Real-Time Sensing on Android

Yin Yan[♯], Shaun Cosgrove[♯], Ethan Blanton[†], Steven Y. Ko[♯], Lukasz Ziarek[♯,†]
♯: SUNY Buffalo †: Fiji Systems Inc
{yinyan, shaunger, stevko, lziarek}@buffalo.edu elb@fiji-systems.com

ABSTRACT

Modern smartphones contain many hardware and software sensors, with numerous additional sensing capabilities scheduled to be added in future iterations of the hardware and Android framework layer. This comes at no surprise as sensing is a crucial aspect of most Android applications. Unfortunately, the Android framework provides few guarantees on the delivery of sensor data to applications that require it. Similarly, the Android specification does not provide much in terms of configuration, making its programming model difficult to utilize in more traditional real-time contexts.

In this paper, we examine the Android's sensor architecture in detail and show why it is not suitable for use in a real-time context. We then introduce the re-design of the sensor architecture within RTDroid, illustrating how the RTDroid approach can provide real-time guarantees on sensing applications. We then introduce real-time extensions to the Android manifest that enable real-time sensing application writers to express static configuration requirements, using jPapaBench as our motivating example. Finally, we experimentally show that our re-designed sensing architecture can provide predictable performance for jPapaBench with memory- and computation-intensive background workloads.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; C.4 [**Performance of Systems**]; D.4.7 [**Organization and Design**]: Real-time and embedded systems

General Terms

Design, Measurement, Experimentation, Performance

Keywords

Real-time Systems, Mobile Systems, Smartphones, Android

1. INTRODUCTION

Most Android based smartphones come equipped with a number of hardware and software sensors, including sensors for motion measurement, orientation detection, as well as environmental monitoring. Android provides OS-level and framework-level support for sensor event gathering, fusion, and processing. This well-established sensor architecture makes Android a robust platform for sensing applications. Indeed, many Android applications rely and utilize the sensors.

There has been much recent interest in exploring the addition of real-time features into Android [9, 14, 27, 28]. Many of the proposed uses of a real-time capable Android system are in the consumer healthcare electronics market segment [1, 3, 4, 5] and rely on Android's sensing architecture as a core component of the application. To apply Android to such setting, real-time guarantees must be provided in the framework layer, virtual machine, and the operating system. It is also important to consider how suitable Android's sensor architecture and its associated APIs are to developing real-time sensing applications.

In this paper, we present a deeper study in Android `SensorManager`, and explain why Android's `SensorManager` is not suitable in a real-time context. Then, we discuss our re-designed sensor architecture within RTDroid [27, 28], and summarize the design decision we make in our re-design. Finally, we demonstrate how to port jPapaBench into RTDroid as a sensing application, and the evaluation result of jPapaBench with RTDroid. In addition, we also define real-time extensions of the Android manifest for streamlined development of real-time sensing applications on RTDroid and provide more information on how the Fiji compiler [20, 21] pre-allocates basic application constructs. It is also a prerequisite to supporting runtime invocation of the components callback function with restrict timing predictability.

Specifically, the contributions of this paper are:

- The design and implementation of real-time capable, Android compliant sensor architecture in the context of RTDroid. We show that Android's stock sensor architecture configuration is not suitable for certain classes of real-time applications and how we re-design the architecture to make it predictable.
- The design and implementation of a static configuration mechanism, allowing an application to provide a specification for the sensor architecture.
- A detailed evaluation of the performance and predictability characteristics of our proposed sensor architecture.

The rest of the paper is organized as follows: in Section 2, we discuss Android's sensor architecture and detail its limitations. We then introduce a re-design for a real-time capable sensor architecture in the context of RTDroid in Section 3. In Section 4, we discuss

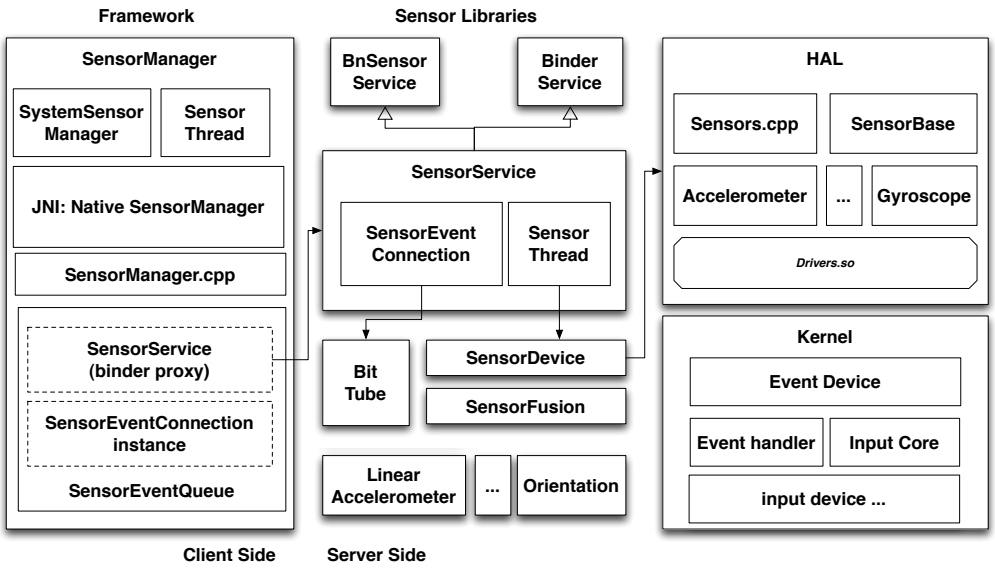


Figure 1: Android Sensor Architecture

how we extend the Android manifest to include static configuration of real-time sensing applications. In Section 5, we demonstrate our evaluation results using jPapaBench [13]. Related work is discussed in Section 6. Finally, we conclude in Section 7.

2. BACKGROUND AND LIMITATIONS

This section provides an overview of Android sensor support. We first discuss what types of sensors are available on Android. We then summarize how Android handles these sensors. We do this in a top-down fashion, starting with how Android provides sensor access to its apps. We then describe the internals of Android's sensor architecture and discuss the limitations of the current architecture in a real-time context.

2.1 Sensors in Android

There are three major types of sensors that Android supports. The first type is *motion sensors* that measure the acceleration and rotation of a device, *e.g.*, accelerometers and gyroscopes. The second type is *environmental sensors* that give the information about the surrounding environment of a device. Examples include barometers, thermometers, and photometers. The third type is *position sensors* that provide positional information for a device, such as orientation. These include orientation sensors and magnetometers.

The sensors can also be categorized into hardware sensors and software sensors. Hardware sensors correspond to physical hardware. Software sensors are those that exist purely in software and fuse sensor readings from hardware sensors. On Nexus S (our experimental device), there are 6 hardware sensors and 7 software sensors provided by Android 4.2.

2.2 How an App Uses Sensors in Android

Android apps use a set of sensor APIs to receive sensor readings. These APIs can be categorized into two types—event handlers and registration/deregistration calls. The first type, event handlers, is essentially callback methods that apps implement. Android defines two event handlers, `onSensorChanged` and `onAccuracyChanged`. `onSensorChanged` is called when a sensor reports a new value. `onAccuracyChanged` is called when a sensor's accuracy changes. Android defines these event handlers as an interface called `SensorEventListener`. Fig. 2 shows a code snippet that implements `SensorEventListener`.

```
public class ExampleSensorListener implements
     SensorEventListener {
  private SensorManager mSensorManager;
  private Sensor mLight;

  @Override
  public ExampleSensorListener() {
    mSensorManager = (SensorManager)
         getSystemService(Context.SENSOR_SERVICE);
    mLight = mSensorManager
            .getDefaultSensor(Sensor.TYPE_LIGHT);
    mSensorManager.registerListener(this, mLight,
         SensorManager.SENSOR_DELAY_NORMAL);
  }

  @Override
  public final void onAccuracyChanged(Sensor sensor,
       int accuracy) {
    // Do something here if sensor accuracy changes.
  }

  @Override
  public final void onSensorChanged(SensorEvent event) {
    // Do something with the sensor event.
  }
}
```

Figure 2: Android `SensorEventListener` Example

The second type, registration/deregistration calls, is provided by an Android framework service called `SensorManager`. An Android app retrieves a handle to this service first, and uses the handle to make registration or deregistration calls for its event handlers. This is demonstrated in Fig. 2 by the constructor code.

After an app completes the registration of sensor event handlers, the Android platform starts delivering sensor events to the app. For example, if an app registers a `onSensorChanged` handler, then the Android platform calls the event handler whenever there is a new sensor reading. We further discuss this delivery mechanism below.

2.3 Sensor architecture in Android

Android has a specialized software stack for sensor support. This stack involves four layers—the kernel, HAL (Hardware Abstraction Layer), `SensorService`, and `SensorManager`. Fig. 1 shows a simplified architecture.

2.3.1 Kernel

The bottom most layer, the kernel layer, enables raw hardware access. Android uses Linux as the base kernel and most of the mechanisms that Android uses for sensor support come directly from Linux. Namely, Android uses Linux's input subsystem (`evdev`) for sensor devices. Hardware sensors are registered as input devices (often using the standard I^2C bus). Raw sensor data access is possible through the /dev file system (`/dev/input/eventX`); a user-space process can access it through POSIX system calls such as `open()` and `read()`. Internally, the kernel stores all the sensor data for a sensor in a circular buffer. From the kernel's point of view, a sensor is just an input device; hence, every time a sensor produces a new value, the kernel reads it and stores it in the corresponding circular buffer. This means that the kernel driver is largely application-agnostic. Even if there is no process to consume any sensor data, the kernel reads all sensor updates.

2.3.2 HAL

The next layer is HAL (Hardware Abstraction Layer), which is a user-space layer that defines a common sensor interface for user-space processes. This layer hides vendor-specific details and hardware vendors must provide the actual implementation underneath the abstraction. This vendor-provided implementation essentially maps the HAL abstract interface to the device driver hardware interface for each sensor. HAL is loaded into a user-space process as a shared library.

2.3.3 SensorService

The next layer, `SensorService`, uses HAL to access raw sensor data. This layer is in fact part of a system process that starts from system boot time. The main job of this layer is two-fold. First, it re-formats raw hardware sensor data using application-friendly data structures. Second, it fuses readings from multiple hardware sensors to generate software sensor data. Thus, this layer enables complete sensor access for both hardware and software sensors.

In order to accomplish this, `SensorService` polls each sensor through HAL. Within `SensorService`, the `SensorDevice` class represents each sensor whether it be a hardware sensor or a software sensor. The `SensorFusion` class is a helper class for software sensors that fuses multiple hardware sensor readings to generate software sensor data. All sensor data is made available to the next layer, `SensorManager`, through `SensorEventConnection`, which provides a channel for sensor data access.

2.3.4 SensorManager

`SensorManager` is an Android library linked to each app at runtime. As mentioned earlier, it provides registration and deregistration calls for app-implemented event handlers. Once an app registers an event handler, `SensorManager`'s `SensorThread` reads sensor data from a `SensorService`. A `SensorEventQueue` is a buffer that holds sensor data pulled from the `SensorService`. The communication between `SensorManager` and `SensorService` takes two forms; an inter-process communication mechanism of Android called `Binder`, and a domain socket. `Binder` is used for communicating commands; a domain socket is used for data transfer.

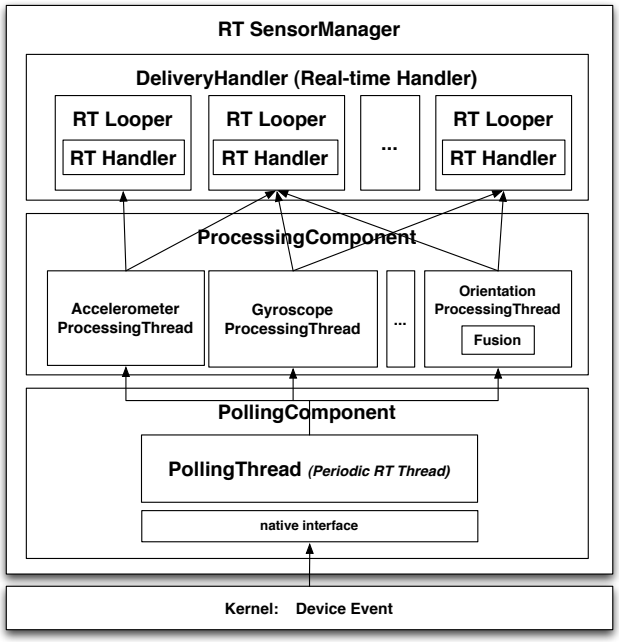


Figure 3: RTDroid Sensor Architecture

2.4 Limitations

The current Android sensing architecture does not provide predictable sensing for two primary reasons. The first reason is that Android does not have any priority support in sensor data delivery. As mentioned earlier, all sensor data delivery follows a *single* path from the kernel to apps, going through various buffers such as `SensorEventQueue`. Thus, even when there are multiple threads using a sensor having differing priorities, the sensor delivery mechanism makes no distinction between delivery to higher-priority threads and delivery to lower-priority threads.

The second reason is that the amount of time it takes to deliver sensor data is unpredictable. This is because the current Android sensor architecture relies heavily on polling and buffering to deliver sensor data. This happens at all layers—the kernel, `SensorService`, and `SensorManager`—and each layer pulls sensor data periodically from the layer underneath it and buffers the data.

3. DESIGN DECISIONS

We solve the two problems we discovered by re-designing the sensor architecture in Android. Our re-design is essentially an event-driven processing architecture that supports prioritization. Similar architectures have been proposed for high-performance Web servers [17, 26], but we have additional prioritization support built into the architecture. Fig. 3 depicts our design.

3.1 Event-Driven Architecture

An event-driven design means that each layer in our architecture *pushes* data to the layer directly above it. This is different from the current Android since each layer in Android *pulls* data from the layer underneath it. We use this architecture since it makes it easier to prioritize sensor data delivery based on the priorities of receivers, *i.e.*, app threads (as we detail in Section 3.2).

This push mechanism is implemented by thread notification. As we show in Fig. 3, each layer either has a single thread or a set of threads that process sensor data. When a thread in any layer is done with its processing, it notifies a thread in the layer above it. This chain of thread notification ends in the top-most layer, where an app's sensor event handler is called and executed.

The threads in each layer are specialized for different tasks. The bottom-most layer has one *polling thread* that pulls data out of the kernel periodically. Just like Android, this data is in a raw format and needs to be re-formatted using application-friendly data structures. For this purpose, the polling thread pushes the raw sensor data to the next layer.

The next layer has a pool of *processing threads* and we dedicate one processing thread for each sensor type. Each of these processing threads implements the same functionalities found in `SensorService` of the original Android architecture. This means that a processing thread for a hardware sensor re-formats the raw data using application-friendly data structures. A processing thread for a software sensor fuses multiple hardware sensor data and generate software sensor events. Once the data is properly generated and formatted, a processing thread notifies a thread in the next layer in order to actually deliver sensor data to apps.

The top-most layer has a pool of threads that deliver sensor data to apps. This layer relies on `RT Looper` and `RT Handler` described in our previous papers [27, 28]. The reason why we use `RT Looper` and `RT Handler` instead of regular threads is to reuse the original `SensorManager` code in Android and preserve the semantics of the original Android. `SensorManager` in Android uses `Looper` and `Handler` to execute event handlers; since our `RT Looper` and `RT Handler` are drop-in replacements for `Looper` and `Handler` that provide real-time guarantees, we can mostly reuse the existing code from Android to execute event handlers.

3.2 Receiver-Based Priority Inheritance

In addition to our event-driven re-design, we add priority support. Our goal is to avoid having a single path of delivery for all sensor data as Android does. For this purpose, we create a dedicated, prioritized data delivery path for each thread that registers a sensor event handler.

More specifically, consider a case where an app thread of priority p registers a sensor event handler for a particular sensor, say, gyroscope. In our architecture, this priority is inherited by all threads involved in delivering gyroscope sensor data to the thread, *i.e.*, an `RT Looper`, an `RT Handler`, the processing thread for gyroscope, and the polling thread. Since we create one processing thread per sensor and multiple app threads might want to receive data from the same sensor, a processing thread can be involved in delivering sensor data to multiple app threads. In this case, the processing thread simply inherits the highest priority.

Overall, our event-driven architecture with priority inheritance effectively creates a dedicated, prioritized path for each thread that registers a sensor event handler.

4. REAL-TIME MANIFEST EXTENSION

Essential properties and behaviors of an Android application are defined by a declarative *manifest* and a collection of *resources*. These items define the capabilities of the application, permissions it requires to function, mappings between logical identifiers and user interface (UI) elements such as character strings and bitmaps, and mappings between UI components and the source methods that implement their behaviors. The primary configuration file used by Android applications for this purpose is called the *App Manifest* [1].

The primary goals of the manifest are to:

- provide the name of the Java package for the application,
- describe the components of the application including activities, services, broadcast receivers, and content providers,
- name the classes that implement each of the components listed above and to define their capabilities for inter- and intra-component communication,
- declare the libraries against which the application must be linked,
- and list the permissions required by the application.

The Android Manifest is defined by an XML schema. We provide extensions to this schema for the specification of real-time configurations of the scheduling requirements of the application and its tasks. Especially, the design of our extensions is driven by the need for real-time sensing applications. For example, Fig. 4 shows the extensions to the manifest used to configure the listener task for one of the sensors utilized by jPapaBench, a real-time benchmark we ported to run on RTDroid. In this case the figure shows two listener configurations for jPapaBench tasks: `RTSimulatorIRTaskHandler`, which is responsible for processing IR data, and `RTSimulatorFlightModelTaskHandler`, which is responsible for maintaining the flight model based on sensor and positioning data.

4.1 RTDroid Boot Process

To take full advantage of the static configuration defined by the manifest, RTDroid employs a multi-stage configuration and boot process. Fig. 5 shows the entire system boot process, starting with compile time configuration processing, which is divided into five logical stages. During compile time, the Fiji VM compiler is responsible for parsing the manifest XML file. The compiler emits configuration classes for each of the configured objects in the manifest. It also emits the `Configuration Object` which provides a unique handle to all configuration classes and objects needed by the boot sequence. Booting is accomplished by four steps. First, the VM performs its own start up process and instantiates the `Configuration Object`. Once this is complete, the VM hands off control to the RTDroid system. RTDroid is responsible for initializing all system services. Once the system services have been initialized, RTDroid initializes all Android components required by the application running on RTDroid. Information for all components and system services is defined in the manifest. Lastly, an intent (conceptually an asynchronous event) is delivered to the application, which triggers the application execution.

[1]Further details on the Android Manifest including the XML schema can be found here: `http://developer.android.com/guide/topics/manifest/manifest-intro.html`

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:rtdroid="http://fiji-systems.com/schema/rtdroid/0"
    package="com.fiji.android.example"
    android:versionCode="1"
    android:versionName="1.0">
  <rtdroid:gc maxHeap="64M">
      <rtdroid:priority scheduler="NORMAL" priority="1" />
  </rtdroid:gc>
  <application android:label="jUAV" android:icon="@drawable/ic_launcher">
      <service android:name="RTSimulatorIRTaskHandler">
          <rtdroid:priority scheduler="FIFO" priority="27" />
      </service>
      <service android:name="RTSimulatorFlightModelTaskHandler">
          <rtdroid:priority scheduler="FIFO" priority="26" />
          <rtdroid:periodic>
              <period ms="25" ns="0" />
          </rtdroid:periodic>
      </service>
  </application>
</manifest>
```

Figure 4: Partial RTDroid manifest configuration for jPapaBench.

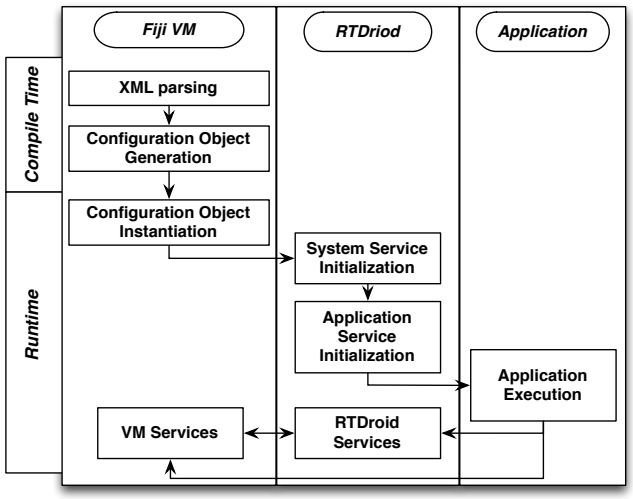


Figure 5: RTDroid configuration and boot sequence

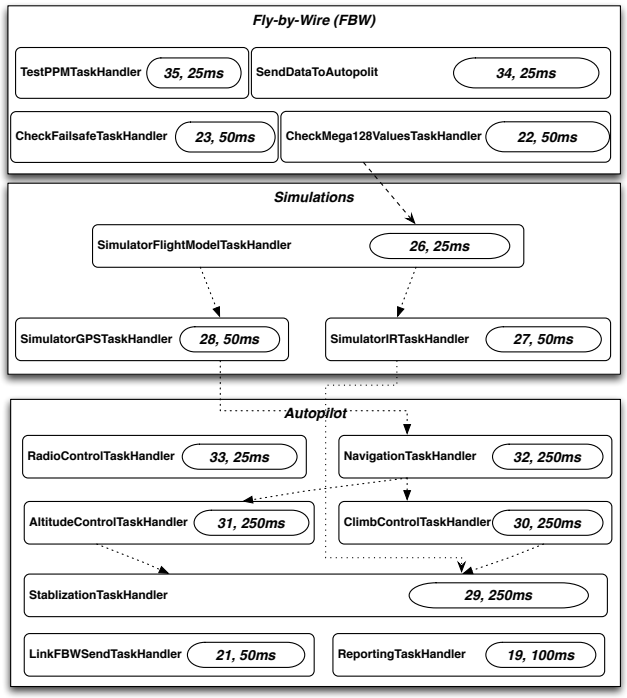


Figure 6: jPapaBench Task Dependency

5. EVALUATION

To evaluate our new design of the `RT SensorManager`, we leveraged a port of jPapaBench to execute on top of RTDroid. We first present the changes necessary to jPapaBench to execute with RTDroid and then present performance results using RTEMS running on a LEON3 development board as well as RTLinux running on a Nexus S smartphone.

5.1 jPapaBench implementation in RTDroid

jPapaBench is designed as a Java real-time benchmark to evaluate Java real-time virtual machines. It mirrors the function of Paparazzi, and changes its the cyclic execution model into separated real-time tasks in Java threads. The jPapaBench code is conceptually divided into three major modules: the autopilot, which controls UAV flight and is capable of automatic flight in the absence of other control; the fly-by-wire (FBW), which handles radio commands from a controlling station and passes information to the autopilot to be integrated into flight control; and the simulator, which collects information from each of the other modules, determines the UAV's location, trajectory, and generates input from the environment (such as GPS data, servo feedback, *etc.*). Two of these modules, the autopilot and fly-by-wire (FBW), are housed in different microcontrollers on the conceptual hardware, and the jPapaBench code simulates a serial bus between them—they have no other direct communication path. The simulator is only loosely coupled to the FBW module, but shares a moderate amount of state with the autopilot. A high-level overview of the jPapaBench system is provided in Fig. 6.

As noted by Blanton *et. al.* [6], 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 lower layers 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.

5.2 Experimental Results

To measure and validate the `RT SensorManager` design of RTDroid, we tested our implementation on two system level configurations: (1) a Nexus S smartphone running RT Linux that represents a soft real-time system deployment, and (2) a LEON3 development board running RTEMS that represents a hard real-time system deployment. The first configuration, Nexus S, is equipped with a 1 GHz Cortex-A8 and 512 MB RAM along with 16GB of internal storage. It has an accelerometer, a gyroscope, a proximity sensor, and a compass. We run Android OS v4.1.2 (Jelly Bean) patched with RT Linux v.3.0.50. For the second configuration, LEON3, is 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 [2] configuration running at 50Mhz. The development board has an 8MB flash PROM and 128MB of PC133 SDRAM.

In all our results, we show end-to-end latency as well as the breakdown of the latency. In Fig. 7 through Fig. 10, the green pluses show the overall end-to-end latency from simulated sensor event generation till event delivery in jPapaBench. As stated earlier, we feed the simulated sensor data generated by jPapaBench's simulator into an intermediate buffer first. This buffer emulates a typical kernel behavior. Then our polling thread pulls simulated sensor data out of it. Thus, the end-to-end latency measures the buffering delay in addition to the latency incurred purely in our architecture. The red circles show the buffering delay, and the blue diamonds 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 microseconds and the x-axis is the time of release of the simulator task in jPapaBench. As shown in Fig. 7 through Fig. 10, since the sensors are periodically simulated there is little difference between the Nexus S and the LEON3 and the data is generated at a rate ten times that of the hardware sensors' capacity on the Nexus S. Fig. 7 and Fig. 8 show the baseline performance of the `RT SesnorManager` on the 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

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

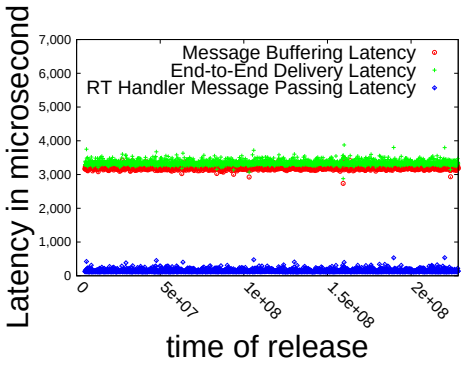


Figure 7: `RT SensorManager` performance base line on Nexus S

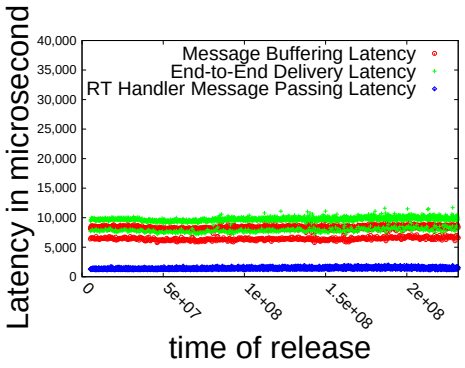


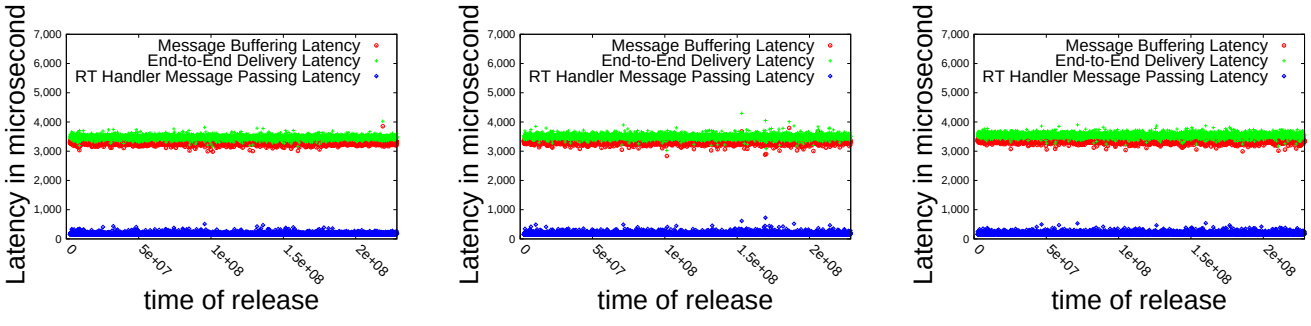
Figure 8: `RT SensorManager` performance base line on LEON3

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.

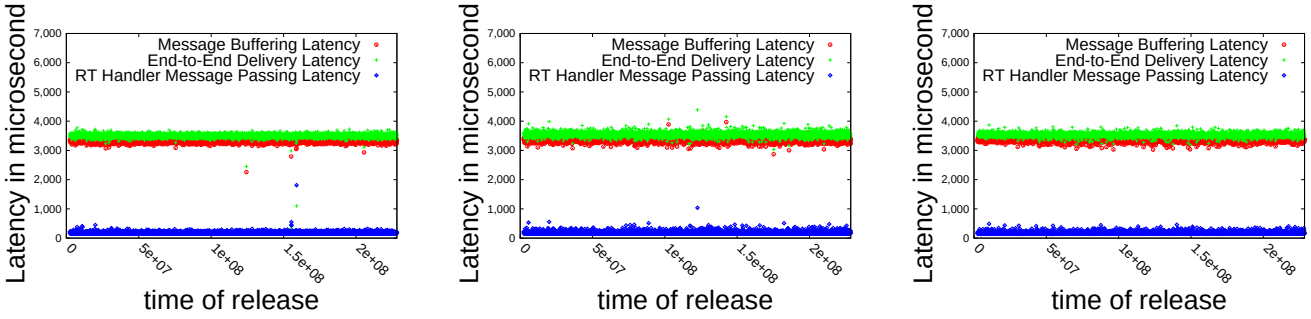
Fig. 9 and Fig. 10 show performance results obtained on Nexus S and LEON3, respectively. The figures illustrate two workload configurations for each system level configuration: 5 noise generating threads and 30 noise generating threads respectively. Interested readers can view additional experimental results and raw data on our website [3]. The baselines for both hardware platforms are provided in Fig. 7 and Fig. 8. The overall latency on Nexus S ranges from 3.2 ms to 3.8 ms, and the overall latency on LEON3 ranges from 7.5 ms to 10.1 ms. The two platforms have significant differences in their computation abilities and overall memory footprint. This contributes to the variations in latency between the platforms. In our experiments, the data polling rate is 25 ms; this means that the worst case buffering delay is 25 ms when a polling occurs just before data generation.

Fig. 9 shows the results on Nexus S with the same three types of

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

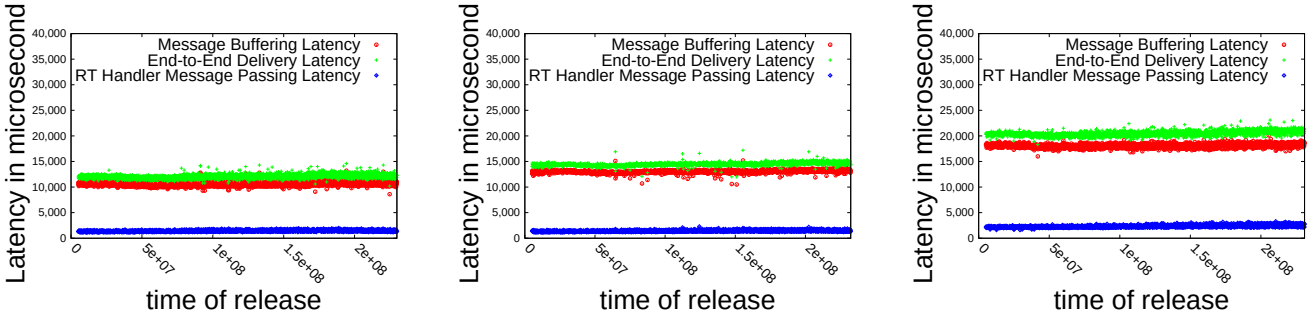


(a) 5 Memory Threads (b) 5 Computational Threads (c) 5 Listeners

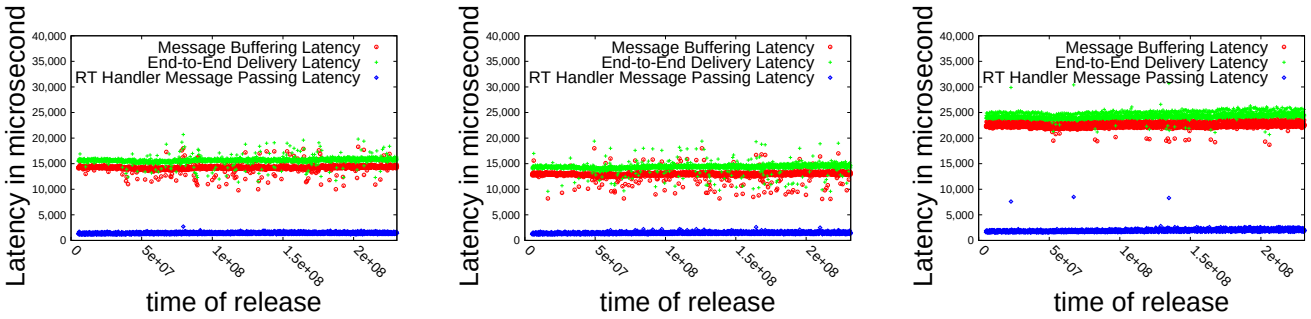


(d) 30 Memory Threads (e) 30 Computational Threads (f) 30 Listeners

Figure 9: `RT SensorManager` stress tests on Nexus-s



(a) 5 Memory Threads (b) 5 Computational Threads (c) 5 Listeners



(d) 30 Memory Threads (e) 30 Computational Threads (f) 30 Listeners

Figure 10: `RT SensorManager` stress tests on LEON3

workload—memory, computation, and listener. The performance remains the same as the baseline shown in Fig. 7. This is not too surprising as the Nexus S has a fast processor and larger main memory, creating a lot of slack time in the system as a whole. Even additional listeners do not pose any additional overheads on the system. Performance degradation only becomes observable with fifty or more noise making threads. Even with a heavy low priority load, the Nexus S is able to execute jPapaBench easily.

On LEON3, the overall latency shown in the baseline experiment in Fig. 7 is divided into two level clusters, one at 7.5 ms and another at 10.1 ms. This separation is caused by the different interval between the time that the sensor simulator task injects data into the intermediate buffer and the time that the polling thread reads data. The following execution are repeatedly scheduled in the same pattern. In Fig. 10, we observe a latency variation of 7.5 ms in Fig. 10a to 30 ms in 10f. Message buffering dominates the overall latency, and it is roughly bounded to 25 ms based on our polling rate. We obtain a few latencies that are 30 ms in Fig 10f. There is little difference between 30 memory and 30 computational threads. This indicates that predominating overhead lies within the system scheduler. There is enough slack in the system for the GC to keep up with memory allocations performed in the memory noise generating threads. In general, additional listeners pose the most stress on the system as they create contention for the data within the system. This overhead is entirely attributed to additionally buffering costs as the data is replicated for each priority level and the delivered to corresponding handlers. The overhead is proportional to the number of low priority listeners and is the combination of the largest interval in polling including the preemption cost.

6. RELATED WORK

There is much interest in exploring the use of Android in real-time contexts. For example, US and UK each have launched satellites with Android smartphones in order to explore the possibility of using Android as a satellite platform [19, 23]. Also, there are active discussions in the healthcare industry as to how to adapt Android for their medical devices [1, 2, 3, 4, 5]. In all these use cases, real-time sensing is a central functionality necessary.

In academia, making Android real-time compatible has gained recent interest. To the best of our knowledge, Maia *et al.* are the first ones to discuss potential high-level architectures [15]. They discuss four potential architectures based on the existing Android architecture. These four architectures target various use cases, ranging from running real-time applications and Android applications on top of a real-time OS to running a real-time Java VM along with Dalvik. Although there are differences in these architectures, they all start from the observation that replacing Android's Linux kernel with a real-time counterpart, such as a real-time OS or a real-time hypervisor, is necessary. Our own RTDroid starts from the same observation.

Oh *et al.* [16] characterized the suitability of Dalvik in a real-time setting. Their observation is similar to our findings; although Android performs well in general, it depends heavily on how other parts of the system at times. We believe that our approach addresses this issue.

Kalkov *et al.* [14] and Gerlitz *et al.* [9] are the first ones to add real-time guarantees in Dalvik. They discuss how to modify Dalvik to enable real-time garbage collection and also discuss how to apply an RTLinux patch to an older version of Android's Linux. Especially, their observation is that Dalvik's garbage collection is the one of "stop-all" since all other threads are blocked when it starts garbage collecting. The suggested solution is to add new APIs that enable explicit garbage collection requested by applications. This approach is complementary to ours since our automated garbage collection can supplement application-triggered garbage collection.

The sensing literature in mobile systems has been focusing on innovative applications using sensors. Examples include sleep quality monitoring [10], activity detection and monitoring [12, 18, 24], transportation detection [11], environment tracking [7, 25], logging everyday activities [8, 12], unobtrusive sound recording [22], etc. Although these applications do not necessarily need real-time guarantees, they can all benefit from tight timing guarantees that RTDroid provides for accuracy reasons, since many techniques require periodic and timely processing of sensor data.

7. CONCLUSION

In this paper, we have provided a detailed description of the RTDroid sensor architecture. We have discussed the original Android sensor architecture and the limitations of it in a real-time setting. We have also described how we re-design it in RTDroid to provide real-time guarantees while preserving the Android APIs. In addition, we have provided the description of our new real-time extensions to Android manifest. These real-time extensions enable real-time sensing application writers to express static configuration requirements, such as periodicity, priority, and scheduling algorithm to be used. We have tested the performance of our system on jPapaBench on an Nexus S smartphone and a LEON3 embedded board. Our results show that regardless of the platform and the workload, our sensing architecture provides consistent, predictable performance.

References

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

[2] Mobile medical applications – guidance for industry and food and drug administration staff. `http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/UCM263366.pdf`.

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

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

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

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

[7] Cory Cornelius, Ronald Peterson, Joseph Skinner, Ryan Halter, and David Kotz. A wearable system that knows who wears it. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 55–67, New York, NY, USA, 2014. ACM.

[8] Dan Feldman, Andrew Sugaya, Cynthia Sung, and Daniela Rus. idiary: From gps signals to a text-searchable diary. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 6:1–6:12, New York, NY, USA, 2013. ACM.

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

[10] Tian Hao, Guoliang Xing, and Gang Zhou. isleep: Unobtrusive sleep quality monitoring using smartphones. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 4:1–4:14, New York, NY, USA, 2013. ACM.

[11] Samuli Hemminki, Petteri Nurmi, and Sasu Tarkoma. Accelerometer-based transportation mode detection on smartphones. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 13:1–13:14, New York, NY, USA, 2013. ACM.

[12] Cheng-Kang Hsieh, Hongsuda Tangmunarunkit, Faisal Alquaddoomi, John Jenkins, Jinha Kang, Cameron Ketcham, Brent Longstaff, Joshua Selsky, Betta Dawson, Dallas Swendeman, Deborah Estrin, and Nithya Ramanathan. Lifestreams: A modular sense-making toolset for identifying important patterns from everyday life. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 5:1–5:13, New York, NY, USA, 2013. ACM.

[13] Tomas Kalibera, Pavel Parizek, Michal Malohlava, and Martin Schoeberl. Exhaustive testing of safety critical Java. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 164–174, 2010.

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

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

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

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

[18] Abhinav Parate, Meng-Chieh Chiu, Chaniel Chadowitz, Deepak Ganesan, and Evangelos Kalogerakis. Risq: Recognizing smoking gestures with inertial sensors on a wristband. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 149–161, New York, NY, USA, 2014. ACM.

[19] `http://www.nasa.gov/home/hqnews/2013/apr/HQ_13-107_Phonesat.html`.

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

[21] Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real time java on resource-constrained platforms with fiji vm. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 110–119, New York, NY, USA, 2009. ACM.

[22] Tauhidur Rahman, Alexander T. Adams, Mi Zhang, Erin Cherry, Bobby Zhou, Huaishu Peng, and Tanzeem Choudhury. Bodybeat: A mobile system for sensing non-speech body sounds. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 2–13, New York, NY, USA, 2014. ACM.

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

[24] Junjue Wang, Kaichen Zhao, Xinyu Zhang, and Chunyi Peng. Ubiquitous keyboard for small mobile devices: Harnessing multipath fading for fine-grained keystroke localization. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 14–27, New York, NY, USA, 2014. ACM.

[25] Yan Wang, Jie Yang, Yingying Chen, Hongbo Liu, Marco Gruteser, and Richard P. Martin. Tracking human queues using single-point signal monitoring. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 42–54, New York, NY, USA, 2014. ACM.

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

[27] Yin Yan, Shaun Cosgrove, Varun Anand, Amit Kulkarni, Sree Harsha Konduri, Steven Y. Ko, and Lukasz Ziarek. Real-time android with rtdroid. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 273–286, New York, NY, USA, 2014. ACM.

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