposes a system architecture for real-time mobile computing by utilizing a real-time OS kernel, a real-time Java Virtual Machine (JVM) and redesigned real-time constructs as core components in its framework. Secondly, RTDroid introduces a real-time programming model that adds real-time semantics for real-time application development, explores a pauseless region-based memory management techniques in its runtime and enables component interactions via its real-communication channels. Thirdly, RTDroid provides a static application validation mechanism which serves as a starting points to the support of downloading and installing new application at the system runtime. It check the real-time configuration of a newly downloaded application with regard to the schedulability of other apps present in the system.

From a system designer's point of view, RTDroid makes the following contributions as listed by each layer in its software stack.

◇ **Kernel Layer:**

- Enables real-time features on specific device with real-time OS support: RT-Linux on Nexus S or RTEMS on Leon3 development board.
- Identifies incompatible kernel modules in RT-Linux.

◇ **Runtime/VM Layer:**

- Enables a real-time JVM, FijiVM [63], on Android-equipped smartphones.
- Implements data transfer mechanism for RTDroid channels.

◇ **Framework Layer:**

- Provides real-time components with scoped memory, *i.e.*, real-time service, real-time receiver and periodic task for the application development.
- Implements real-time communication channels for various real-time communication semantics.
- Extends Android's manifest schema for real-time configurations.

◇ **Application Layer:**

- Implements real-time system services for research needs, *e.g.*, `RealTimeAlarmManager`, `RealTimeSensorManager` and `RealTimeAudioManager`.
- Integrate Cheddar with the compilation process for RTDroid's applications for the validation of real-time configurations.

1.2.1 System Architecture: RT OS, RT JVM and RT Framework

Our first attempt is to analyze real-time capabilities of stock Android and its limitations by examining the entire Android software stack. We have identified that not only the Android's customized Linux kernel and Java virtual machine

(DalvikVM or ART) are needed to be replaced with real-time counterparts, but Android's internal constructs in the framework layer are also problematic. For example, Android's core communication constructs—`Looper` and `Handler` for message passing—process their incoming messages in FIFO ordering, which results in arbitrary processing latencies. RTDroid's [86, 85, 84] propose a clean-slate system architecture that leverages an existing real-time system kernel, an "off-the-shelf" real-time JVM and redesigned constructs for *a single real-time application execution environment*. It provides real-time predictability in terms of timing, memory and resource management. Even focusing on *a single real-time application execution*, this goal alone has many hard challenges associated and still has broad applicability in utilizing smartphones for real-time usages such as control, medical, and military devices. It is also a prerequisite to the further development of RTDroid for later discussion of RTDroid's programming model.

1.2.2 Real-Time Programming Model

From the programming perspective, RTDroid provides an event-driven programming model with declarative constructs and pause-less memory management. To preserve Android-like programming fashion, RTDroid's constructs are directly extended from Android's application constructs. It also extends Android' manifest schema to express real-time properties for timeliness and other resources requirement, so that it separates construct logic and real-time properties. Such separation decouples real-time requirement from the actual application functionalityand reduces the complexity of app development. To enable interaction between constructs, RTDroid's programming model defines three priority-aware communication channels, including real-time message passing channel, real-time intent broadcasting channel, bulk data transfer channel and cross-context channel. These channels define semantics of interactions between real-time and non-real-time context. For real-time feature, RTDroid's programming model is enlightened lessons learned from the Real-Time Specification of Java (RTSJ) as well, and adapts them to the realities of the Android ecosystem. The past experience with real-time computing in Java [39] suggests that real-time garbage collection works well when it is possible to predict and bound the allocation rate of the program. RTSJ offers a way to avoid garbage collection using the scoped memory API. The challenge with scoped memory was that it was a pervasive change to the infrastructure which proved incredibly difficult to use with legacy code. RTDroid's programming model uses scoped memory for memory management but hides the complexity of using scoped memory area in its framework implementation.

1.2.3 Real-Time Application Validation

Based on the real-time programming model, RTDroid provides an off-line validation mechanism for its application. It utilize real-time properties declared in RTDroid's application manifest for scheduling analysis and feasibility tests. RTDroid's compiler converts its application manifest to a format that understood by a real-time scheduling framework, named Cheddar [71]. We leverage Cheddar's built-in scheduling simulation and feasibility tests to validate whether an application is configured correctly to meet its timing constraints in term of response times, execution rates and the number of missed deadlines. Such validation mechanism serves two purposes: (1) It provides an automatic solution for application developers to check if the application configuration is valid. (2) It provides a starting point for investigating a hybrid method for the worst case execution time, which is critical for deplying a newly downloaded app at runtime.

Chapter 2

Background and Related Research

This chapter presents an overview of Android architecture and discusses existing research that has explored real-time capabilities in Android. Given Android is a vast system, we limit ourselves to the work that is essential to understand the topic of this dissertation, mainly focusing on the system architecture and the application programming model. For the real-time system, we acknowledge the contributions that have studied thoroughly on real-time theories and real-time systems since the early 1990s and 2000s. Within the scope of this dissertation, we emphasise on the latest relevant works targeting to use mobile systems in the real-time context. This chapter is organized into two sections: The first section briefly introduces the architecture of the stock Android while highlighting challenges of using Android for real-time systems. The later chapter 3 and 4 will discuss these challenges in details, respectively. Then, The rest of section studies existing work that has been looking into enable real-time ability on Android.

2.1 Background to Android

Since its inception, Android is designed as a mobile system, which builds upon a modified Linux kernel and executes applications in its Java Runtime Environment. As a mobile system, Android is optimal for connectivity, mobility, friendly user experience and long battery life *etc*, rather than the predictability

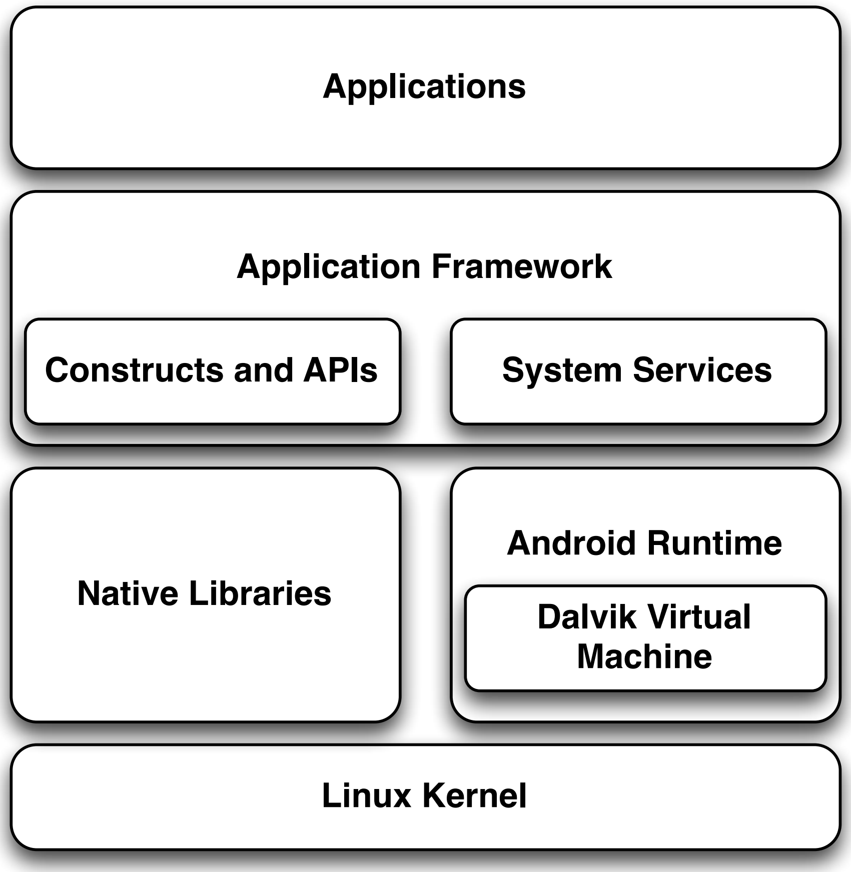


Figure 2.1: Simplified Architecture in Android

or the deterministic. It is important to introduce necessary knowledge about Android before we dive into the RTDroid project.

2.1.1 The Linux Kernel and Java Runtime Environment

Android's system architecture consists of many layers from bottom to top, consisting of an operating system kernel layer, a Java runtime environment (Davlik/ART), an application framework layer and an application layer, as Figure 2.1 shows.

At the bottom-most layer, a modified Linux is used as Android's operating system kernel. Its kernel has been configured for mobile devices. There are a number of components are unique to the mobile system. For example, the Android's Linux kernel schedule uses a completely fair scheduler as the default processes scheduler. Such scheduler aims to higher overall CPU utilisation while maximising interactive performance. Such scheduler defines the importance of a task as the activeness of tasks, instead of the priority of tasks. Similarly, Android doesn't have a swap partition. Instead, it has an out of memory (OOM) killer that enforces recycling memory by terminating less active *apps* when the available memory is lower than a threshold.

Android executes its *apps* in separate JVMs. Rather than using the traditional Hotspot JVM, Android has its own JVM, known as DalvikVM (DVM), which replaced by ART [16] later. DVM is a register-based virtual machine [70] that executes DEX bytecode [3] and uses just-in-time compilation (JIT) [10] to optimize apps at runtime. Kalkov *et al.* [40] observed that DVM's garbage collection mechanism could pause all application thread until GC is finished. To avoid the suspension from GC, they have extended DVM's GC with new APIs, which allow developers to trigger GC as needed in their applications. Although this design decision seamlessly integrates into applications, it and relied on developers to achieve predictability, which adds a layer of complexity to application development. Since Android 5.0 (*Lollipop*), ART has replaced DVM as Android's new JVM. It is equipped with an Ahead-of-Time (AOT) compiler for the native optimization and improved concurrent garbage collectors.

These optimisation aims for better performance in the average case, but not the worst case which is essential to a real-time system. Many works have evaluated the real-time capability on Android [59, 62, 65, 52]. They have concluded that Android's kernel and runtime implementation provide no real-time guarantees and its garbage collector can arbitrarily stall application threads regardless of priority, resulting in non-deterministic behaviour. Thus, it is well-understood that the bottom two layers need real-time support to provide a predictable platform. However, it is still not sufficient to provide real-time capability on Android even with the proper real-time features at the kernel and runtime layers. This is due to the fact that the nature of Android's event-driven programming model in the application framework layer.

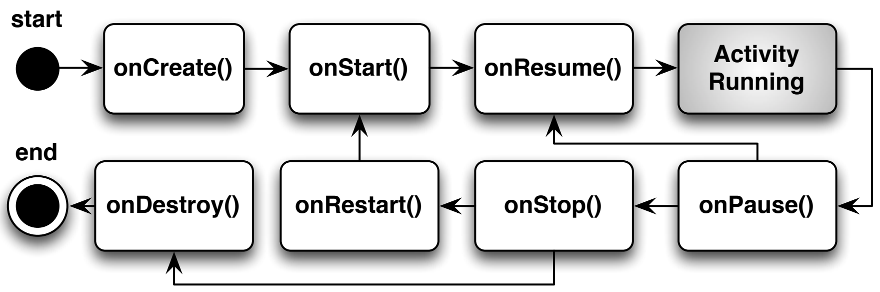


Figure 2.2: The State Transition for `Activity`.

2.1.2 The Event-Driven Programming Model

Android applications are an event-driven, all components are defined as a set of callbacks and executed via messages. Its unique programming and execution model requires careful consideration as to how we can support Android compatibility while providing real-time capabilities. This section focus on explains the characteristic of Android components while discussing challenges of using them in real-time context at a high level. The later chapter 4.1 will details challenges with a concrete example and more in-depth analysis.

Application Components: There are four main components for implementing applications—`Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`—which are essentially abstract Java classes that define callbacks. An application needs to extend and implement at least one of these four classes to run on Android.

Briefly speaking [1], 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 database-like storage by implementing a `ContentProvider` class.

Each of these four main component classes defines callbacks that an application can implement. The Android platform invokes one of these callbacks at an appropriate time depending on what state the application is in. For instance,

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

an `Activity` class defines a number of callbacks such as `onCreate`, `onStart`, `onResume`, `onPause`, `onStop`, etc. When an application starts for the first time, the Android platform invokes `onCreate` and `onStart` in that order. When an application 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. 2.2 shows a simplified state transition diagram for an `Activity`. Likewise, all other components define a set of callbacks as entry points to an application.

System Services: Android also excels in supporting a wide variety of hardware with different CPUs, memory capacities, screen sizes, and sensors. Android's APIs also ease the access of onboard devices in applications, such as the touchscreen, Wi-Fi, GPS, Bluetooth, telephony, accelerometer, camera, *etc*. Android mediates all access to its core system functionalities through a set of system services. 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.

As a concrete example, consider a health monitoring app that samples vital indicators of a patient and transfers data to a nearby caretaker's device for alter information via Android's system service, `BluetoothManager`. Assume a critical component, *an Application Service*, posts an alarm message to the caretaker's device via `BluetoothManager` when a vital indicator is below a threshold. However, other non-critical components leverage the same `BluetoothManager` to send messages as well. Thus, these non-critical can delay the delivery of the emergency messafe for an amount of time required to process the pending messages. Typically, to handle a message sent to `BluetoothManager` requires creation of a packet and the transmission of the packet.

As the above senario, both of application components and system services interact with others via messages and asynchronously execute the message processing as callbacks. By default, system services run as separate processes, an application component *must* leverage APIs for accessing them are implemented via RPC—Android's Binder calls. There are two main concerns that Android introduces through its programming and execution models: 1) Android leverages extensive usage of callbacks—a registered callback executes in the context of its caller, which may lead to non-deterministic and unbound execution latencies; 2) The core communication of Android processes incoming messages in FIFO order.

Given to these difficulties, most of the research that uses Android in the real-time domain is limited to enable real-time guarantee in a specific domain. For example, Moazzami *et al.* [53] have built ORBIT as a smartphone-based sensing platform using Android. Kim *et al.* [42] have established SounDroid that enables a high degree of QoS for audio-based applications. The dissertation proposed a general programming mode for the real-time development in the chapter 4. The rest of this chapter discusses existing work relevant research in real-time mobile systems.

2.2 Real-Time System

The real-time scheduling theory [49, 67, 46, 68] has been thoroughly studied for the past several decades. It originates from the need to control physical process in complexity from automobile ignition system or controllers for flight system and nuclear power plants. Such systems are referred to as *hard* real-time systems, where time constraints must be met at all cost. The violation of time constraints may lead to human or material damage. On the other hand, if it is desirable to meet a tasks deadline but occasionally missing the deadline can be tolerated, then these systems are so-called *soft* real-time systems. For example, the flight navigation for an avionics system, the sensor data processing in autonomous vehicles, and the signal processing in radio communication systems. The scheduler for these systems must coordinate resources to meet the timing constraints of the physical system. This implies that the scheduler must be able to predict the execution behaviour of all tasks within the system. This basic requirement of real-time systems is **predictability**. Unless the behaviour of a real-time system is predictable, the scheduler cannot guarantee that the computation deadlines of the system will be met.

To prove the satisfaction of time constraints, a scheduler needs to know the upper bounds of execution times in dynamic systems. In fact, it is hard to obtain

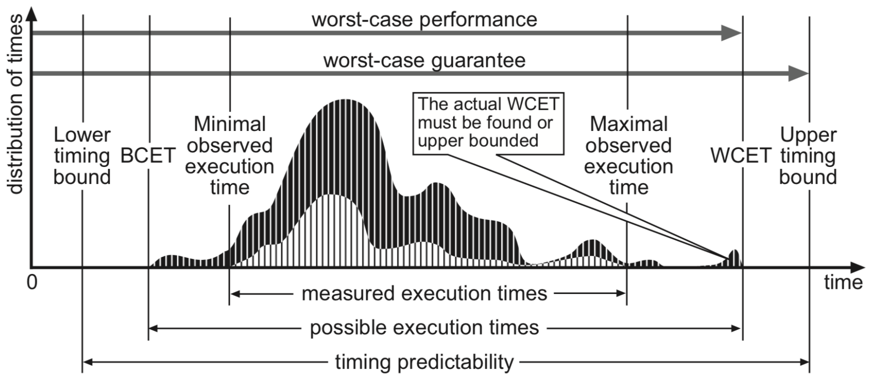


Figure 2.3: Basic notions concerning timing analysis of systems. The lower curve represents a subset of measured executions. Its minimum and maximum are the minimal observed execution times and maximal observed execution times, resp. The darker curve, an envelope of the former, represents the times of all executions. Its minimum and maximum are the best-case and worst-case execution times, resp., abbreviated BCET and WCET.

such upper bounds for programs. Otherwise, we would know how to solve the halting problem. For a real-time system, we generally have a priori known number of tasks in the system, but the worst-case execution time is still not known and hard to derive. It is a separate research topic which is out of the scope of this work. The research is dedicated to exploring different techniques to produce task properties, e.g, worst-case execution times for timing analysis. As Figure 2.3 shows, the execution times of a task has a range of spectrum depending on its input arguments as well as environment setups. For instance, all execution times is shown as the upper curve. The shortest execution time is called the best-case execution time (BCET), the longest time is the worst-case execution time (WCET). In most cases, the state space of execution times is too large to exhaustively enumerate all possible executions with different input data. For timing analysis, this work focuses on system schedulability, feasibility analysis and etc and uses worst-case response time since we have a mixed criticality environment with shared resources.

2.2.1 Real-Time Tasks

In real-time scheduling, a real-time task can be categorized into one of three types given to its release pattern and deadline [55]. A task with a regular release time is a **periodic task**, which is commonly used to process data and update the current state of the system on a regular basis. At the system runtime, an execution of the periodic logic of the task is considered as a job release, also known as a *release*. The second type of task has an irregular release time, so-called an **aperiodic task**, it is typically used to handle the processing requirements of random events such as operator requests. The third type of task is a **sporadic task**, it is an aperiodic task with a fixed release interval. As definitation in [73], we have the following three types of real-time tasks:

- ◇ **Hard and Soft Deadline Periodic Tasks**: A periodic task has a regular interarrival time equal to its period and a deadline that coincides with the end of its current period. Periodic tasks usually have hard deadlines, but in some applications, the deadlines can be soft.
- ◇ **Soft Deadline Aperiodic Tasks**: An aperiodic task is a stream of jobs arriving at irregular intervals. Soft deadline aperiodic tasks typically require a fast average response time.
- ◇ **Hard and Soft Sporadic Tasks**: A sporadic task is an aperiodic task a minimum release time.

2.2.2 Real-Time Scheduling

Most of the real-time scheduling algorithm is based on the result of Liu *et al.* [49] that shows any set of independent periodic tasks is schedulable by the rate monotonic algorithm if the condition of Theorem 1 is met.

Theorem 1. *A set of n independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if*

$$\frac{C_1}{T_1} + ... + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) = U(n)$$

where C_i and T_i are the execution time and period of task respectively.

Liu *et al.* [49] present a scheduling bound for the rate monotonic algorithm under the worst case scenario, but the set of periodic tasks must be statically configured. Later, Lehoczky *et al.* [48] performs stochastic analysis for the rate monotonic algorithm in which the set of periodic tasks are randomly generated, and conclud that the scheduling bound can be relaxed to approximately 88% system utilization in the average case. Later, Sha *et al.* [69] propose a period transformation method, and approved that the utilization threshold can close to 100% in theory. For instance, Borger *et al.* [75] demonstrated the Navys Inertial Navigation System with a schedulable utilization level of 99%.

Another concern for scheduling algorithms is transient overload when periodic tasks are randomly generated, the stochastic execution can boost the desired system utilization greater than its scheduling bound. Sha et al. described their period transformation to guarantee the deadlines of critical tasks can be met [66], for a randomly generated task set, if a set of given tasks with utilization greater than the bound of Theorem 1, can meet its deadlines until the conditions of Theorem 1 is checked. The theorem 2 provides the exact schedulability criterion for independent periodic task sets under the rate monotonic algorithm.

Theorem 2. *A set of n independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if and only if*

$$\forall i, 1 \leq k \leq i, \min_{(k,l)\in R_i} \sum_{i}^{j=1} C_j \frac{1}{T_k} \left\lceil \frac{lT_k}{T_j} \right\rceil \leq 1$$

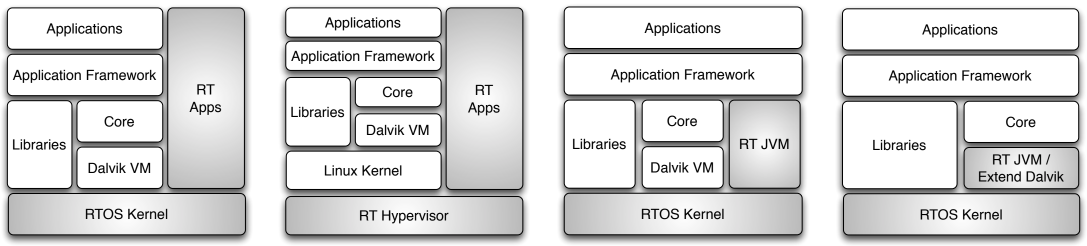
where C_i and T_i are the execution time and period of task respectively.

$$R_i = \{(k,l)|1 \leq k \leq i, l = 1, ..., \left\lfloor \frac{T_i}{T_j} \right\rfloor \}$$

For aperiodic, Strosnider [77] greatly improved the average response time for soft deadline aperiodic tasks by using a deferrable server algorithm. This is compatible with the rate monotonic scheduling algorithm, in which soft deadline aperiodic tasks is polled into the deferrable server or background service if there is free time slice while still prioritize periodic tasks with hard deadlines. For sporadic tasks, a sporadic server [73] is designed upon the deferrable server algorithm. It guarantees deadlines for hard-deadline aperiodic tasks and maintains a good responsiveness for soft-deadline aperiodic tasks. For shared resources and precedence constraints, Sha *et al.* [69] developed a priority inheritance protocol for resources sharing, and Rajkumar *et al.* [48] derived a set of sufficient conditions where allows the priority inheritance protocol using the rate monotonic algorithm, and M. Spuri [73] proposed a method that integrates both of the precedence constraint and the shared resource in the theory of the real-time scheduling.

2.3 Real-time System Architecture on Android

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 [51]. Their proposed models are depicted in Figure. 2.4. The first system model (Figure. 2.4a) 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 (Figure. 2.4b) 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 (Figure. 2.4c and Fig. 2.4d) 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



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

Figure 2.4: Proposed Real-Time Android System Models. [2]

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.

The overall performance and predictability of DVM in a real-time setting was first characterized by Oh *et al.* [59]. 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 [62]. Similarly, Mongia *et al.* have also showed that deadlines were frequently missed in Android with delays ranging from 1 *ms* to 500 *ms* [65]. 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.* [52] 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 VM, runtime libraries, and the communication mechanisms. Kalkov *et al.* [40] outline how to extend DVM to support real-time; they observed that DVM's garbage collection mechanism

[2]Shaded components represent additions or changes to the Android architecture.

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 redesign of the whole Dalvik GC, relying on developers to achieve predictability adds a layer of complexity. These solutions are in line with RTDroid's system architecture, which utilizes a real-time kernel and a real-time JVM, while also points out the problems inherent in Android's framework layer.

2.4 Real-time Extensions to Android

Moazzami *et al.* [53] built a smartphone-based platform for data-intensive, embedded sensing applications, named ORBIT. ORBIT distributes processing/control tasks, sampling tasks, and complicated algorithms over smartphones, external peripheral boards, and cloud servers, and achieves energy-efficiency and timeliness requirements for sensing apps. Similarly, Kim *et al.* [42] implemented SounDroid with low audio stream dispatching latency, which enables a high degree of QoS for sound apps by introducing extra scheduling policies for audio stream requests. However, both of them do not address the lack of real-time support problems inherent in Android software stack. RTDroid redesigns and re-implements necessary communicating components in RTDroid's framework layer. Moreover, RTDroid extends Android's programming model by extending Android's basic application constructs and declarative manifest schema with time and memory guarantees.

The memory guarantee of RTDroid's programming model is based on region-based memory management [80]. Scope memory was introduced in the RTSJ [34] to avoid GC interference. Scope memory allows the system designer to prove properties about the predictability of the overall system including static memory bounds [79]. In RTDroid's programming model, scopes are mostly *hidden* from the application developer. The developer needs to configure the system to specify necessary bounds, but does not need to worry about adhering to the scope memory rules enforced by RTSJ. Bounds are specified declaratively through RTDroid's manifest extensions, instead of programmatically, thereby abstracting out configuration from function. Since application constructs communicate through message passing, the complexity of reasoning about cross scope references and scope nesting levels (scope stacks) is handled seamlessly by our underlying system. This largely removes the cognitive burden from the developer of using scope memory in application development.

For real-time communication, Kalkov *et al.* explored how different components within a single app (or across multiple apps) interact through the `Intent` broadcasting mechanism on Android, and redesigned it for predictability [41]. This design confirm RTDroid's redesign of the real-time communication constructs [85, 86]. In this sense, Kalkov's work is an independent confirmation point that verify the necessity of redesign the framework components for real-time on RTDroid. Moreover, RTDroid's programming model provides real-time communication channels that introduces communication semantics into application development.

RTDroid utilizes ownership transfer to enable the data transfer between scoped memory areas. The ownership as a language property has been extensively studied in the context of programming language [17, 18, 29, 21, 22, 47], specifically through the development of ownership types [25, 61], which were developed to simplify the reasoning about object oriented programs by controlling aliasing and modification of objects. There also has been work on leveraging implicit ownership types to simplify the RTSJ scope memory model [87] and to prove that well typed RTSJ programs [19] are free of scope violations (*i.e.* an object lower on the scope stack will not point to an object higher on the scope stack). However, no one has yet investigated ownership transfer [56] of scopes.

2.5 Real-time Application Verification

The study of real-time scheduling started with the hard deadline case in which any task with a missed deadline was considered to be a failure [50, 35, 12]. To build a system that would guarantee temporal constrains as well as logical behavior, it was necessary to consider a worst-case formulation. Later, Gardner *et. al.* [32, 31] model real-time constrains with soft deadlines. They are defined with soft deadlines and a lateness constraint. For example, it defines $\alpha(x)$ to be the long run fraction of jobs that miss their deadlines by more than x time units, then lateness constraints are typically of the form $\alpha(x) \leq \beta$.

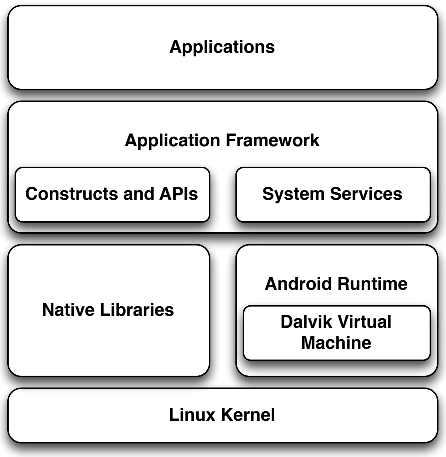
Meanwhile, the real-time task model has also has been evolved to include with more real-world aspects, such as blocking due to synchronization, precedence constraints, mode changes, operating system overhead and architectural details [76, 37, 76, 30, 9]. The task model in our work requires a task model with the support of these aspects. Unfortunately, according to our experiences, although real-time scheduling theory has been most fully developed to predict whether a set of tasks will meet their hard deadlines given a certain set of assumption about the task set and system, fewer analytic results are available for soft deadlines.

Chapter 3

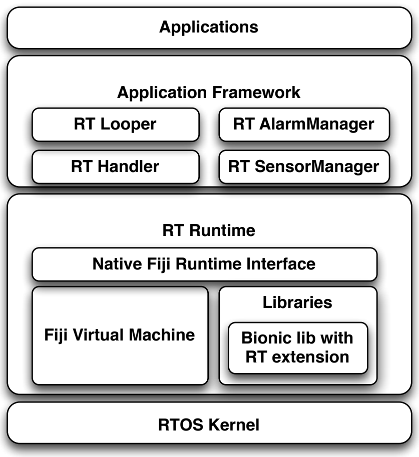
RTDroid's System Architecture

This chapter first examines the stock Android software stack and discuss how to address real-time capabilities with Android *as a whole system*. The primary goal is to provide predictability in three aspects, timing, memory, and resource predictability for *a single real-time application*. Then we present our system architecture in RTDroid which utilizes a real-time OSes, a real-time JVM and redesigned framework constructs.

The rest of this chapter is orgnized as following: Section 3.3 details three deployment profiles with various real-time guarantees, Section 3.4 reports lessons learned in enabling RTOSes and real-time JVM on targeting devices. We specify our redesign real-time constructs in framework layer, including real-time messaging passing constructs, `RTLooper` and `RTHandler` which are essential to Android's framework, and two redesigned real-time *system services*—`RTAlarmManager` and `RTSensorManager`. `RTAlarmManager` is an example that shows how to utilize real-time APIs provided in real-time JVM to implement real-time timer. `RT-SensorManager` exemplifies how to leverage the hardware abstraction layer on Android's customized kernel, and enables hardware access with real-time predictability. The evaluation results are available in RTDroid's published work [86, 85, 84]. Note that the redesign framework constructs is mainly used for proof of concept. In the next Chapter 4, we will illustrate RTDroid's event-driven programming model inherits from Android's model with real-time expressiveness, real-time communication semantics and static memory guarantees. The real-time constructs in programming model will replace the redesigned framework



(a) Simplified Android Architecture



(b) RTDroid Architecture

Figure 3.1: Comparison of Simplified Android and RTDroid Architectures

constructs discussed in this chapter.

3.1 Android Background

Figure 3.1a 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 Figure 3.1a 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 [40, 33] 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 well-understood that the bottom layers need real-time support in order to provide a predictable platform. However, we also found 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 Figure 3.1a. The first problem lies in the category shown on the left, *constructs and APIs*, which provides programming constructs and APIs that application developers can use such as `Looper`, `Handler`, and `AsyncTask`. This category poses a problem for real-time applications since the constructs do not provide any time or memory predictability as well as priority awareness. The main issue is that the latency of message delivery in these mechanisms is unpredictable; lower priority threads can unnecessarily prevent higher priority threads from making progress. In Section 3.5, we discuss this problem in more detail and present our real alternatives.

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 3.6 and 3.7, 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 System Architecture

In order to provide real-time support in all three layers depicted in Figure 3.1a, we advocate a clean-slate redesign of Android in Figure 3.1b. 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 redesign RT-Droid's real-time constructs in framework layer with time, memory and resource predictability, while preserve *Android compatibility* with best effort. *Android compatibility* means providing the same set of Android APIs as well as preserving their semantics for real-time applications.

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 [63], 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 [64], 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 currently not 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.

3.3 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 RT Linux kernel and closed source drivers as we explain below.

- ◇ **Soft Real-time Smartphone:** This profile provides the loosest guarantees due to its reliance on unverified closed source drivers and a partially preemptible RTLinux kernel as opposed to a *fully* preemtible RTLinux kernel. [1] As the mobile-oriented kernel modification in Android's kernel, the customized Android Linux kernel 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, RTDroid's published results [86, 85, 84] show 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, at least from the kernel or the drivers.

3.4 Real-time Building Blocks

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 An-

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

droid has introduced extensive changes in the kernel that are not compatible with RTLinux patches. In the following section, we first briefly describe our x86 and LEON3 environments. We then report our experience with the smartphone environment in detail.

3.4.1 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 FijiVM [63] as the real-time VM. FijiVm 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.

3.4.2 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.
Bionic: Android does not utilize glibc as the core C library, instead it uses its own library called Bionic [28]. 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.

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.

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) [4] and *CPUFreq governors* [27]. 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 [64]. 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 [23]. 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 [54] to validate the hardware and leverage this timing analysis to modify the kernel and VM schedulers appropriately.

3.5 RT Looper and RT Handler

As discussed in Section 3.1, the first issue that the application framework poses lies in its message-passing constructs. These constructs do not provide any predictability or priority-awareness. We detail this issue in this section and discuss how we address it in RTDroid.

3.5.1 Android's Looper and Handler

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 con-

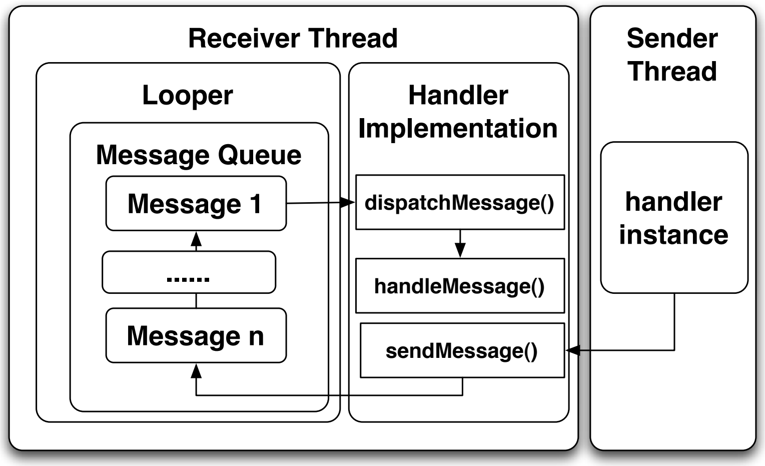


Figure 3.2: The Use of `Looper` and `Handler`

structs, 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.

Figure 3.2 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 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 multi-

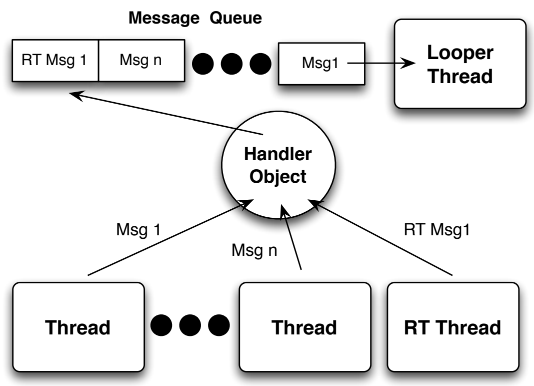


Figure 3.3: Android's Looper and Handler: The thread in which the looper executes processes the messages sent through the associated handler object in the order in which they are received.

ple 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 Figure 3.3. 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.

3.5.2 Real-Time 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

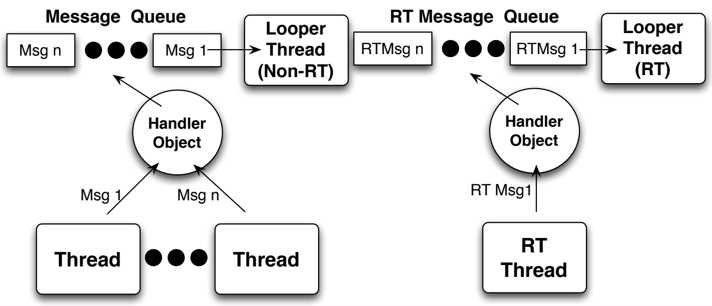


Figure 3.4: An Example of `Looper` and `Handler` in RTDroid. Each message has a priority and is stored in a priority queue. Processing of messages is also done by priority. The example shows one high-priority thread and multiple non-real-time threads.

queue to process the messages according to its priority. Figure 3.4 shows our new implementation for `Looper` and `Handler`. Since we now process each message according to its sender's priority, messages sent by lower priority threads do not delay the messages sent by higher priority threads. For memory predictability, queues can be statically configured in size.

3.5.3 Worst-Case Execution Time Analysis

To understand the worst-case execution characteristics of the Real-time `Looper` and `Handler` we must reason about how the constructs process a series of messages and execute each message's callback function. We define T_i^j to be the i^{th} message issued by the application from a thread with priority j. The messages are passed into a real-time `Looper` that has the same priority as the messages and then they are enqueued in a `MessageQueue`. The time cost for handling the i^{th} message in priority level `j` is shown as S_i^j in Equation (3.1):

$$S_i^j = \sum_{l=0}^{i} (h_l^j + deq(T_l^j)), \tag{3.1}$$

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 (3.2):

$$phase_0(T_i^j) = \sum_{p>j} S_{last}^p + S_i^j. \tag{3.2}$$

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 [2] 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 3.2. 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 (3.3):

$$phase_n(T_i^j) = \sum_{p\geq j} \sum_{i=0}^{phase_{n-1}(T_i^j)*R_p} \left(h_i^p + deq(i) + enq(i)\right). \tag{3.3}$$

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

[2]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.

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 (3.4):

$$\begin{aligned} WCET(T_i^j) &= phase_0(T_i^j) + phase_1(T_i^j) \\ &\quad + ... + phase_{n-1}(T_i^j) + phase_n(T_i^j). \end{aligned} \tag{3.4}$$

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

3.6 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—

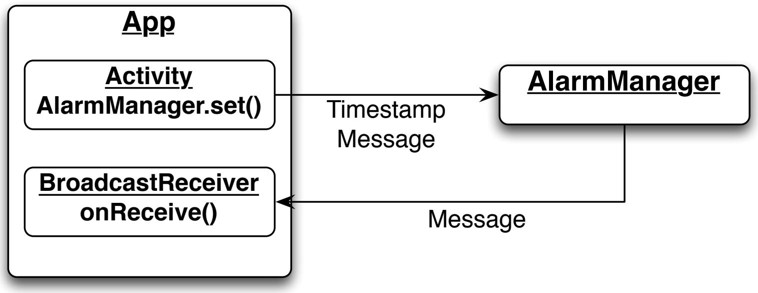


Figure 3.5: An Example Flow of `AlarmManager`. An application uses `AlarmManager.set()` to register an alarm. When the alarm triggers, the `AlarmManager` sends a message back to the application, and the application's callback (`BroadcastReceiver.onReceive()` in the example) gets executed.

`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.

3.6.1 Android's Alarm Manager

`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`.

Figure 3.5 shows how `AlarmManager` works, including alarm registration and alarm delivery. An IPC call, with a message [3] and execution time, is made to the `AlarmManager` every time an application registers an alarm. When the the alarm triggers at the specified time, the `AlarmManager` sends a message back to the application, and the associated callback is executed. The issue with `AlarmManager` is that it provides no guarantee on when or in what order alarm messages are delivered, hence does not provide any timing guarantee or priority-awareness.

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

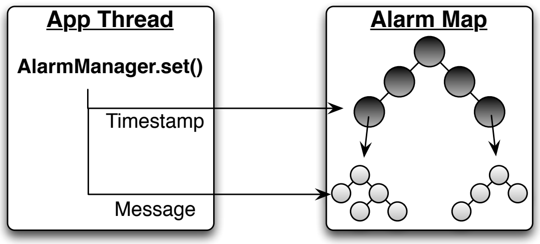


Figure 3.6: The Implementation of Alarm Execution on RTDroid. The tree colored black at the top maintains timestamps. The trees colored gray are per-timestamp trees maintaining actual alarm messages to be delivered.

3.6.2 Real-Time 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 Figure 3.6. This means that we can make the registration process predictable based on the complexity of red-black tree operations, *i.e.*, the longest path of a tree is no longer than twice the shortest path of the tree. We use one red-black tree for storing timestamps and pointers to per-timestamp red-black trees. Per-timestamp trees are leveraged to order alarms with the same timestamp by their sender's priority. Thus, our alarm registration process is essentially one insert operation to the timestamp tree and another insert operation to a per-timestamp tree. By organizing the alarms based on senders' priorities, we guarantee that an alarm message for a low priority thread does not delay an alarm message for a high priority thread. Expired alarms are discarded. Note that this ensures that low priority threads whose alarm registration rate exceeds the alarm delivery capacity of the system cannot prevent a high priority alarm from being triggered.

For alarm delivery, we create an `AlarmManager` thread and assign the highest priority for timely delivery of alarm messages. This thread replaces the original multi-process message passing architecture of Android. It wakes up whenever an application inserts a new alarm into our red-black trees, then it schedules a new thread at the specified time for the alarm. We associate the application's callback for the alarm message with this new thread. For precise execution timing of this callback thread, we implement Asynchronous Event Handlers (AEH) that Real-Time Specification for Java (RTSJ) [34] specifies the interface for.

We have implemented two versions for AEH. The first is a per-thread AEH implementation used in our workshop paper [85], 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 [43].

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 [26].

3.6.3 Worst-Case Execution Time Analysis

The worst-case execution scenario for `AlarmManager` is similar to that discussed for the `Looper` and `Handler` in Section 3.5.3. 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 (3.1), (3.2), (3.3), (3.4), but couched in terms of alarm processing instead of message delivery.

1. T_i^j represents the i^{th} alarm registered by application with priority j.
2. S_i^j represents the time cost for handling the i^{th} alarm in priority level `j`.
3. h_i^j is the cost of time to execute the alarm T_i^j.
4. *last* is the last alarm in priority level `j`.
5. enq(T_i^j) is the cost of alarm registration.

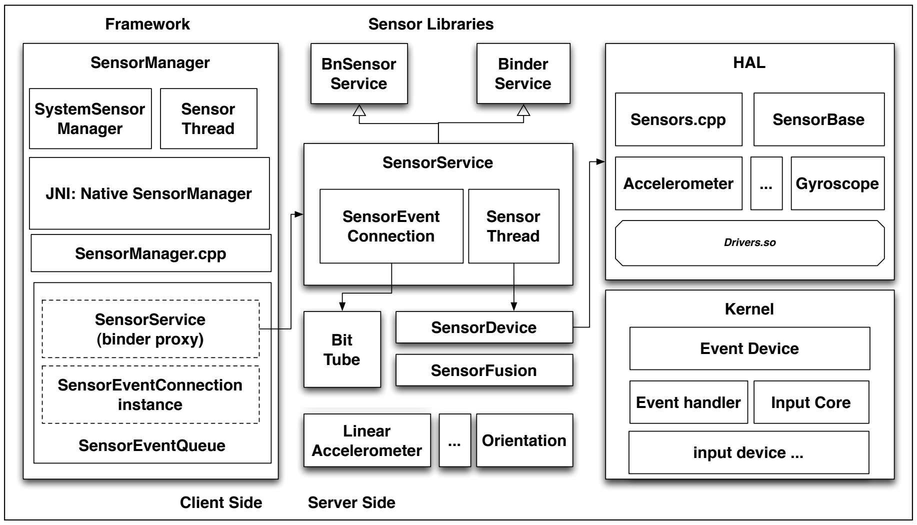


Figure 3.7: Android Sensor Architecture

6. deq(T_i^j) is the cost of alarm delivery.

3.7 RT Sensor Manager

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.

3.7.1 Android's Sensor Manager

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`. Figure 3.7 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

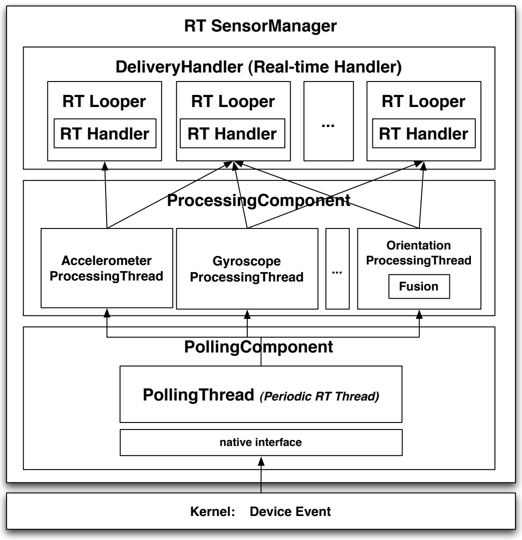


Figure 3.8: RTDroid Sensor Architecture

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.

3.7.2 Real-Time 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 [60, 82]. We first describe the architecture and discuss how we address the two problems with our new architecture.

As shown in Figure 3.8, 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 Figure 3.8, *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*.

3.7.3 Worst-Case Execution Time Analysis

The worst-case execution scenario for `SensorManager` is slightly different than what we have discussed in Section 3.5.3 and 3.6.3. The upper bound for delivery of the sensor event to a sensor listener, l, consists of three parts: (1) the time cost of the system delivery the sensor event to all sensor listeners that registered a listener that are greater or equal to the priority of l, (2) recursively integrate the time cost for register and deliver of the sensor data for the new higher-priority listener arriving at a per priority rate, and (3) the time cost for polling the data from each sensor kernel module. The WCET equation for `SensorManager` is in the same fashion as previously defined in Equation (3.1), (3.2), (3.3), (3.4), and includes the sensor data polling cost as shown in in Equation (3.5), (3.6) :

$$phase_0(T_i^j) = \sum_{p \geq j} P_j(sensor_e) + \sum_{p > j} S_{last}^p + S_i^j \tag{3.5}$$

$$phase_n(T_i^j) = \sum_{p \geq j} \sum_{i=0}^{phase_{n-1}(T_i^j) * R_p} \left(h_i^p + deq(T_i^j) + enq(T_i^j)\right). \tag{3.6}$$

1. T_i^j represents the i^{th} sensor listener in application with priority j.
2. S_i^j represents the time cost to execute the i^{th} callback of sensor listener in priority level `j` .
3. h_i^j is the amount of time to execute the callback of sensor listener of T_i^j.
4. deq(T_i^j) is the cost of listener registration.
5. $P_j(sensor_e)$ is the cost of sensor data polling.

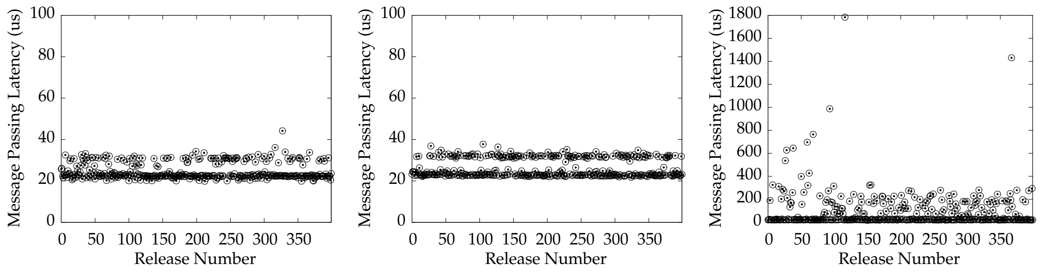
3.8 Evaluation

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 3.3. 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. 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 3.5.3, 3.6.3, and 3.7.3.

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.

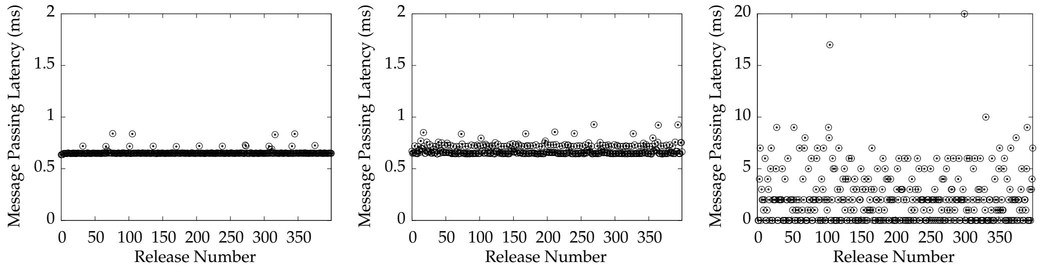
3.8.1 RT Looper and RT Handler Microbenchmarks

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 applica-



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

Figure 3.9: The observed raw latency of `Looper` and `Handler` on x86.



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

Figure 3.10: The observed raw latency of `Looper` and `Handler` on LEON3.

tion 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. 3.9 and Fig. 3.10 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.

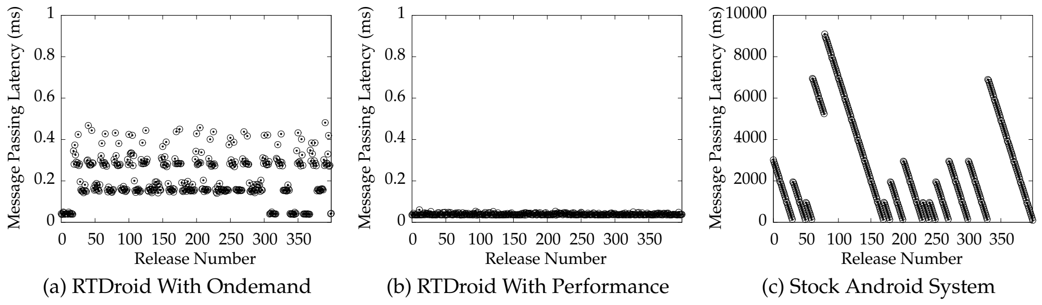


Figure 3.11: The observed raw latency of `Looper` and `Handler` on Nexus S.

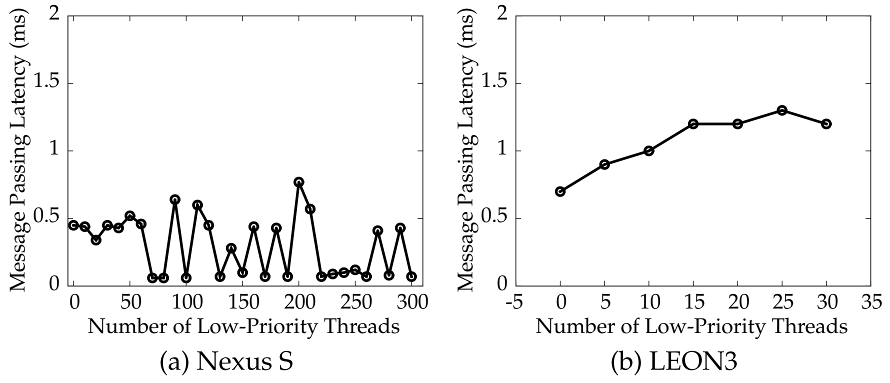
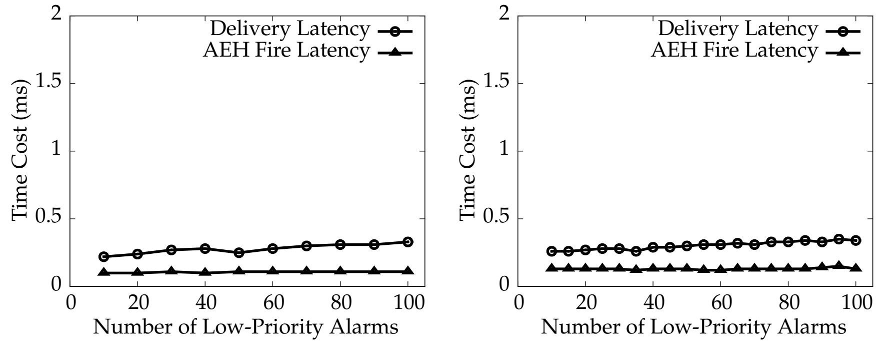


Figure 3.12: Observed WCETs of Message Passing Latency

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. 3.11 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. 3.11a 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. 3.11a, 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. 3.11c.



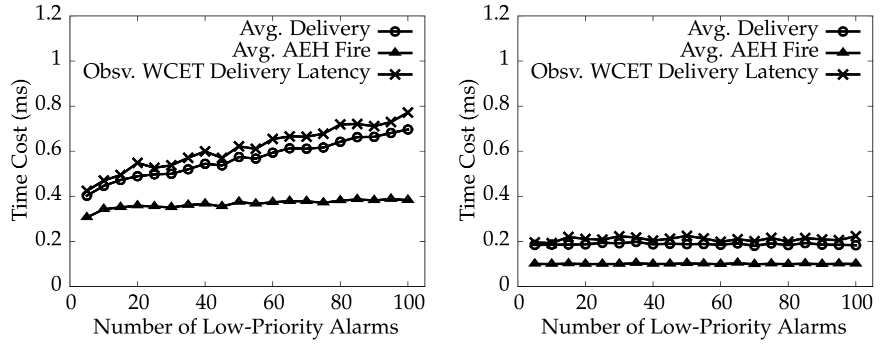
(a) RTDroid: Per Thread (b) RTDroid: Thread Pool

Figure 3.13: RT AlarmManager Per Thread vs Thread Pool on x86.

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. 3.12 shows observed WCETs of message passing latency over 10 executions with increasing number of low-priority threads. Fig. 3.12a 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-preemtible Linux RT kernel on Nexus S. We do not observe such variance on LEON3, show in Fig. 3.12b, 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.

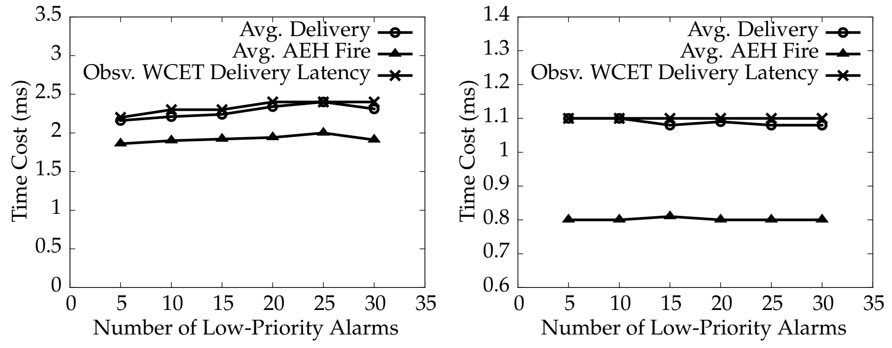
3.8.2 RT AlarmManager Microbenchmarks

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



(a) RTDroid: Per Thread (b) RTDroid: Thread Pool

Figure 3.14: RT AlarmManager - Per Thread vs Thread Pool on Nexus S.



(a) RTDroid: Per Thread (b) RTDroid: Thread Pool

Figure 3.15: RT AlarmManager - Per Thread vs Thread Pool on LEON3.

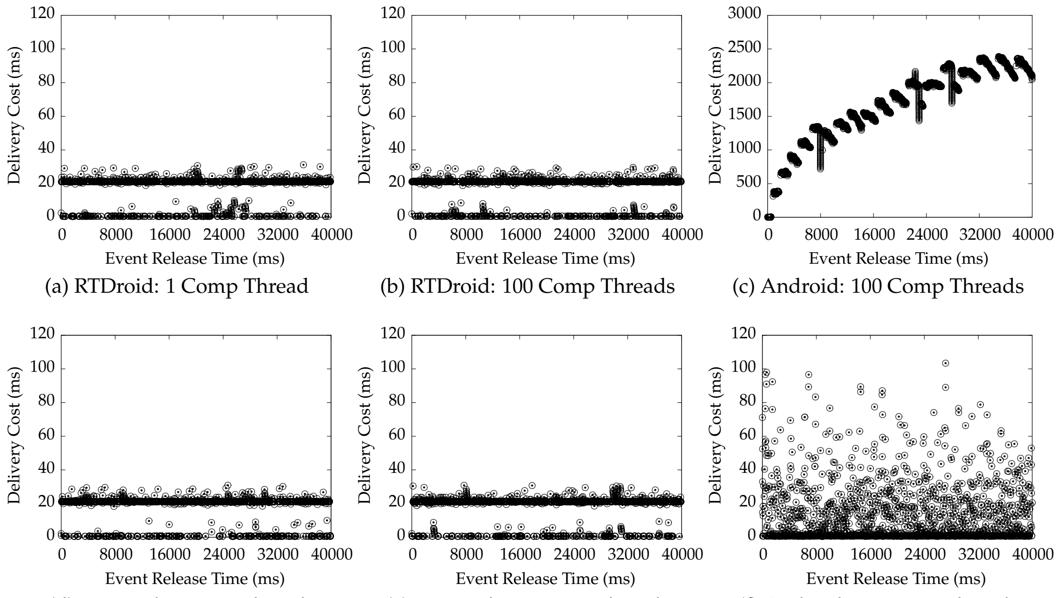
how long it takes for the system to deliver an alarm from the `AlarmManager` to the app. We run the experiment on all three platforms. These results show the timing and latencies of the alarm execution process and indicate that the `RT AlarmManager` is efficient at prioritizing high-priority alarms and scheduling them at their specified time.

As mentioned in Section 3.6, we have implemented two techniques for alarm management in `RT AlarmManager`—one with a per-thread AEH implementation used in our previous workshop paper [85] 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. 3.13 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. 3.14 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. 3.14a 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. 3.14b 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. 3.15 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 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



(a) RTDroid: 1 Comp Thread (b) RTDroid: 100 Comp Threads (c) Android: 100 Comp Threads

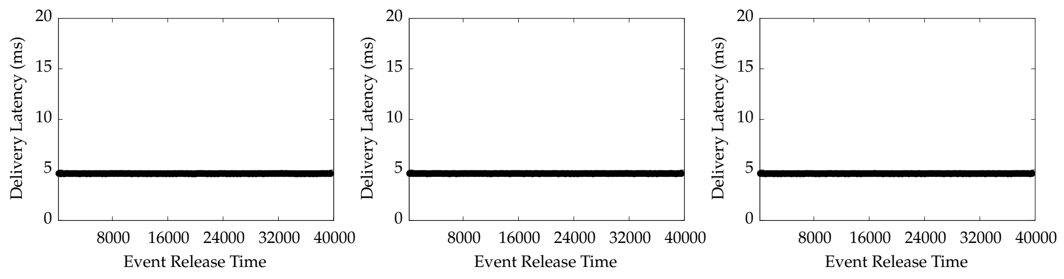
(d) RTDroid: 1 Mem Thread (e) RTDroid: 100 Mem Threads (f) Android: 100 Mem Threads

Figure 3.16: Memory and Computation stress test for the Fall Detection Application on Nexus S.

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. 3.15a and Fig. 3.15b.

3.8.3 Applications on Real-Time SensorManager

To validate the predictability of our sensor architectures in data delivery, we have developed two applications. The first application is a soft real-time fall detection application that leverages our `SensorManager` outlined in Section 3.7. We designed two experiments with two different types of workloads: (1) a memory



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

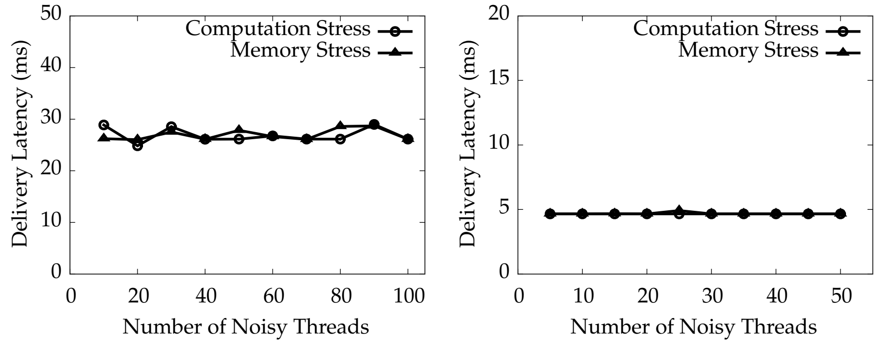
Figure 3.17: Memory and Computation Stress Test for Fall Detection Application on LEON3.

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 [64]. 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.

3.8.3.1 Fall Detection Application

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 3.7, 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. 3.16 illustrates the observed latency of the sensor event delivery for the fall detection application. To stress the predictability of our `SensorManager` im-



(a) Nexus S (b) LEON3

Figure 3.18: `RT SensorManager` Observed WCET of Sensor Data Delivery in Fall Detection.

plementation, 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. 3.16a, Fig. 3.16b, Fig. 3.16d and Fig. 3.16e 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. 3.16c and Fig. 3.16f to show the effect of low-priority threads on sensor event delivery in stock Android.

Fig. 3.17 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. 3.16.

Fig. 3.18 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

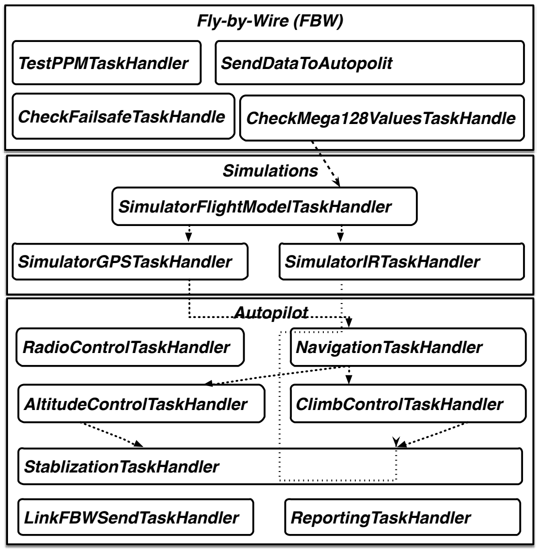
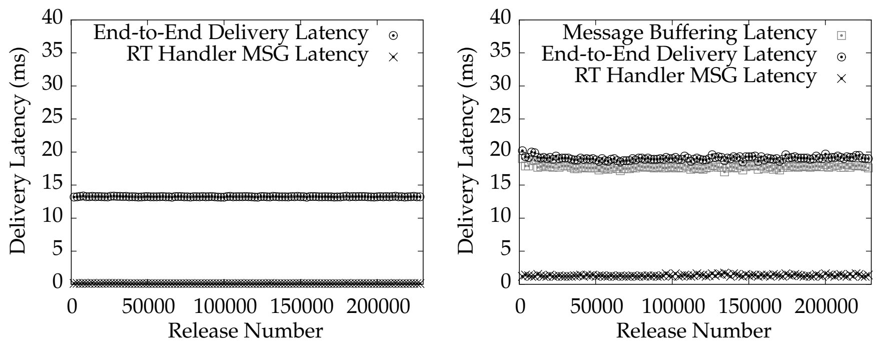


Figure 3.19: jPapaBench Task Dependency

`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 i^{2}c bus on RTEMS, instead of buffering and polling from kernel buffer.

3.8.3.2 jPapaBench

The second application is a port of a traditional benchmark application, named jPapaBench [38]. It is designed as a Java real-time benchmark to evaluate Java real-time virtual machines. It mirrors the function of paparazzi [20], 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



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

Figure 3.20: `RT SensorManager` performance base line

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. 3.19.

As noted by Blanton *et. al.* [14], 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.

In all our results, we show end-to-end latency as well as the breakdown of latency. In Fig. 3.20 through Fig. 3.22, 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

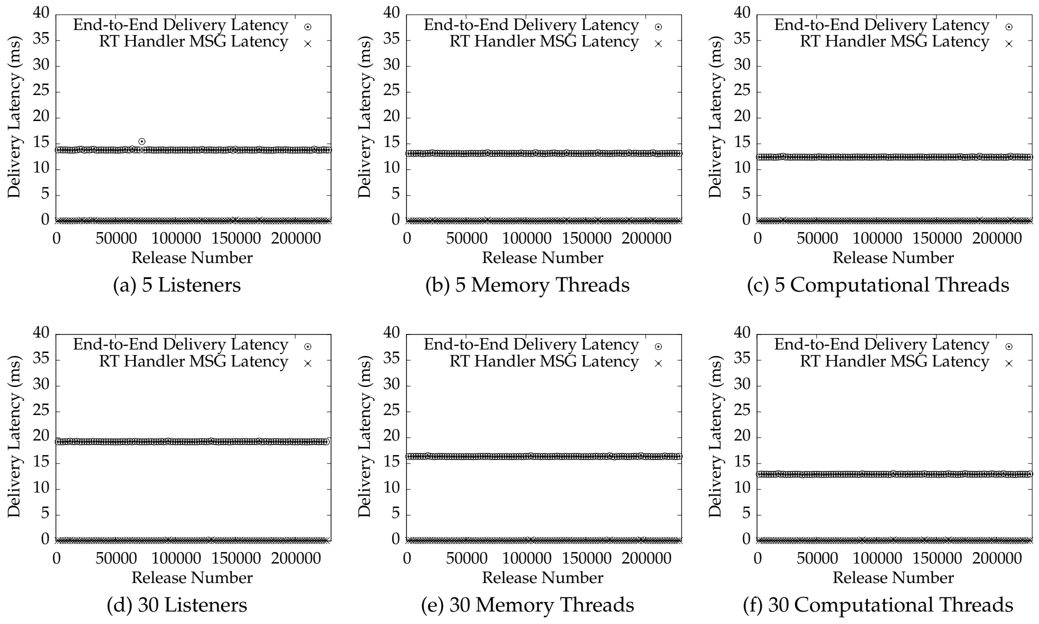


Figure 3.21: `RT SensorManager` stress tests on Nexus S

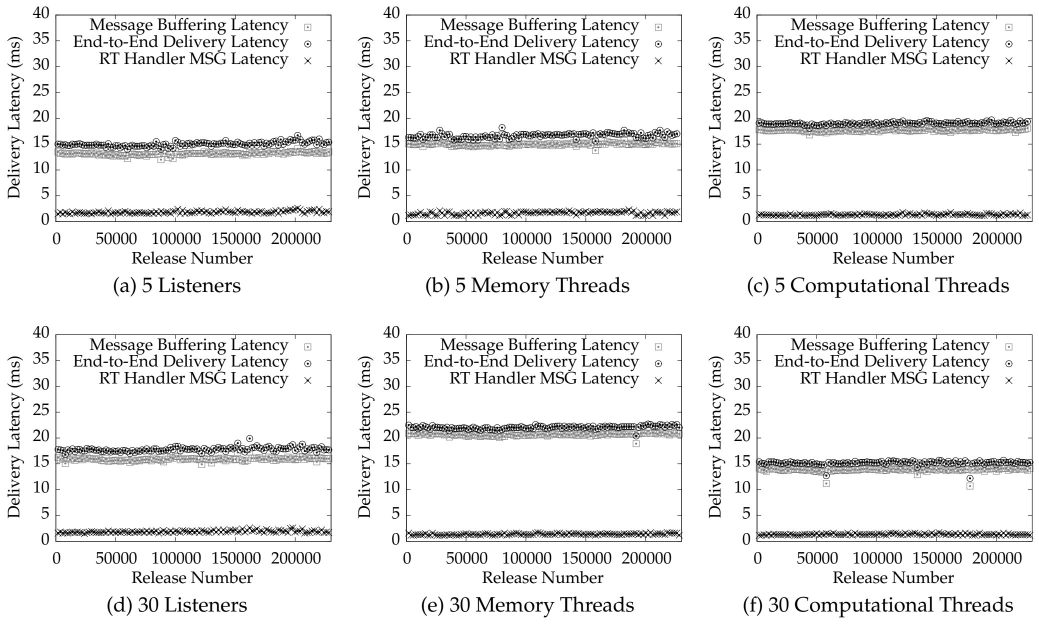
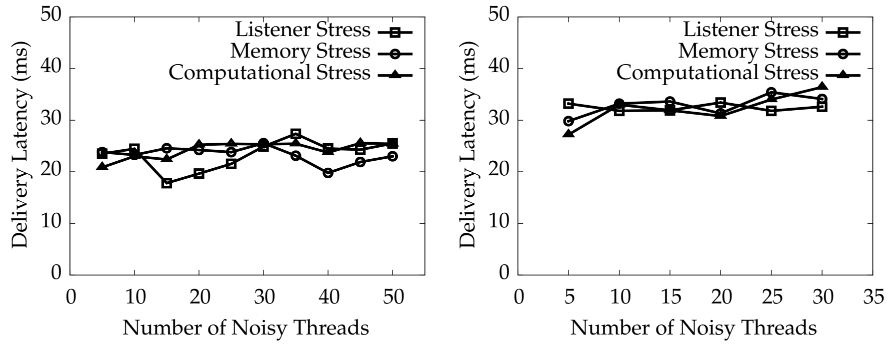


Figure 3.22: `RT SensorManager` stress tests on LEON3



(a) Nexus S (b) LEON3

Figure 3.23: `RT SensorManager` Observed WCET of Sensor Data Delivery in jPapaBench Application

to the latency incurred purely in our architecture. The square points 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. 3.20 through Fig. 3.22, 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. 3.20 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. 3.21 and Fig. 3.22 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, respectively. Interested readers can view additional experimental results and raw data on our website [4]. The baselines for both hardware platforms are provided in Fig. 3.20a and Fig. 3.20b. Fig. 3.23 presents the observed WCETs of the end-to-end delivery latency over 10 executions with an increasing number of noisy threads.

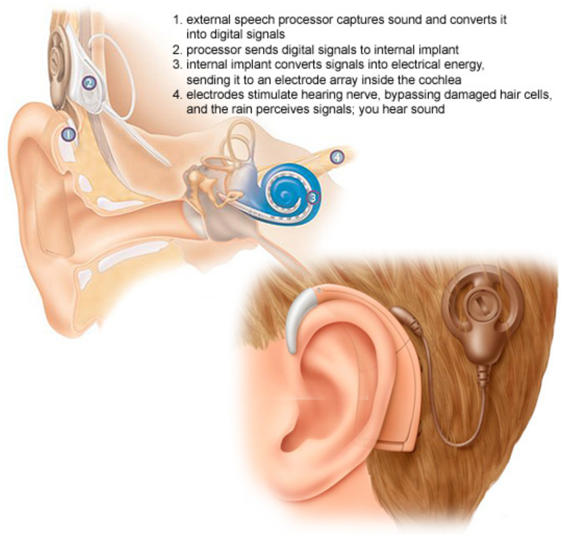
The observed WCETs of sensor delivery range from 19.65 *ms* to 25.55 *ms* with 0.19 *ms* to 5.31 *ms* standard deviations on Nexus S. On LEON3, the results of LEON3 reflect the same trend with larger values, WCETs are from 27.20 *ms* to 35.40 *ms* with and 3.7 *ms* to 7.47 *ms* standard deviations. Such 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.

[4]Full results available: `http://rtdroid.cse.buffalo.edu`

Chapter 4

RTDroid's Real-Time Progamming Model

This chapter explains an event-drive programming model that built upon RTDroid. This programming model consists four parts for the application development: (1) a set of real-time components for the real-time application development, (2) fourreal-time communication channels which defines real-time semantics for components interactions, (3) unpauseless region-based memory management for memory guarantee at runtime, and (4) an extension to Android's manifest to express the real-time requirements of an application.



1. external speech processor captures sound and converts it into digital signals
2. processor sends digital signals to internal implant
3. internal implant converts signals into electrical energy, sending it to an electrode array inside the cochlea
4. electrodes stimulate hearing nerve, bypassing damaged hair cells, and the rain perceives signals; you hear sound
1
2
3
4

Figure 4.1: Cochlear Implant[1]

```
class ConfigurationUI extends
    Activity {
  ClickListener l = new
      ClickListener() {
    public void onClick(View
        v) {
      //change processing config
  } };
  public void onStart(){
    button.setOnClickListener(l);
  }
}
```

Figure 4.2: Audio Configuration UI written in Android.

```
class ProcessingService
    extends Service {
  public void onStartCommand()
      {
    /* periodic audio processing
        */
    while (true) {
      //process every 8 ms
    }
  }
  ...
}
```

Figure 4.3: Audio Processing Service written in Android.

To illustrate the programming model, we use a cochlear implant application as an example. The cochlear implant is a device that restores hearing abilities through a surgically inserted electronic mechanism in the patient's inner ear. As Figure 4.1 shows, an external component for capturing ambient audio and converting that audio into digital signals is also required. Lastly, a processing device, which translates signals into electrical energy, triggers implanted electrodes to simulate hearing nerves. Recently, there has been interest in leveraging smartphones [8] to reduce the size of the components external to the body and to reduce the number of devices the patient must carry. In such a scenario, the smartphone is used to record audio streams, perform audio processing, and send the converted signals to the implant. For this hybrid system to be useful, the audio processing must process sound samples every 8 *ms*.

The rest of this chapter is organized as following: Section 4.1 presents the software architecture of the cochlear implant and discusses limitations and challenges of implementing such an application in Android. Section 4.2, Section 4.3 and Section 4.4 detail how these challenges are addressed in RTDroid's programming model. Lately, Section 4.5 demonstrates evaluation results in microbenchmarks and applications.

[1]Source www.bcfamilyhearing.com/my-child-has-a-hearing-loss/hearing/cochlear-implants/

4.1 Android-enabled Real-time Applications

Using the stock Android platform for real-time computing is challenging for a number of reasons which we summarize here. Android provides three software architectural elements, *services*, *activities*, and *broadcast receivers*, for, respectively, background computation, foreground computation with user input, and handling system-wide events. The Android scheduler is not priority aware and there is no mechanism to assign priorities to threads. Android offers two communication mechanisms: messages and intents. Messages are received by a `Handler` which is a unique mailbox for all messages directed at a component. As there is no notion of priority for messages, the first-in first-out queue associated with a handler can lead to priority inversion. An `Intent` is an event that triggers execution of callbacks in components that have registered for it. Intents can lead priority inversion as callbacks are executed by the receiver which may have different timeliness requirements than the component that raised the intent. Memory pressure is also a concern. Android provides no mechanism other than garbage collection to manage memory, and its garbage collector does not have real-time guarantees. To makes matters worse, there is no way to bound memory consumed by different components. Thus a stray non-critical component can affect the whole system.

Even with theses limitations, the health care industry has been studying how to adapt Android for wearable and implantable health devices, like *cochlear implants*. A cochlear implant restores hearing abilities through an electronic device surgically inserted in the inner ear. It relies on external components to capture ambient audio, convert it into digital signals, and translate the signals into electrical energy. There is interest in leveraging smartphones [8] to provide additional services such as on-the-fly translation or noise cancellation. In such a scenario, a smartphone records audio streams and processes them. To provide acceptable performance sound samples must be handled at rate of one every 8 *ms*.

A plausible design for such an application would be to split the user interface that controls volume and noise reduction from sound processing. The UI can be implemented as an activity as shown in Figure 4.2. It deals with configuration parameters set by the user. On the other hand, sound processing is best modeled as a service (Figure 4.3), which repeatedly processes sound samples. Even in such a simple use-case, it is important to ensure that sound processing will not be delayed by UI processing. When components have to interact through Android-based communication mechanisms, ensuring non-interference becomes even more tricky.

Figure 4.4 shows the architecture of our solution in RTDroid. It separates real-time (`RecordingService`, `ProcessingService`, and `OutputReceiver`) and non-real-time components (`VolumeReceiver` and `ConfigurationUI`). The former have priorities attached and use communication services that prioritize messages. `ConfigurationUI` has a `Handler` for other components to update the UI, and a non-real-time receiver listens on volume key events. It also receives messages from real-time components. Similarly, the `ProcessingService` receives messages from non real-time components (`VolumeReceiver` and `ButtonListener`) and a real-tine component (`RecordingService`). RTDroid allows these components to communicate while enforcing memory bounds. Each real-time component is provided a fixed amount of memory for its exclusive use. That memory is divided into two sections, one persists for the lifetime of the component, the other is cleared each time the component yields control. Messages are pre-allocated. Non real-time components allocate messages in heap memory. RTDroid extends the Android manifest to enable developers to declare properties of components that include priority, periodicity and memory bounds.

4.2 RTDroid's Application Components

RTDroid supports three different real-time components: services, tasks, and receivers. A `RealtimeService` is a counterpart to Android's service used for one-shot aperiodic or sporadic computation. As the notion of periodic computation is foreign to Android, we introduce the `PeriodicTask` class to model such behavior. Tasks are used internally within a real-time service. A `RealtimeReceiver` is used to react to system-wide events delivered via intents. We do not provide a counterpart to Android's activities; they are used for UI programming and we have not yet observed real-time requirements for these. We do allow for in-

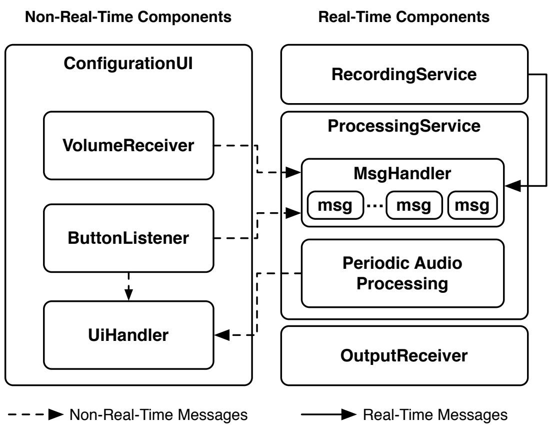


Figure 4.4: Architecture of Cochlear Implant Application.

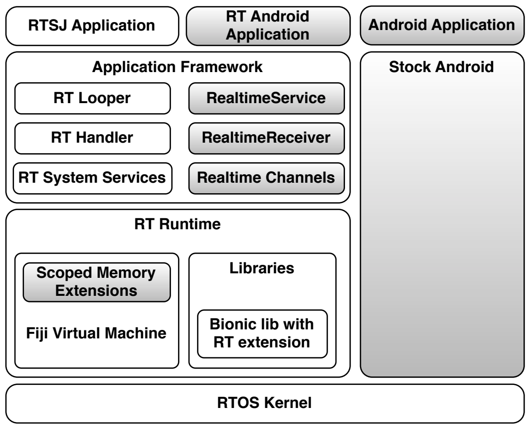


Figure 4.5: RTDroid runtime architecture.

Gray elements are the extensions introduced in this paper. Notably, we now support multiple, interacting real-time applications, real-time applications written using an Android like programming model, as well as legacy code and stock Android applications.

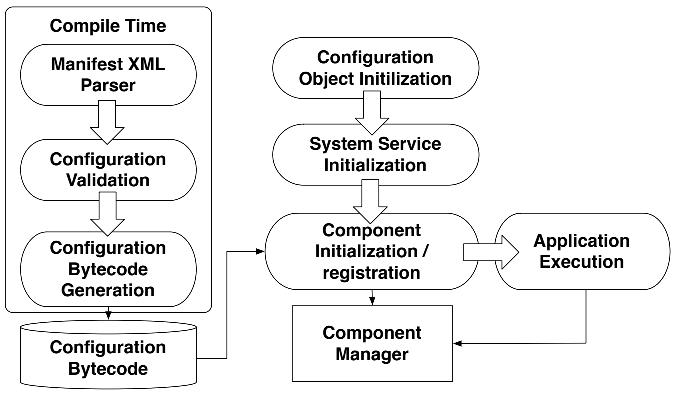

Compile Time
Manifest XML Parser
Configuration Validation
Configuration Bytecode Generation
Configuration Bytecode
Configuration Object Initilization
System Service Initialization
Component Initialization / registration
Application Execution
Component Manager

Figure 4.6: Bootstrap sequence.

teraction between UI components and real-time components through message channels.

Real-time components are statically assigned the following: a priority, a starting time, a deadline, and a memory limit. This is done declaratively by extending Android's manifest with properties (`priority`, `memSizes`, `release`, etc.). The association between a periodic task and its parent is also specified in the manifest by a `periodic-task` tag. The manifest provides information for boot-time verification and pre-allocation of components. RTDroid ensures that the total memory requested specified for a component equals the objects in its persistent memory, its per-release memory, and that of its sub-components. Figure 4.7 shows a manifest for the processing service of our running example.

Managing the lifetime of components requires: (1) ensuring priorities, deadlines, and periodicity of components, (2) automatically managing memory allocated by components, and (3) guaranteeing per-component memory bounds. We extend RTDroid's priority based scheduler and introduce a declarative specification for configuration of component requirements to ensure point (1). We introduce memory regions and specialized channels for ensuring points (2) and (3). Our VM parses this declarative specification and pre-allocates all necessary constructs, memory regions, and channels.

The concept of region-based memory allocation is an old one. The idea is to avoid having to manage individual objects, instead objects are allocated in regions, which can be deallocated in one fell swoop. The RTSJ introduced this idea to Java to provide an alternative to garbage collection. In the RTSJ, each thread may be associated to a particular scope, and scopes can be nested to make a cactus stack. For RTDroid, region-based allocations has two important benefits, threads that are using it need not be paused during garbage collection and they make it possible to bound the amount of memory allocated by any thread.

The RTDroid system supports a much simpler form of scoped memory than the RTSJ. Each component has access to two scopes, one is *Persistent Memory* for data that lives as long as the component and the other is *Release Memory* which is cleared before each release of a periodic task. The size of these scopes is given in the manifest. The total memory of a component is the sum of its persistent memory, release memory, and the memory of internal components.

4.2.1 Real-Time Service

A real-time service is an abstract class; a programmer needs to implement its callbacks. These callbacks are directly inherited from Android's service and they are invoked at different points in the lifetime of a service. The `onCreate()` callback is invoked at service creation. The `onStartCommand()` method is called at startup and usually implements application logic. Figure 4.9 shows a ser-

```
<service name="pkg.ProcessingService"
    priority="79">
  <memSizes total="3M"
      persistent="1M" release="1M" />
  <release start="0ms">
  <periodic-task
      name="processingTask">
    <priority priority="79"/>
    <memSizes release="1M"/>
    <release start="0ms"
        periodic="8ms" />
  </periodic-task>
  <!-- subscribes to the msgHandler
      channel -->
  <intent-filter count="2"
      role="subscriber">
    <action name="msgHandler"/>
  </intent-filter>
```

Figure 4.7: An extended Android manifest.

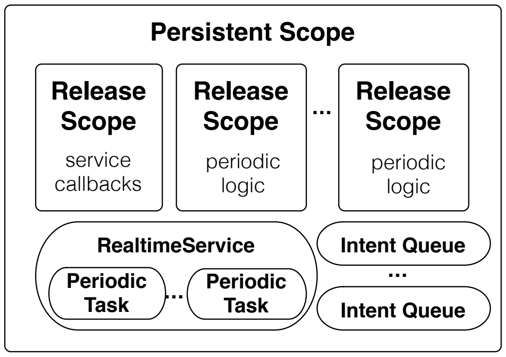


Figure 4.8: Scope Structure for a Service.

vice that starts a periodic task. Unlike Android services which run in the main thread, RTDroid services execute in dedicated threads. This change is necessary in order to allow services to run with different priorities.

Services are bound to real-time threads from the underlying real-time JVM. By default, when a service is initialized, it is assigned a persistent memory scope that has the same lifetime as the service. The scope is allocated when the service starts and deallocated when the service terminates. Static initializers for the service are run in this scope. In addition, if the service uses communication channels, intent queues are allocated in persistent memory. Callbacks execute within the scope of release memory for the associated service. The release memory is cleared when the callback returns. Similarly, when a periodic task is started in a service, it is also assigned a release scope. Note that our manifest requires specification of memory bounds for callbacks and periodic tasks, and this information is used to size release scopes appropriately. Figure 4.8 depicts the scope structure for a service as well as pre-allocated objects during boot.

```
class ProcessingService extends RealtimeService{
  PeriodicTask task = new PeriodicTask(){
    public void onRelease(){
      /* periodic audio processing logic */
  } }; ...
  public int onStartCommand(...){
    /* Each registered task starts after the
      onStartCommand() callback. */
    registerTask("processingTask", task);
  } }
```

Figure 4.9: Real-Time Service and Periodic Task.

```
<channel name="msgHandler"
         type="rt-msg" >
  <order> priority-inheritance </order>
  <execution>
    component-priority
  </execution>
  <drop>priority&oldest</drop>
  <data size="256B" \
        type="app/octet-stream"/>
</channel>
```

Figure 4.10: Real-time Channel Declaration.

4.2.2 Periodic Task

A periodic task is a sub-component of a service. In addition to the characteristics of its parent service, a task needs a period. Figure 4.9 shows an example which processes audio input periodically. A programmer needs only to implement `onRelease()` to specify a periodic computation.

4.2.3 Real-Time Receiver

In Android a new broadcast receiver is allocated whenever an intent is received, which results in frequent object allocation and deallocation if many intents are sent from a component. In RTDroid, a real-time receiver is a persistent construct, we reuse the same receiver to reduce memory pressure. As a direct consequence, a receiver can only process one intent at a time. Application logic is expressed in callbacks. The `onReceive()` defines logic to react to events, it is invoked when an intent is received. A new callback, `onClean()`, resets class variables in a receiver. This callback is used to cleanup any state between intents and is necessary if the programmer wishes to have stateless processing. This callback is not needed if the receiver only modifies local variables as they live in release memory and will be cleared automatically. In our running example we implement `OutputReceiver` as a receiver to react to the processed audio output sent by the `ProcessingService`.

One important design choice is the priority of a callback, RTDroid decouples intent delivery from the callback execution. Intents are delivered according to policy enforced by real-time channels (described later). Callbacks are executed at the priority of their component. Multiple callbacks triggered by a series of intents are serialized and will be executed in-order. In the cochlear implant, `ProcessingService` sends audio to `OutputReceiver` through a real-time channel. The channel guarantees that intents are delivered to the receiver with the priority of the `ProcessingService`, and the callback is invoked asynchronously with the priority of the receiver.

In implementation terms, a receiver is bound to an asynchronous event handler in the underlying JVM and backed by a priority message queue. An asynchronous event handler can serialize multiple releases from different senders, and the priority queue ensures the intent delivery order is based on the sender's priority. The callback is executed by the asynchronous event handler, which is assigned the priority of callback method's owner.

4.3 Real-Time Communication Channels

RTDroid provides four types of real-time channels for communication: (1) message passing channels, (2) broadcast channels, (3) bulk data transfer channels, and (4) cross-context channels to communicate with non real-time components. Following Android conventions, programmers declaratively specify channel name, events, data type, and size. Real-time components must specify the number of messages that they send or receive per release. This ensures that we can preallocate the messaging objects and enforce memory bounds for all channels. There is one primordial cross-context channel to facilitate interaction with other Android applications and services. All other channels are explicitly created by programmers.

Figure 4.10 shows a real-time message passing channel declaration with a name attribute as an event identifier. Each channel should define its runtime behavior via: `type` attribute (channel communication type), `order` (message delivery order), `execution` (execution priority of the invoked function), `drop` (message dropping policy), `data size` and `data type`. Components can use `intent-filter` to identify themselves as *publishers* or *subscribers* of a channel and to specify the number of messages sent or read in each callback release.

One of the major benefits of using declarative manifest in our programming model is that it provides information for static verification. RTDroid guarantees the correctness of the application in two aspects: (1) **Memory boundary checking:** the total memory of a component should be equal to the sum of objects of its persistent memory, its release memory and the release memories of all its sub-components. (2) **Channel overflow checking:** The incoming message rates should not exceed the message processing rates for each channel.

4.3.1 Message Channels

A real-time message passing channel has three distinctive characteristics: (1) the associated `RealtimeHandler` must be registered in a real-time service; (2) only primitive arrays (or fixed length byte-buffers) can be exchanged on it; and (3) the number of waiting messages is bounded.

Our implementation creates a fixed-length message queue for each channel. Along with the message queue, message objects are also pre-allocated. They live in persistent memory of the receiver. Figure 4.11 illustrates the scope memory hierarchy of our design.

Queuing of messages is handled at the sender's priority, while de-queuing is done according to the receiver's priority. If a queue is full, high-priority component can steal a message from a low priority sender. When this happens, the high-priority component will be able to enqueue its message while the low-priority component will receive an exception.

Sending messages is slightly tricky as they are pre-allocated and senders should not retain references messages after the message has been sent. The protocol for sending a message is thus indirect. A `MessageClosure` is allocated by the sender, and the `genMsg()` callback is used to populate the message's payload with data. This unifies message population and queuing and is shown in Fig-

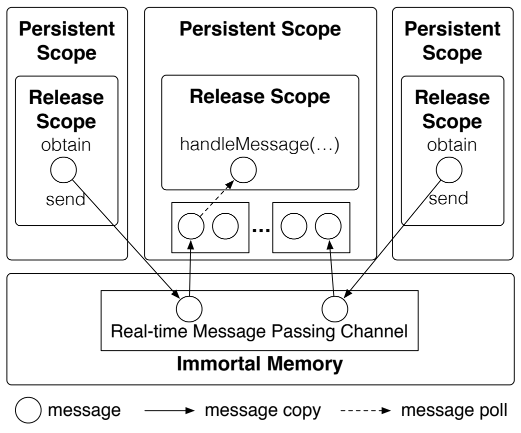


Figure 4.11: Real-time Message Passing Channel.

```java
MessageClosure c = new
    MessageClosure(){
    @Override
    public RTMSG genMsg(RTMSG
        m){
        Bundle b = m.getData();
        b.setInt(idx, 3);
        ...
        return m;
    }
};
rmsg.send("channel", c);
```

Figure 4.12: Message Passing Interface.

ure 4.12. In our running example, high-priority messages from `RecordingService` can be prioritized over the messages from non-real-time `ConfigurationUI`.

The message is served based on sender priority as a message pool. As a direct consequence, the obtain operation can fail when no messages are available as they have been given to higher priority component. An exception is raised in this case. If a high priority component attempts to obtain a message and all message objects are currently in use, the high priority thread can steal message objects that are currently being used by low priority and non real-time senders. Since messages are obtained during the send method of `MessageClosure`, all message objects in use will correspond to messages that have been enqueued, but not yet received. If the message is stolen from a construct, an asynchronous exception is delivered to the construct by utilizing the RTSJ `AsynchronousInterruptedException` mechanism.

Once a message has been obtained, the sender must copy the data to be sent from its local allocation context to the message pool of the the channel. This ensures that a sender cannot utilize or fill the allocation context of a receiver directly, the receive must choose to receive the message. Since each channel is itself bounded, non real-time senders cannot overflow the channel. The message content will only be copied to the receiver when the receiver is ready to receive and process the message. Once the message is copied to the receivers handler, the message object is returned the message object pool. This strategy keeps the amount of memory dedicated to message passing constant. The sender must utilize its own memory (heap or its release scope) to create the data that it wishes to send and cannot use system resources to store this data unless it is able to obtain a message object.

4.3.2 Broadcast Channels

Real-time broadcast channels are used to invoke callbacks of real-time components. We decouple the priority of intent delivery from invocation of callbacks which execute at their own priority, however intents are by default delivered in priority order in the same was as messages over a message channel. The main difference between intents and messages is the number of recipients. For messages this is always one and for intents this is the number of subscribers. Subscription to an intent must be declared in the manifest. Figure 4.13 shows how an intent object persists in immortal memory until it is copied to the intent queues in multiple subscribers. Although the message will be replicated for each subscriber to the intent, only one message is stored in channel itself. A count is associated with the message identifying the number of recipients subscribed to the intent. On receipt, when the message is copied to the receivers intent queue the count is decremented. The last recipient releases the message back to the message pool in exactly the same fashion as the message passing channel. The memory usage of the broadcast channel is bounded, because we pre-allocate intent objects in each subscriber's intent queue based on the size and type of data in the manifest as well as a bounded number of intent messages.

4.3.3 Bulk Data Channels

The bulk data transfer channel allows zero-copy data transfers for large messages. To support bulk transfers we extend the notion of nested memory regions with transferable nested scopes. A nested scope, which in this case encapsulates the bulk data, is removed from the scope stack (a tracking structure used for correctness guarantees) of the sending construct and pushed onto the scope stack of the receiving construct. As a result, the sender can no longer allocate into the scope, nor can the sender write to the memory of the scope. We observe that ownership transfer only works if the scope being transferred is at the top of the scope stack and the scope stack is linear. Since our programming model does not expose scopes to programmers, the constraints are ensured by the structure of the channel as well as the real- time constructs. Communication with bulk channels thus entails, a sender creating a transferable scope, populating it with data, and relinquishing access to the scope.

4.3.4 Cross-Context Channels

Cross-context channels allow Android's activities to communicate with real-time components. In this scenario communication is occurring between two separate VMs, one of which is executing the non real-time application and RT-Droid executing a real-time application. This allows us to support interaction with both legacy Android code as well as other Android applications. We note that cross-context channels are not required for communication between multiple real-time applications as the Fiji VM supports multiple VMs in the same address space.

To enable such communication an Android application must declare a service (`RTsProxyService`) that subscribes to channels declared in an real-time application that uses our real-time constructs. For communication in the other direction, a real-time application need only to subscribe to intents the non real-time application has declared in its manifest. Since our manifest is an extension of the Android manifest, no changes are require to the configuration of Android. The proxy service allows non-real-time code to send an intent to real-time components. Communication in the other direction requires the activity can subscribe to intents defined by real-time code. To preserve memory bounds, the number of intents in a cross-context channel is bounded and each intent has a fixed-length payload. Figure 4.14 shows how the bi-directional communication is established through sockets between RTDroid and Android. To do so, we leverage two proxy components in each runtime, To avoid interference, the Android proxy component is executed in heap memory, and it runs at the lowest real-time priority. The incoming message objects are translated to real-time in-

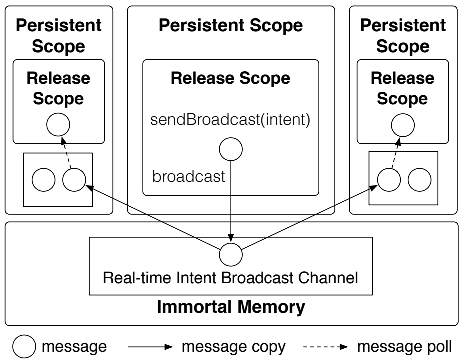


Figure 4.13: Real-time Intent Broadcast Channel.

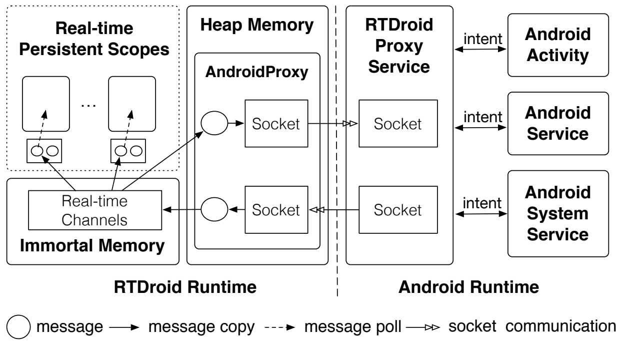


Figure 4.14: Cross-Context Channel.

tents or messages with the lowest priority and sent to the subscribing real-time components via real- time channels. Only one message is deposited into a real-time channel at a time, preventing non real-time components from exhausting memory used by real- time constructs. Non-real-time components can exhaust the heap, but this will not affect real-time components using pre-allocated memory regions.

4.4 Memory Management

For real-time applications, providing memory usage guarantees implies that the underlying system provides predictable allocation – object allocation should not be blocked by the memory usage of any other construct, and predictable reclamation – the underlying memory management scheme should not interfere with the execution of a real-time component. To achieve both, we use scoped memory, a region based memory management scheme. Scoped memory provides fixed amount of memory for real-time tasks through the usage of memory regions and predictable object allocation and deallocation within scopes. Additionally, scoped memory ensures that real-time threads executing within scopes are not blocked during GC if they only utilize scoped memory. The RTSJ provides three types of memory areas: (1) *heap memory*, which is garbage collected, (2) *immortal memory*, which is never reclaimed, and (3) *scoped memory*, which provides bounded memory regions. To guarantee referential integrity, RTSJ imposes a number of rules on how scoped memory must be used, such as (1) the objects in a scope are only reclaimed after all threads in that scope have finished, (2) every thread must enter a scope from the same parent scope, and (3) a scope with a longer lifetime cannot hold a reference to an object allocated in a scope with a shorter lifetime. Fundamentally, we leverage scoped memory to provide memory bounds corresponding to the lifetime of different computations as well as data across computations.

To provide memory boundary for RTDroid's components, we defines a *release* scope and a *persistent* scope for each real-time component. A real-time component *must* be declared with memory scopes in RTDroid's application manifest file. For an instant, a real-time service has two designated scopes, *persistent* and *release*, which are preserved and guaranted by the runtime. The *persistent* scope is never reclaimed until the service is terminated. The *persistent* of the service stores the service object and other objects that have the same lifetime as the service. The *release* scope is used for allocating local variables during an invocation of the service's callback functions. It will be cleaned up once each invocation finishes. Each component run is bound to its own thread of control that starts in *immortal memory*. This assures that its memory necessary for creating the execution context for the thread is always available, even if the construct has to be terminated and restarted. Similarly channels are allocated in *immortal memory* as well.

To enable data transfer between constructs in communication channels, we provides a *transfer* scope with the notion of object ownership. From a developer's perspective, a *transfer* scope represents an object of a memory region (`TransferScope`) used for cross-scope data transfer. It contains a *fixed-size, priority-aware* message queue with a set of APIs allowing the developer to read/write the message in the scope. More specifically, The following properties make it unique to the default scoped memory:

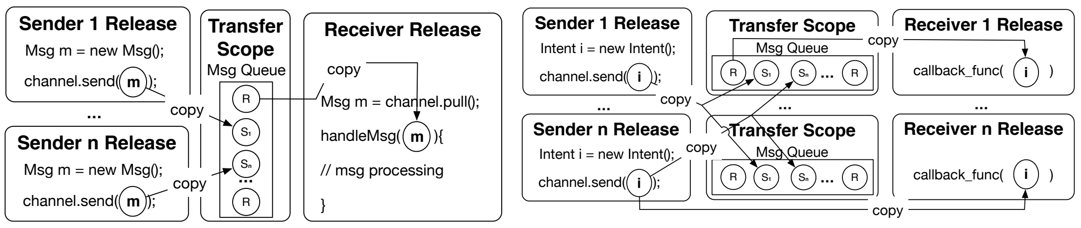
1. All objects allocated in a *transfer* scope has an owner. A default owner is set when the scope is instantiated.

2. When a *transfer* scope is created, a "placeholder" thread enters into the *transfer* scope in order to keep the scope "alive".

3. Instead of implicitly entering into a *transfer* scope, a developer uses APIs of `TransferScope` class to read/write objects in the message queue. This means that the *transfer* scope is never pushed onto scope stack, instead, it utilizes VM-level object copy or reference pointer to enable data transfer. The ownership of objects in a *transfer* scope is set internally with the invocation of its APIs. For example, when a sender sends a message to a *transfer* scope by calling its write operation, the write operation sets the sender as the owner of the written object. With a receiver, the read operation works in a similar fashion as the write operation, but the read operation will also reset the owner of the object to the default owner after reading operation finished.

4. An exit method is used for programmatically tearing down the entire *transfer* scope. When the exit method is called, it will check whether all objects in the *transfer* scope are owned by the default owner, if not an `ObjectForceRecliamationInterruption` exception is thrown to notify the owner of objects.

With these properties of *transfer* scope, we have untilized the *transfer* scope to implement two VM-level primitives and build real-time channels based on them.



(a) Real-Time Message Passing Channel (b) Real-Time Intent Broadcasting Channel

Figure 4.15: Transfer Scope with Data Copy Primitives

Transfer Scope with Data Copy: The first primitive is a data copy primitive that enables data transfer between the release scope of a sender to the release scope of a receiver via a *transfer scope*. Figure 4.15a and Figure 4.15b demonstrate data transfer procedure for the real-time message channel and the real-time intent broadcasting channel, respectively. Essentially, both of them depends on the same set of `TransferScope`'s APIs, `TransferScope.copyIn(Msg)` and `TransferScope.copyOut(Msg)`. When the transfer scope is initialized, all pre-allocated messages are owned by its receiver. A sender first allocates its message object in its own release scope. When `channel.send(msg)` is called, the message is copied from the sender's release into the *transfer* scope. All messages from senders are queued with respect to the priority of their senders. The object's owner then is set to its sender after the write operation. Later, when the receiver processes messages by copy them out the *transfer* scope, the object owner is switched back to the thread of the receiver. Due to the limited size of message queue, if all messages in a message queue are owned by different senders, it means an operation of message dropping operation is required, *i.e.*, the message with the lowest priority will be reclaimed. At the same time, an `ObjectForceRecliamationInterruption` exception is thrown to the reclaimed message's sender. Notice that the entire data transfer procedure does not allow any release scope stores a reference to objects in the *transfer* scope, both senders and receivers have to use object data copy APIs that guarded by underlying virtual machine. These APIs utilizes native compiler built-in atomic functions and read/write barriers for thread safe.

Transfer Scope with Enclosed Objects: The another data transfer primitive is a *one-to-one* data sharing primitive that utilizes *transfer* scope for bulk data transfer with a zero-copy operation. Since it is a *one-to-one* communication, the lifetime of *transfer* scope is tied to the lifetime of both ends. To achieve zero-copy operation, it is safe to have a temporary reference in their release scope to read-/write objects in *transfer* scope, since the release scope is reclaimed after every invocation of callback functions. Figure 4.16 shows how the bulk data transfer channel works with *transfer* scope. The message dropping operation is required as well as the data copy operation. However, because the sender of the channel uses a message reference to populate data, the sender can potentially delay the message enqueue operation by performing any arbitrary computation from the time when it holds the reference to the time when it calls message send function. To avoid such delay, we introduce a new interface, `EnclosedMessage`, as shown in Figure 4.16. It wraps the sender's message population logic in genMsg(), and trigger it in RTDroid's framework to encapsulate message request and message

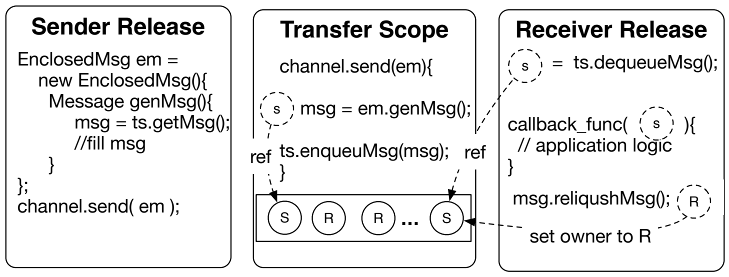


Figure 4.16: Transfer Scope for Real-Time Bulk Data Transfer Channel

data population until the enqueue operation in RTDroid's framework.

The message ownership in bulk data transfer channel is used for indicating messages that can be forcedly reclaimed when the message queue is full. Notice that the default owner in *transfer* scope is marked as 'T' in Figure 4.16. During the message population by `EnclosedMessage`, the message is marked as 'S'. After enqueue operation, the enqueued messages are marked as 'T' again. When a message is processed by its receiver, its message's owner is the receiver, marked as 'R'. Then, after processing, the message is switched back to 'T'. Therefore, when an enforced message reclamation is performed, we do not reclaim any message has been started to be processed, we only recycle a message marked as 'T' based on their arriving time, since all messages are sent from the same sender.

Simplified Scope Checks: Another benefit of the linear scope stack is that it simplify the scope checks required at runtime. Firstly, the *single parent rule* checks can be simplified with the following rules:

- ◇ **Persistent scope** must be entered from the *immortal* scope.
- ◇ **Release scope** must be entered from the *persistent* scope of the same construct.
- ◇ **Transfer scope** must never be entered.

To implement these rule, we can set the expected parent scope when these scopes are created. When a scope is entered, it just needs to check whether its parent scope (the current active scope) is the expected parent scope by comparing scope intensifier.

Secondly, the *assignment* checks can be even replaced by static compiler analysis. Table 4.1 lists a set of reference rules between different scopes. Notice

	Ref to Transfer	**Ref to Release**[ii]	**Ref to Persistent**[ii]	**Ref to Immortal**
Transfer	Yes	No	No	Yes
Release	Yes	Yes	Yes	Yes
Persistent	No	No	Yes	Yes
Immortal	No	No	No	Yes

The release scope and persistent scope of the same constructs.

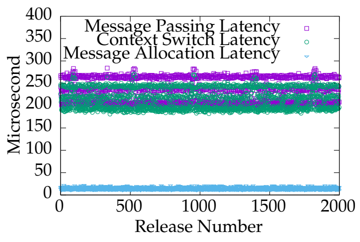
Table 4.1: RTDroid's Static Scope Check

reference rules are only considering the references between the release scope and the persistent scope of the same construct. Any cross-construct reference is forbidden.

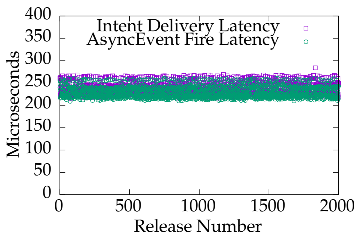
4.5 Evaluation

This section shows our experimental results of micro-benchmarks and applications in RTDroid. To evaluate the predictability and the code efficiency of our programming model, we have conducted a series of stress tests for message delivery latency and three case studies which compare task processing duration and code efficiency between RTDroid and Android. The three applications are a cochlear implant application described in Section 4.1, a UAV flight control system, jPapaBench [38], and a turbine health monitoring application. We use these case studies to compare against Android as well as RTSJ. To test the correctness of our audio framework, we have conducted stress tests of the audio framework and performed evaluation over two applications, including device coordination and distance estimation.

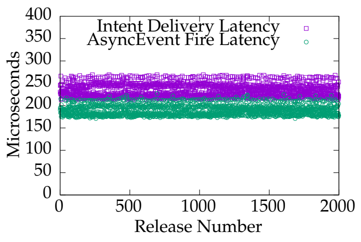
All results are collected on a Raspberry Pi Model B, which has a single-core ARMv6-based CPU with 512 MB RAM, and runs Debian with Linux preemptive kernel v3.18 and on a Google Nexus 5 smartphone, which has a quad-core 2.3 GHz Krait 400 Processor and 2GB RAM, running Android v6.0.1. On both platforms we only enable one core and fix CPU frequency. For the turbine health monitoring application, we use an external Wolfson audio codec in order to provide high-quality audio playback and capture for vibro-acoustic analysis. Raw data and plotting scripts can be found under the publications tab and cases



(a) Message Passing

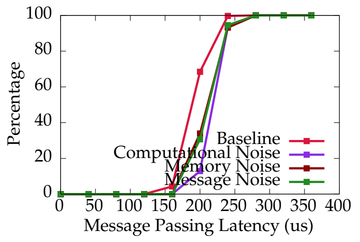


(b) Intent Broadcast

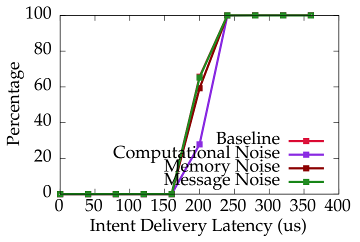


(c) Bulk Data Transfer

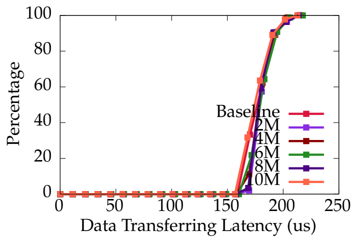
Figure 4.17: Real-time Communication Channels: Baseline Scatter Plot for Micro benchmarks.



(a) CDF for Message Passing



(b) CDF for Intent Broadcast



(c) CDF for Bulk Data Transfer

Figure 4.18: Micro-benchmarks for Real-time Communication Channels.

study code under the application tab on our website: http://rtdroid.cse.buffalo.edu.

4.5.1 Micro Benchmarks

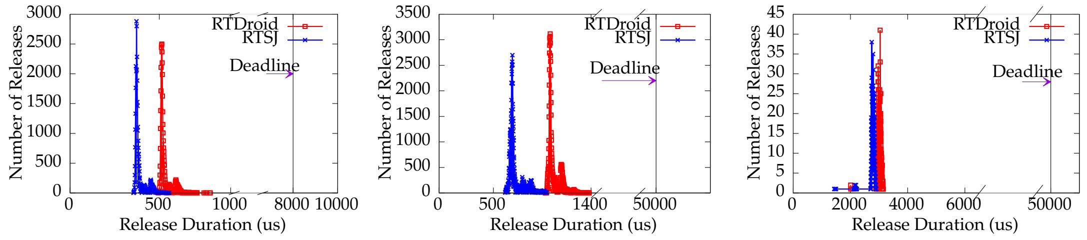
We have conducted a set of micro-benchmarks for our communication channels and audio playback latency for the audio framework. The micro-benchmark for communication channels runs two real-time services and one non real-time service. One real-time service acts as a sender that sends a message every 100 *ms* with the highest priority and one as a receiver of the message. The third service, executing in heap memory, starts 30 noise-making threads with the lowest priority to inject noise into the system. Similarly, the audio micro-benchmark has a real-time service which contains a user-defined audio session and play a fixed number of audio data from a MP3 file.

To stress the system, we use three types of noise-making threads: (1) heap noise that allocates an array of 512 KB in the heap memory every 200 *ms*, (2) computational noise that computes π every 200 *ms*, and (3) message noise for the message latency measurement, which sends a low-priority message to the receiving service every 200 *ms*, or stream noise for the audio playback measurement, which delivers a fix-number of audio samples to play every 200 *ms*.

Message delivery latency in communication channels: Figure 4.17 shows raw performance measurements for the baseline performance of our channel implementations. Message passing consists of message allocation by the sender, message delivery, and context switch from the sender to the receiver. Figure 4.17a shows this breakdown with just the sender and the receiver, and it is our baseline performance. In the figure, we plot the latency of 2000 message passing events. For each event, the message allocation latency is the amount of time it takes for a sender to instantiate a message. The message passing latency is the time taken for delivery. The context switch latency is the difference between the time the sender sends a message and before the receiver processes the message. As shown, all three types of latency are tightly bounded across all events, and there is no outlier that takes much more time to process than others. It shows that without any other background load, our implementation provides stable and predictable performance.

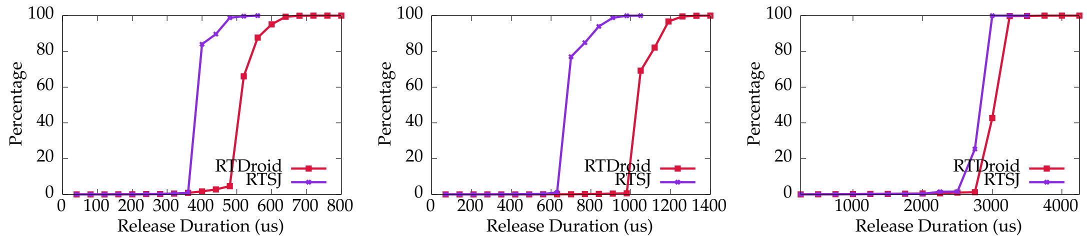
We have conducted a similar experiment to evaluate our `Intent` delivery channel. The experimental configuration is the same except that we use our `Intent` broadcast channel instead of message passing; the sender sends an `Intent` every 100 *ms*, and the receiver executes a dummy callback that responds to the `Intent`. The `Intent` delivery latency is the overall latency for each `Intent` event, and the callback trigger latency is the amount of time it takes to spawn a new callback. Figure 4.17b shows the baseline performance. Similarly Figure 4.17c shows the baseline performance for bulk data transfer, which also leverages the `Intent` mechanism for delivery, but is specialized to use the bulk data transfer channel.

Figure 4.18 shows cumulative distribution function (CDF) plots comparing the performance of all three types of channels. The CDF illustrates what percent of the total measured points are equal to or less than a given time value. For basic messaging, shown in Figure 4.18a, our implementation effectively provides an unchanged overall latency profile, regardless of the types of background



(a) Cochlear Implant: Audio Processing Duration (b) jPapaBench: Stabilization Task Duration (c) Turbine Monitoring: Audio Recording Duration

Figure 4.19: Performance Measurements on Raspberry Pi.



(a) Cochlear Implant: Audio Processing Duration (b) jPapaBench: Stabilization Task Duration (c) Turbine Monitoring: Audio Recording Duration

Figure 4.20: CDFs of Performance Measurements on Raspberry Pi.

load. We observe in Figure 4.18b similar performance characteristics for our intent broadcast channel, though we do notice additional overhead as compared to the message passing channel. This is to be expected as the intent broadcast channel results in the creation of a callback, which adds a fixed amount of overhead. Figure 4.18c shows the CDF comparing the transfer latencies with different sizes of data payload for the bulk data transfer channel. The transfer latency is the delivery time of an intent with a bulk data payload. Instead of stressing the system with noise-making threads, we increase the size of data payloads to demonstrate the performance of our *zero-copy* data transfer.

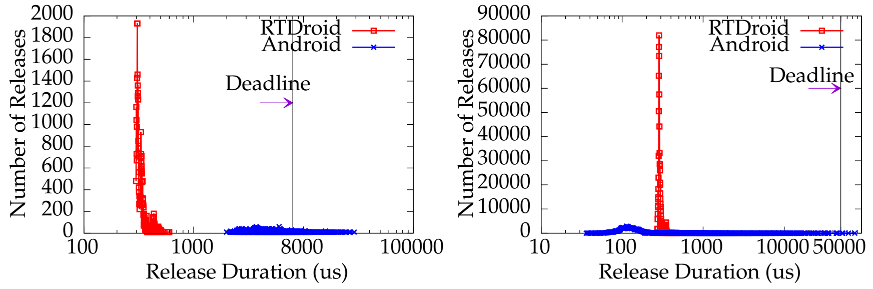
4.5.2 Comparison to Android and RTSJ

We conduct three case studies consisting of a cochlear implant application, a UAV flight control system, and a turbine health monitoring application to compare RTDroid in realistic settings against Android as well as RTSJ.

Simulated cochlear implant platform: The cochlear implant application has a

Application	Cochlear Implant			jPapaBench			Wind Turbine	
	RTDroid	RTSJ	Android	RTDroid	RTSJ	Android	RTDroid	RTSJ
Sampling Numbers	40,000	40,000	40,000	91,840	91,791	92,816	2,295	2,295
Mean (μs)	238	194	5,353	1,055	698	360	3,000	2,779
Standard Deviation (μs)	16	15	2,831	55	49	1,530	107	103
Deadlines Missed	0	0	5,160	0	0	14	0	0

Table 4.2: Task Execution Duration Statistics.

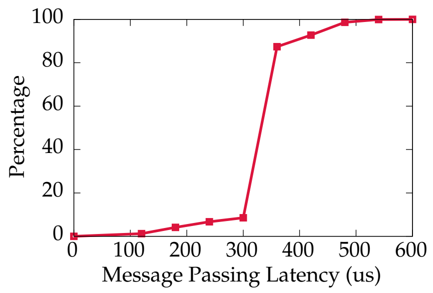


(a) Cochlear Implant: Audio Processing Duration
(b) jPapaBench: Stabilization Task Duration

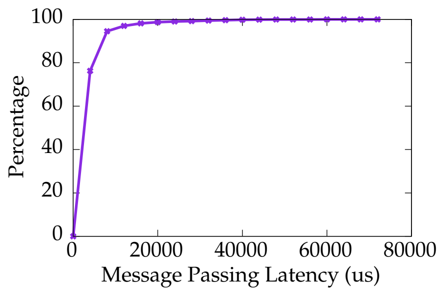
Figure 4.21: Performance Measurements on Nexus 5.

real-time service for audio processing and a real-time receiver for output error checking. Each run of the audio processing needs to acquire 128 audio samples, process them, and send processed audio output to the output receiver. This process should complete within 8 *ms* [8, 2]. Our main measurement and comparison point is this audio processing task since it has a strict timing requirement. We collected 40,000 release durations for each execution, and repeat the experiment 10 times.

jPapaBench: A real-time Java benchmark simulates autonomous flight control. We have ported it to our system as well as Android and divided the code into two services: (1) an autopilot service that executes sensing, stabilization, and control tasks, (2) a fly-by-wire (FWB) service that handles radio commands and safety checks. The original communication is replaced with intent broadcasts. We measure release durations of the autopilot stabilization task, which runs periodically with a 50 *ms* deadline, over 10 benchmark executions. Due variations in the physics simulator, each execution takes roughly 91,000 releases to com-



(a) Audio Processing Duration with RTDroid



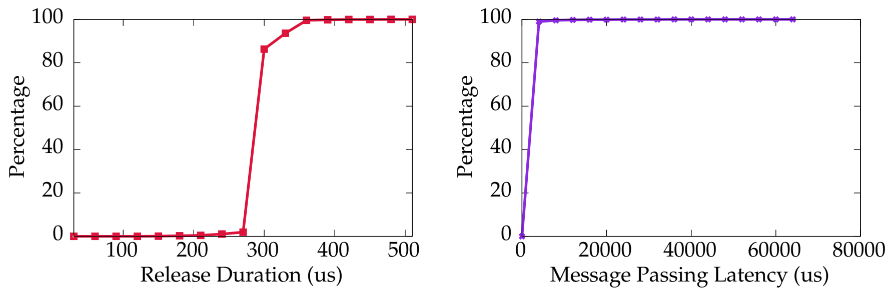
(b) Audio Processing Duration with Android

Figure 4.22: CDFs of Performance Measurements of the Cochlear Implant on Nexus 5.

plete the same flight path.

Wind turbine health monitor: The wind turbine health monitoring application was developed originally using RTSJ. We have also created a version to execute on our system. Since this application requires specialized hardware we did not implement an Android version. The application performs crack detection on turbine blades based on vibro-acoustic modulation [83]. It consists of an probing task that imposes a clean sine-wave audio tone at one side of a blade, a recording task that stores the captured audio from the other end of the blade, and an analyzing task that detects cracks by analyzing the stored audio stream. The audio recording task *must* be executed every 50 *ms* in order to capture meaningful data, and as such is our main point of measurement. We collected release durations of the audio recording task over 2 hours, and only kept releases that perform recording logic. The size of the audio buffer recorded per release is around 2MB and as such we leverage our bulk data transfer channel for communication between the recording and audio processing tasks for the version implemented in our system. The RTSJ version uses a shared memory buffer.

Figure 4.19 shows aggregated task execution durations over each application, and plots the frequency of the execution duration for each release. These results show that the use of scoped memory as well as performing communication over channels does increase the execution duration for each release, but this overhead is bounded. Android, not surprisingly, is not very predictable. Figure 4.21 shows that there is extreme variance in the duration for each release.



(a) Stabilization Task Duration with RTDroid

(b) Stabilization Task Duration with Android

Figure 4.23: CDFs of Performance Measurements of the jPapaBench Stabilization Task on Nexus 5.

To quantify the overhead imposed by our system, we report the statistical results of each application in Table 4.2. Both our system and RTSJ have similar standard deviations even in the presence of scoped memory and channel based communication. Our system's overhead is particularly visible in the stabilization task of jPapaBench. In the RTSJ version, the stabilization tasks reads sensor data from global shared memory buffers, performs at tight numeric computation, and produces control commands for the motors, which are also stored in global shared memory buffers. The version executing on our system, in comparison, receives sensor readings and sends control commands over channels, instead of reading from global buffers.

Figure 4.20 show the CDFs of the experiments detailed in Figure 4.19. We can observe that the curves of the CDFs for RTSJ performance compared to RTDroid performance are similar. Based on this observation as well as similar standard deviations presented in Table 4.2, we can conclude that RTDroid does introduce additional latency, but does not impact the predictability of the code as compared to RTSJ. Figure 4.22 and Figure 4.23 show the CDFs of the experiments detailed in Figure 4.21. Although not surprising, our numbers indicate that a non trivial portion of releases in Android exhibit significant delays, even when not in the presence of a loaded system.

Although our system does induce additional overhead when compared to applications written in RTSJ, it does provide tangible benefits in terms programability. In addition to hiding the complexity of writing code that lever-

Application	**Type of Code**	**SLoC**[3]	**Syn**[4]	**Manifest**[5]
Cochlear Implant	Common	175	0	0
	RTSJ	256	4	0
	RTDroid	235	2	69
jPapaBench	Common	3,844	0	0
	RTSJ	300	6	0
	RTDroid	230	0	149
Wind Turbine	Common	1,387	3	0
	RTSJ	539	9	0
	RTDroid	387	0	52

Table 4.3: Code Complexity Measurements.

ages scoped memory, our system also decouples configuration from application logic and simplifies interactions between components via Android like communication over channels. Table 4.3 shows code metrics over three types of code—the common code in both versions of implementation (mostly the application logic), specific code to our system, and RTSJ specific code, but excludes common libraries (*i.e.* the FFT and signal processing libraries for the cochlear implant). It shows that applications written for our system are implemented with fewer lines of code. This occurs because RTSJ requires developers to manually instantiate all tasks, and provide release logic with the multi-threading APIs. In our system all application components are declared in the manifest and the boot process initiates and starts them. Additionally, since our system uses message passing, it removes explicit programmer written synchronization between interacting components.

Chapter 5

Static Application Validation

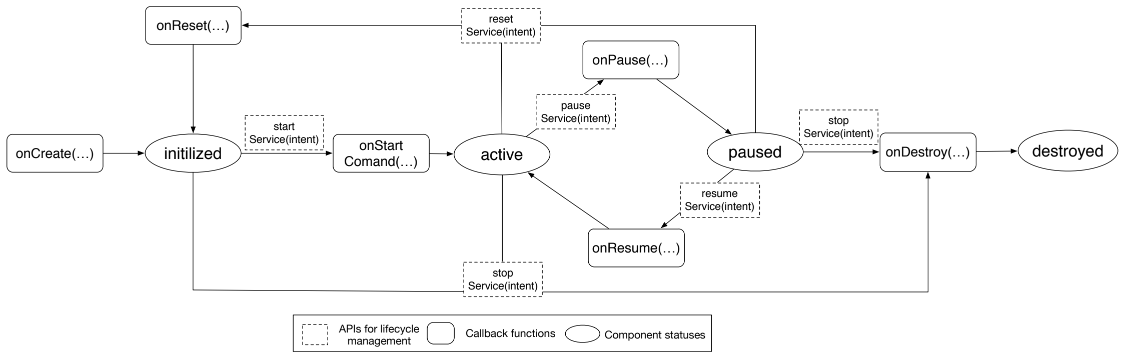


Figure 5.1: Lifecycle management of RTDroid's Real-time Service

RTDroid provides a validatoin mechanism to simulate the task scheduling and check the feasibility of application at runtime. Such validation process is part of the compilation process in FijiVM's compiler, it uses real-time properties declared in RTDroid's manifest and unilizes Cheddar, an open source real-time scheduling tools, for task simulation and scheduling feasibilities.

5.1 Background

RTDroid's programming model is derived from Android's programming model by extending Android's application components and the manifest schema. RTDroid inherits Android's event-driven nature and provides a familiar programing style to Android developers. Both Android and RTDroid programming models rely heavily on message passing based task communication, which significantly complicates the task model for validation. This section first introduces the design of the real-time components that form the bases of RTDroid's programming model. We then discuss how to model these components in a real-time task model that allows task communication.

5.1.1 Real-Time Components of RTDroid

RTDroid provides three basic application components—`RealTimeService`, `RealTimeReceiver` and `PeriodicTask`. To facilitate application validation, RTDroid synthesizes communication components for different types of message handling patterns. These are defines as RTDroid's real-time channels.

Similar to Android's components, RTDroid's real-time components are defined as abstract classes with a set of callback functions, these callback functions are used by developers to implement applications logic and invoked by lifecycle management APIs. Figure 5.1 depicts the lifecycle management of `RealtimeService`. Developers *must extend* the abstract class of `RealtimeService` and *implement* their application logic in callback functions. For example, `onStartCommand()`, which will be triggered, when other components or system services calls `startService(Intent)`. Then, once the execution of `onStartCommand()` is completed, `RealtimeService` enters into the state of "active". The following paragraphs describe the design of each component, respectively.

5.1.1.1 Real-time Service

The real-time service is a standalone component that serves as a controller over a group of nested components, such as real-time receivers and periodic tasks. Although every component in the same group has its own real-time properties, all component in a same real-time service share the same lifetime. This means that lifecycle management APIs listed in Figure 5.1 will start/stop/pause/resume both the real-time service as well as all of its nested components.

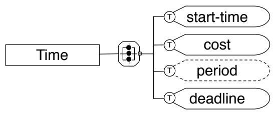


Figure 5.2: Timing Constrains

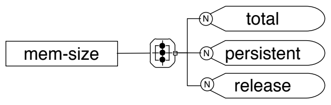


Figure 5.3: Memory Bounds

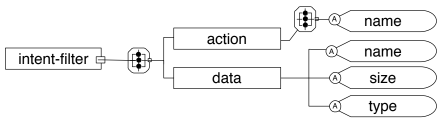
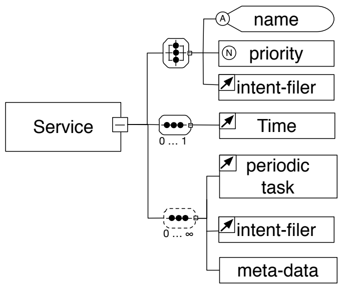
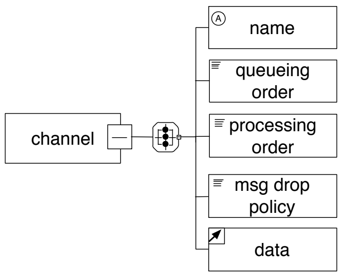


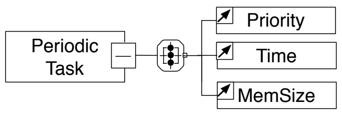
Figure 5.4: Channel Access



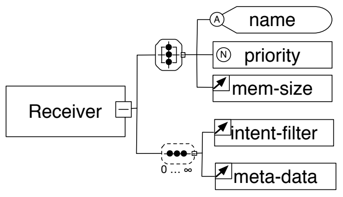
(a) Real-time Service



(b) Real-time Channel



(c) Periodic Task



(d) Real-time Receiver

Figure 5.5: Real-Time Schema for RTDroid's Components

5.1.1.2 Real-time Receiver

The real-time receiver is used to listen to specific events from system services or even other applications. It can be implemented as a standalone component or a nested component of a real-time service. It has only two callback functions: `onReceive()`, which is used to implement responses to events, and `onReset()`

to reset and clear data structures and memory associated with the real-time receiver. The callback function, `onReceive()`, is triggered by an broadcasting API—`sendBroadcast()`. The callback function, `onReset()`, can not be invoked programmatically, it is only invoked if the real-time receiver is reset by the RT-Droid runtime system.

5.1.1.3 Periodic Task

The periodic task can be only be implemented as a nested component of a real-time service. The periodic task component differs from other RTDroid components as it does not have a corresponding component in Android. An instance of periodic task has the same lifetime as the real-time service that contains it. `PeriodicTask` has only one interface, `onRelease()`, which embeds the logic to execute when the periodic task is released by the RTDroid scheduler.

5.1.1.4 Real-time Communication Channel

The real-time communication channels are built-in communication mechanisms on RTDroid. It is not an abstract class. To create a channel, a developer needs to declare the channel in the application manifest and specify the size of message sent in the channel, messages processing pattern and the maximum length of the message queue. If a component sends/receives messages to/from the channel, the component has to explicitly state itself as a producer/consumer of the channel and provides messages sending/receiving rate. There are four types of real-time channel built in RTDroid, we model them as tasks precedence and shared buffers. More details about the channel modeling will be illustrated as part of integration details Section 5.2. This section just focus on the discussion of the task model.

5.1.2 Task Model for RTDroid's Application Validation

Our validation mechanism utilizes a task model that accepts periodic tasks with hard deadlines as well as sporadic and aperiodic tasks with soft deadlines. It is based on a fixed-priority, periodic task model with rate monotonic scheduling. Feasibility tests are defined by [50, 35]. A number of assumptions were made to simplify the complexity of the feasibility tests. For example, every periodic task (τ_i) has an offset from system startup for the first release of the task and the length of time between successive releases must be a constant (T_i). Each release of a task must have a hard deadline (D_i). Most of the time such deadline (D_i) equals to T_i. Thereby, all tasks are independent of others and no interaction is allowed.

To support aperiodic and sporadic tasks and to enable task interactions, we adopt two extensions to the basic model: (1) The inclusion of aperiodic and sporadic tasks with soft deadlines, which defines the periodicity of a sporadic task as the maximal interval of two release and utilizes an aperiodic server to repeatedly check requests of aperiodic releases [74, 7, 13, 11]. (2) Introduction of shared resources and task precedence in the feasibility analysis [76, 9, 30, 37].

Notice that the minimal execution unit of RTDroid's components is the callback function, so one execution of a callback function is referenced as a release of a real-time task in this work. `PeriodicTask` is directly mapped to a real-time periodic task with hard deadlines. If one release of a periodic task cannot be completed before the next release, it means that the application is not schedulable. We model real-time services and the real-time receivers as aperiodic tasks and sporadic tasks with soft deadlines since their releases are irregular. A real-time receiver is normally used to listening to an specific system/application event that occurs repeatedly within a minimum interval, so real-time receiver is modeled as the sporadic task. The real-time service is used to grouping a set of nesting components, it normally serves as a controller to manage the lifecycle of its nesting components. There is no specific interval between two releases, so we model real-time services as aperiodic tasks. The present implementation of RTDroid uses a *per-instance* server to execute `RealtimeService` or `Realtime-Receiver`, the server periodically checks release requests and invokes callback functions.

5.2 Application Validation and Bootstrap

Section 5.1 has describe the task model in our validation process, this section presents integration details between RTDroid's real-time properties declared in its application manifest and feasibility tests in Cheddar [71]. Later, we also explain how the application boot process leverages the output of the application validation for task initialization and memory boundary enforcement.

5.2.1 Integration between RTDroid Manifest and Cheddar

RTDroid's compiler translates RTDroid's application manifest to a XML file understood by Cheddar [71] and performs the validation process, including scheduling simulation, calculating worst-case response times, and producing an upper bound of system utilization. Because RTDroid only supports an environment with a single core processor, all tests for this paper use uniprocessor hardware simulation in Cheddar. We also check memory bounds for real-time components and message bounds for real-time channels. These checks, however, are implemented in RTDroid's compiler and not in Cheddar.

Tags	**Real-Time Tasks**		
	Aperiodic Task	Sporadic Task	Periodic Task
<start>	-	-	The first release time
<cost>	Soft deadline	Soft deadline	The worst-cast time cost
<periodic>	-	MISR[1]	Periodicity
<deadline>	-	-	Hard deadline[2]

[1] The minimum interval between two successive releases.
[2] The period of a periodic task is equal to its deadline.

Table 5.1: Real-Time Semantics of Timing Elements

5.2.1.1 Timing Constraints in Scheduling Simulation

Timing constraints of real-time components are modeled into a task model with fixed-priority scheduling consisting of three types of tasks: a periodic tasks with hard deadlines, sporadic tasks with soft deadlines, and aperiodic tasks with soft deadlines. Figure 5.5 shows a complete manifest schema. Each component *must* be declared with a unique name as its identifier; the importance of the component is defined by a `priority` element. Timing constraints are specified with a `<time>` element consisting of four child elements, given as timing parameters: `<start>`, `<cost>`, `<periodic>` and `<deadline>`, as shown in Figure 5.2.

Channel Name	**Channels in RTDroid**	**Real-Time Entities**
Message passing channel	Multiple producers. Consumers inherits the priority of the message producer. A consumer per priority.	A task precedence per priority.
Intent broadcasting channel	The broadcasting communication pattern with fixed number of participants. All participants declare their rates of message producing/consuming.	A shared buffer with M/M/s/∞/N.
Bulk data transfer channel	One-to-one data communication.	A receiver as the consumer, a task precedence. Periodic task as the consumer, shared buffer with M/D/1.
Cross-context channel	Unbound incoming messages, process messages with best efforts.	A shared buffer with M/G/1.

Table 5.2: Modeling between RTDroid's Channel to Real-Time Entities

In real-time semantics, due to the restrict levels with deadlines, these timing parameters present slightly different meanings. *e.g.*, the element of `<cost>` is used as a soft deadline for the aperiodic and the sporadic task, but it is the worst-cast time cost of each release for a periodic task. Notice that the worst-case time cost is filled by application developers at the current version of RTDroid. Similarly, The element of `<periodic>` is the minimum interval between two successive releases for a sporadic task as listed in Table 5.1.

5.2.1.2 Feasibility Tests and Bounds Checking

There are two feasibility tests: (1) The schedulability test that produces an upper bound of processor utilization and worst-case response times of tasks; (2) The performance analysis with shared buffers/task precedences which leverages the theory of queuing systems to calculate two upper bounds—message waiting times and the maximum number of messages in a given buffer, respectively.

The schedulability test is based on a fixed-priority, preemptive rate monotonic scheduler in [50, 35]. The sporadic and aperiodic tasks are handled through a aperiodic server that periodically checks requests of aperiodic/sporadic tasks. There are different types serves proposed by Spuri *et. al.* [76, 72]. For performance analysis with shared buffers and task precedences, Table 5.2 shows the modeling of RTDroid's channel against queuing system theory [45, 44]. The message waiting times and the maximum number of messages are computed during the performance analysis, the later number is then used as the message bound of a real-time channel to check if any of its producer/consumer sends/receives message exceed this bound.

RTDroid utilizes on-stack memory management schema with two types of memory scopes: `<persistent>` and `<release>`. All real-time components *must* specify their expected memory usage via the element of `<mem−size>` defined in Figure 5.3 with three attributes:

1. *Release:* The memory region is used to allocate temporal objects for the execution of callback functions, this region is reclaimed after each invocation.

2. *Persistent:* The memory region is used to allocate objects that have the same lifetime as real-time components, this region is reclaimed when the associated component is destroyed.

3. *Total:* The sum of the persistent, the release, and the memory usage of all children components if any exist.

Memory bound checking is performed at two levels. The first level is to check every single component if the sum of its persistent, its release, and its children's releases is equal to its total. The next level is to check the sum of total in all component is less than the amount of memory available for the real-time application itself within the RTDroid framework.

5.2.2 Application Bootstrap

The application bootstrap process is divided into two stages: a static compile time and a runtime boot procedure as depicted in Figure 5.6. The first step of static application compilation is to validate the real-time configuration to quantify temporal constrains and check any unbounded behavior over shared buffers defined in the application manifest. Then, RTDroid's compiler emits Java bytecode that overrides the constructor of each component in which application bootstrap instantiates an instance for declared real-time components, which assign timing parameters and allocates memory bounds or message objects for the runtime. Note, RTDroid does not allow any real-time component to manually change its timing properties at runtime. This is enforced by the programming model and exposed interfaces in RTDroid. The memory bound and utilization checks are enforced via instrumented bounds checks, inserted by the compiler, in the RTDroid runtime implementation. If memory constraints are violated, a pre-allocated runtime exception is generated by the runtime system.. For example, when the memory consumption of a real-time component exceeds its memory bound, the runtime checks will throw an `OutOfMemory` exception. This ensures clean failure semantics.

The bootstrap of a real-time application leverages results of static validation and makes the application ready for computation. As Figure 5.6 shows, after the JVM is initialized, it invokes an entrance function of the application and loads generated bytecode. Then, each component is allocated based on the real-time properties defined in the manifest and registered with an internal component manager for lifecycle management during the application runtime. Memory regions and message objects are preserved to guarantee a bounded response time with shared buffers and task precedences as discussed above.

5.3 Case Study: Cochlear Implant Application on RTDroid

This section uses a simulated cochlear implant application as a case study to reports our experiences of integrating RTDroid's validation mechanism with an existing real-time framework, Cheddar. The cochlear implant can restore hearing abilities through a surgically inserted electronic device in a patient's inner ear. Figure 5.7 shows an external device used for capturing ambient audio and converting audio samples into digital signals, and an implanted device that translates signals into electrical energy and triggers implanted electrodes to simulate hearing nerves. Recently, there has been interests in replacing the external device with a smartphone for audio sampling and processing to reduce the number of devices the patient must carry [8, 2] and enable the possibility of changing the built-in signal processing algorithms for better performance. To keep the patient rapidly response in daily conversation, a fixed amount of audio samples (*128 audio samples*) must be processed on the smartphone and delivered to the implanted device every 8ms.

The previous section 4 implemented such cochlear implant application with three independent components on RTDroid, as listed in Table 5.3. It has two real-time services that control periodic tasks for audio recording and processing, respectively. The recording task sends captured audio samples to the processing task via a message passing channel (modeled as a task precedence). Audio sam-

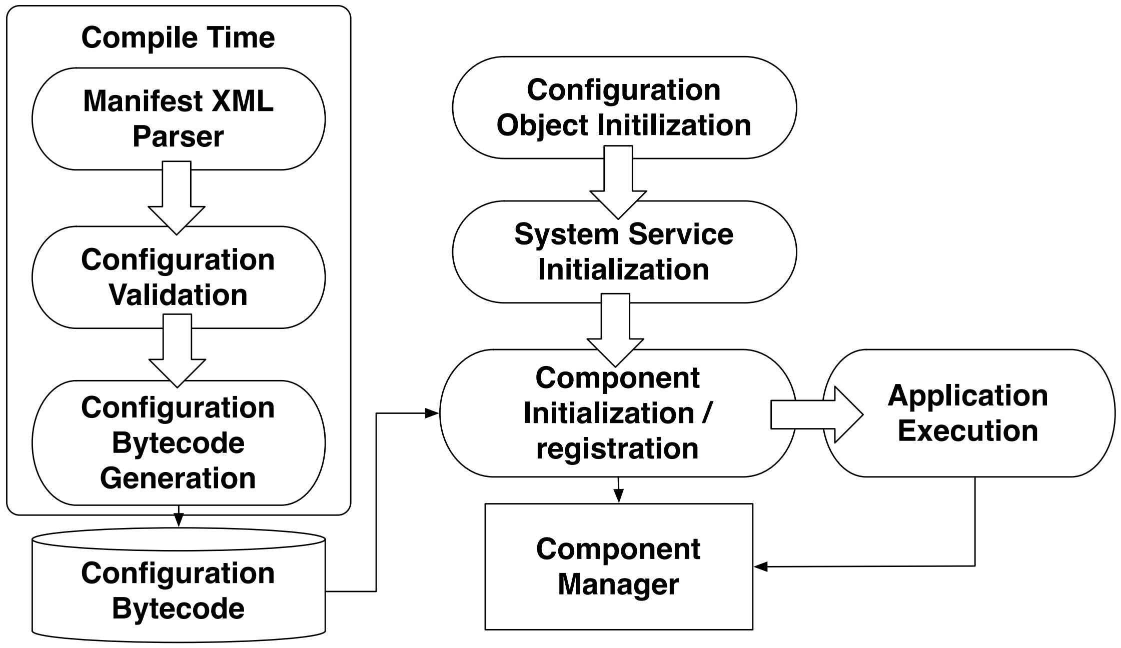

Compile Time
Manifest XML Parser
Configuration Validation
Configuration Bytecode Generation
Configuration Bytecode
Configuration Object Initilization
System Service Initialization
Component Initialization / registration
Application Execution
Component Manager

Figure 5.6: Bootstrap Procedure of RTDroid Application

ples are buffered and processed. After performing the signal processing algorithm, the processing task broadcast its processing signal to a real-time receiver (*a sporadic task*) as an output receiver. The receiver performs error-checking by simulating the behavior of implanted device. To enforce the timing constrain (8ms), we model the recording task, the processing task and the output receiver with descending priorities starting from 90 (the highest priority allowed in RTDroid application), while limits their periods with 8ms and time cost with 1ms. The cost time is estimated through an experimental measurement in [8], the time cost of computation for sampling, processing and error checking mostly less than 1ms.

We have implemented the integration process as part of the procedure in RTDroid's application compilation. The compiler translates RTDroid's application manifest to an input XML file loaded in Cheddar v3.3[1] There are a number of workaround that must be token to utilized the scheduling simulation and feasibility tests. According to our best knowledge, they are due to the fact of limitations of Cheddar's implementation, including the incompleteness of task precedence and shared buffer with the scheduling simulation and buffer feasibility tests. For example, the the simulation doesn't run with the present of aperiodic tasks. The implementation of message queuing models for shared

[1]Compiled from SVN repository revision 1835, last source code changed on 2015-08-19.

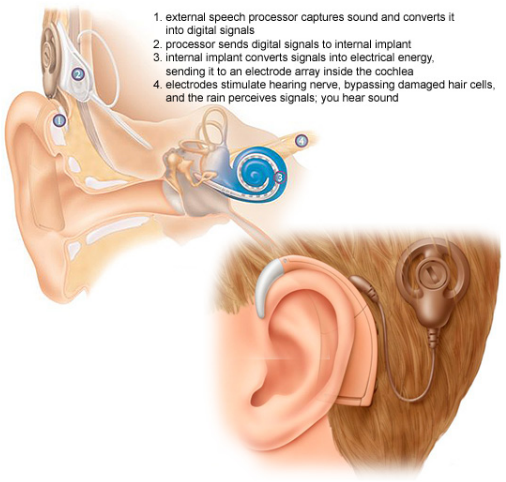
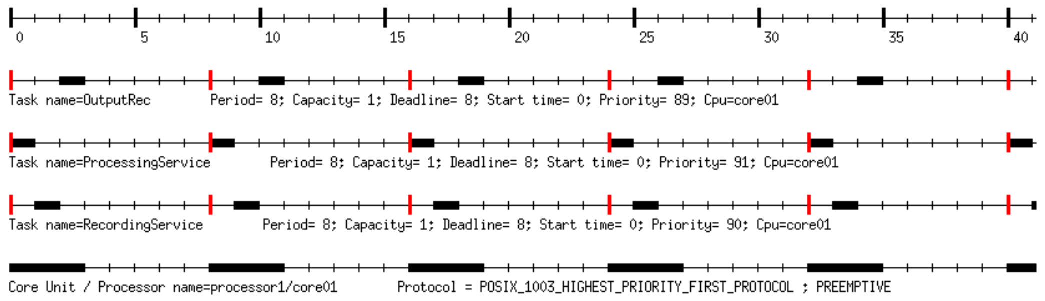

1. external speech processor captures sound and converts it into digital signals
2. processor sends digital signals to internal implant
3. internal implant converts signals into electrical energy, sending it to an electrode array inside the cochlea
4. electrodes stimulate hearing nerve, bypassing damaged hair cells, and the rain perceives signals; you hear sound
1
2
3
4

Figure 5.7: Cochlear Implant



```
Task name=OutputRec          Period= 8; Capacity= 1; Deadline= 8; Start time= 0; Priority= 89; Cpu=core01
Task name=ProcessingService          Period= 8; Capacity= 1; Deadline= 8; Start time= 0; Priority= 91; Cpu=core01
Task name=RecordingService          Period= 8; Capacity= 1; Deadline= 8; Start time= 0; Priority= 90; Cpu=core01
Core Unit / Processor name=processor1/core01          Protocol = POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL ; PREEMPTIVE
```

Figure 5.8: Scheduling Simulation of Cochlear Implant Application in Cheddar

buffers are not completed. We have made the following workaround to get task simulation and feasibility tests: (1) we manually convert the output receiver (a sporadic tasks) triggered by task precedences to a periodic task since it is triggered in every release by the processing service. (2) The feasibility tests on shared buffer can only produce meaningful results with P/P/1 model.

We report the scheduling simulation of the cochlear implant application in Cheddar, as shows in Figure 5.8. As they are in a descending order and scheduled with the policy of `POSIX_HIGHTEST_PRIORITY_FIRST`, the recording service is scheduled first, then processing service and the output receiver at last within 80ms repeatedly. Additionally, the scheduling feasibility test calculates processor utilization at 37.5 % and WCRTs are 1ms, 2ms, 3ms with the descending priorities. Given to the scheduling simulation, the message size bound of the massage passing channel is always 1, the maximum waiting time is 1 as well.

Task	**Priority**	**Start**	**Cost**	**Period** (Dead-line)
Recording Service	90	0	1	8
Processing Service	89	0	1	8
Output Receiver	88	0	1	8

Table 5.3: Real-Time Properties in Cochlear Implant

Bibliography

[1] Android and RTOS Together: The Dynamic Fuo for Today's Medical Devices. http://embedded-computing.com/articles/android-rtos-duo-todays-medical-devices/.

[2] Android-Based Research Platform for Cochlear Implants. http://www.utdallas.edu/~hussnain.ali/publications/CIAP_2015_Poster_Android_CRSS-CIL.pdf.

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

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

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

[6] Why Android Will Be The Biggest Selling Medical Devices in The World By The End of 2012. http://goo.gl/G5UXq.

[7] T. F. Abdelzaher, V. Sharma, and C. Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Trans. Comput.*, 53(3):334–350, Mar. 2004.

[8] H. Ali, A. P. Lobo, and P. C. Loizou. Design and Evaluation of A Personal Digital Assistant-Based Research Platform for Cochlear Implants. *Biomedical Engineering, IEEE Transactions on*, 60(11):3060–3073, 2013.

[9] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, page 28, Washington, DC, USA, 1995. IEEE Computer Society.

[10] B. B. 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] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190. IEEE, 1990.

[12] I. Bate and A. Burns. Schedulability analysis of fixed priority real-time systems with offsets. In *Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on*, pages 153–160. IEEE, 1997.

[13] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS '99, pages 68–, Washington, DC, USA, 1999. IEEE Computer Society.

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

[15] W. C. Blog. What OS Is Best for a Medical Device? http://www.summitdata.com/blog/?p=68.

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

[17] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[18] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 213–223, New York, NY, USA, 2003. ACM.

[19] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 324–337, New York, NY, USA, 2003. ACM.

[20] P. Brisset, A. Drouin, M. Gorraz, P.-S. Huard, and J. Tyler. The Paparazzi Solution. In *MAV 2006, 2nd US-European Competition and Workshop on Micro Air Vehicles*, pages pp–xxxx, 2006.

[21] N. Cameron, J. Noble, and T. Wrigstad. *Tribal ownership*, volume 45. ACM, 2010.

[22] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *ACM SIGPLAN Notices*, volume 42, pages 441–460. ACM, 2007.

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

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

[25] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 15–58. Springer, 2013.

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

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

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

[29] W. Dietl, S. Drossopoulou, and P. Müller. Separating ownership topology and encapsulation with generic universe types. *ACM Trans. Program. Lang. Syst.*, 33(6):20:1–20:62, Jan. 2012.

[30] J. J. G. Garcia and M. G. Harbour. Optimized priority assignment for tasks and messages in distributed hard real-time systems. In *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*, WPDRTS '95, pages 124–, Washington, DC, USA, 1995. IEEE Computer Society.

[31] M. K. Gardner. *Probabilistic Analysis and Scheduling of Critical Soft Real-time Systems*. PhD thesis, Champaign, IL, USA, 1999. AAI9953022.

[32] M. K. Gardner and J. W.-S. Liu. Analyzing stochastic fixed-priority real-time systems. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 44–58, London, UK, UK, 1999. Springer-Verlag.

[33] T. Gerlitz, I. Kalkov, J. Schommer, D. Franke, and S. Kowalewski. Non-Blocking Garbage Collection for Real-Time Android. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, 2013.

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

[35] R. Ha and J. W. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*, pages 162–171. IEEE, 1994.

[36] iOmniscient. Fall and Man Down Detection. http://iomniscient.com/index.php?option=com_content&view=article&id=155&Itemid=53.

[37] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Real-Time Systems Symposium, 1992*, pages 89–99. IEEE, 1992.

[38] T. Kalibera, P. Parizek, M. Malohlava, and M. Schoeberl. Exhaustive Testing of Safety Critical Java. In *Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 164–174, 2010.

[39] T. Kalibera, F. Pizlo, A. L. Hosking, and J. Vitek. Scheduling Real-time Garbage Collection on Uniprocessors. *ACM Trans. Comput. Syst.*, 29(3):8:1–8:29, Aug. 2011.

[40] I. Kalkov, D. Franke, J. F. Schommer, and S. 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.

[41] I. Kalkov, A. Gurghian, and S. Kowalewski. Predictable Broadcasting of Parallel Intents in Real-Time Android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 57:57–57:66, New York, NY, USA, 2014. ACM.

[42] H. Kim, S. Lee, W. Han, D. Kim, and I. Shin. SounDroid: Supporting Real-Time Sound Applications on Commodity Mobile Devices. In *Real-Time Systems Symposium, 2015 IEEE*, pages 285–294, Dec 2015.

[43] M. Kim and A. 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.

[44] L. Kleinrock. *Queueing systems, volume 2: Computer applications*, volume 66. wiley New York, 1976.

[45] L. Klennrock. Queueing systems volume 1: theory. *New York*, 1975.

[46] H. Kopetz. *"Real-time systems: Design Principles for Distributed Embedded Applications"*. Springer Science & Business Media, 2011.

[47] P. Kumar. Modal logic & ownership types: Uniting three worlds. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 778–779, New York, NY, USA, 2006. ACM.

[48] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171. IEEE, 1989.

[49] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[50] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[51] C. Maia, L. Nogueira, and L. M. Pinho. Evaluating Android OS for Embedded Real-Time Systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels, Belgium*, OSPERT '10, pages 63–70, 2010.

[52] W. Mauerer, G. Hillier, J. Sawallisch, S. Hönick, and S. Oberthür. Real-time Android: Deterministic Ease of Use. In *Proceedings of Embedded Linux Conference Europe, ELCE*, volume 12, 2012.

[53] M.-M. moazzami, D. E. Phillips, R. Tan, and G. Xing. ORBIT: A Smartphone-based Platform for Data-intensive Embedded Sensing Applications. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, IPSN '15, pages 83–94, New York, NY, USA, 2015. ACM.

[54] S. Mohan, F. Mueller, M. Root, W. Hawkins, C. Healy, D. Whalley, and E. Vivancos. Parametric timing analysis and its application to dynamic voltage scaling. *ACM Trans. Embed. Comput. Syst.*, 10(2):25:1–25:34, Jan. 2011.

[55] A. K.-L. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[56] P. Müller and A. Rudich. Ownership transfer in universe types. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 461–478, New York, NY, USA, 2007. ACM.

[57] Y. Murphy. Northrop Grumman News Release: DARPA ASPN Project Article. http://www.irconnect.com/noc/press/pages/news_releases.html?d=10029353.

[58] Northrop to Demo DARPA Navigation System on Android. http://goo.gl/bgRggD.

[59] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. 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.

[60] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the 1999 USENIX Annual Technical Conference*, USENIX ATC'99, 1999.

[61] M. Patrignani, D. Clarke, and D. Sangiorgi. Ownership types for the join calculus. In *Formal Techniques for Distributed Systems*, pages 289–303. Springer, 2011.

[62] L. Perneel, H. Fayyad-Kazan, and M. Timmerman. Can Android Be Used for Real-Time Purposes? In *Computer Systems and Industrial Informatics (ICCSII), 2012 International Conference on*, pages 1–6. IEEE, 2012.

[63] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. 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.

[64] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. 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.

[65] G. J. Rajguru. Reliable Real-Time Applications on Android OS. *International Journal of Management, IT and Engineering*, 4(6):192, 2014.

[66] L. Sha. Solutions for some practical problems in prioritized preemptive scheduling. In *Proc. 7th IEEE Real-Time Systems Symposium, 1986*. IEEE Computer Society Press, 1986.

[67] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004.

[68] L. Sha and J. B. Goodenough. ”real-time scheduling theory and ada”. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1989.

[69] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode changes in a prioritized preemptive scheduling environment. *Real-Time Systems Journal*, pages 27–60, 1989.

[70] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4):2:1–2:36, Jan. 2008.

[71] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*, volume 24, pages 1–8. ACM, 2004.

[72] B. Sprunt. *Aperiodic task scheduling for real-time systems*. PhD thesis, PhD thesis, 1990.

[73] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[74] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, Jun 1989.

[75] M. Spuri. *Analysis of deadline scheduled real-time systems*. PhD thesis, Inria, 1996.

[76] M. Spuri and J. A. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *IEEE Transactions on Computers*, 43(12):1407–1412, 1994.

[77] J. K. Strosnider. Highly responsive real time token rings. 1988.

[78] M. E. Systems. Rugged Handheld Computers Suit Up with Android on The Battlefield. `http://mil-embedded.com/articles/rugged-suit-with-android-the-battlefield/#`.

[79] D. Tang, A. Plsek, and J. Vitek. Static Checking of Safety Critical Java Annotations. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 148–154, New York, NY, USA, 2010. ACM.

[80] M. Tofte and J.-P. Talpin. Implementation of the Typed Call-by-value λ-calculus Using a Stack of Regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM.

[81] Strand-1 Satellite Launches Google Nexus One Smartphone into Orbit. `http://www.wired.co.uk/news/archive/2013-02/25/strand-1-phone-satellite`.

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

[83] Robust Distributed Wind Power Engineering. `wind.cs.purdue.edu`.

[84] Y. Yan, S. Cosgrove, E. Blantont, S. Y. Ko, and L. Ziarek. Real-time sensing on android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 67:67–67:75, New York, NY, USA, 2014. ACM.

[85] Y. Yan, S. H. Konduri, A. Kulkarni, V. Anand, S. Ko, and L. Ziarek. RTDroid: A Design for Real-Time Android. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, New York, NY, USA, 2013. ACM.

[86] Y. Yan, S. H. Konduri, A. Kulkarni, V. Anand, S. Ko, and L. Ziarek. Real-Time Android with RTDroid. In *The 12th International Conference on Mobile Systems, Applications, and Services*, MOBISYS '14, New York, NY, USA, 2014. ACM.

[87] T. Zhao, J. Baker, J. J. Hunt, J. Noble, and J. Vitek. Implicit ownership types for memory management. *Sci. Comput. Program.*, 71(3):213–241, May 2008.