Real Time Sound Processing on Android

Girish Gokul, Yin Yan, Karthik Dantu, Steven Y. Ko, Lukasz Ziarek
University at Buffalo, The State University of New York
{g8, yinyan, kdantu, stevko, lziarek}@buffalo.edu

ABSTRACT

The introduction of the Internet of Things (IoT) and smartphone ubiquity has generated interest in leveraging smartphones for deploying and running sophisticated mobile sound applications. Modern smartphones have good connectivity, a significant amount of processing, and are always with us, making them an ideal candidate for envisioned applications. Many such applications have stringent latency requirements and in some cases, special I/O capabilities. One example is a class of real-time audio applications that require the ability to overlay sounds of different frequencies for joint playback. Currently, Android is unable to provide such tight timing requirement due to the design decisions for the underlying platform.

In this paper, we present a new architecture for scheduling and managing sound playback that supports overlays. We show that our prototype has competitive performance and significantly reduces latency spikes compared to stock Android. We evaluated our system on both micro benchmarks as well as sound applications including a wind farm monitoring application that leverages sound to detect micro fractures and a multiphase surround sound playback application that leverages sound overlays to coordinate between multiple devices.

1. INTRODUCTION

The introduction of the Internet of Things (IoT), and smartphone ubiquity and sophistication has generated interest in leveraging smartphones for deploying and running complex sound applications. For example, the medical device industry has proposed the usage of smartphones as an additional sound processing device for next generation cochlear implants [4]. The entertainment industry envisions using sound processing between multiple phones for surround sound playback [2, 10] and virtual music instrument applications [1]. Others have proposed to leverage smartphone-based sound processing for gesture recognition [11], voice recognition [27], calorie monitoring and tracking [20], indoor localization [16, 19, 22], sound based distance measurements [16], and as a component of large virtual and augmented reality devices [5]. Given a smartphoneś computing, communication, and sensing capabilities, and their near-permanent presence with their owner, they are ideal for these proposed applications and more.

A number of acoustic mobile applications which use audio signals for indoor localization [16, 19, 22], gesture recognition [11], device collaboration [10] and others have already been developed and demonstrated in research. Unlike traditional sound applications that play single music or audio clips, these applications not only require strict timing guarantees, but also the ability of multiplexing the sound device to record and/or play multiple audio stream sessions at the same time.

Modern smartphone software infrastructure, however, has not been designed to provide predictable low-latency execution. Many researchers have evaluated the real-time predictability of stock Android [15, 17, 21], leading to calls for adding predictability to Android. Previously, there has been some study on system architectures for smartphones that focused on using real-time operating systems and virtual machines for real-time scheduling policies and more predictable garbage collection mechanisms [12]. A real-time extension of Android's DalvikVM has also been developed [7] that reduces the effect of the GC on the execution of time-constrained components. RTDroid [25, 26] has also identified that the design of Android's framework can cause arbitrary delays at runtime, as Android's communication constructs—message passing and intent broadcasting—process their incoming messages in first-in-first-out ordering. These communication constructs are essential to the runtime system in Android, since not only the application, but other built-in system constructs also rely on them. RTDroid addresses the fundamental question of how to add real-time support to Android *as a whole* by leveraging an existing real-time kernel, an off-the-shelf real-time JVM (FijiVM [18]) and redesigned core components in its framework. SounDroid [9] has identified two key aspects of enabling low-latency multiplexed audio signals: (1) *Scheduling* audio requests with respect to timing constraints and audio frequencies of different stream sessions. (2) *Dispatching* the control of the audio device to the scheduled requests. SounDroid focuses on leveraging extra scheduling mechanisms for low-latency, on-time audio playback/record. However, SoundDroid does not provide priority-awareness for different stream sessions. Moreover, it also fails to consider the uncertainty of using Android audio manager that can cause arbitrary latency. We discuss this in more detail in Section 2.

In this paper, we present a prototype sound processing system that is capable of providing predictable, low-latency recording and playback on Android smartphones as well as the capability of overlaying and multiplexing multiple audio streams for joint playback. We implement our prototype as an extension to RTDroid [26] . Our extension provides a specialized sound processing service coupled with a declarative mechanism for specifying the sound processing requirements of an application. We also describe a prototype val-

idation mechanism for an application that requires joint playback of multiple audio streams. We provide preliminary empirical results, showcasing the performance of our system on multiple micro benchmarks as well as a surround sound benchmark that leverages multiple smartphones running our prototype, and a wind farm health monitoring application that leverage sounds processing to detect micro-fractures in wind turbine blades.

2. BACKGROUND

Before presenting our own sound processing framework, we first discuss how Android applications can use the audio system, how Android processes sound, and potential limitations of Android.

2.1 Android Audio APIs

Developers can integrate audio/video functionalities using the Android multimedia framework (`android.media` package). Here we describe the most common APIs used for playing or recording audio.

2.1.1 *MediaPlayer*

`MediaPlayer` provides the primary APIs used for playback. A developer can use a `MediaPlayer` object to play audio data from a file or a specified URI/URL. The source is passed to the object using the `setDataSource()` API and applications can leverage this object if they want to play audio from a fixed source. The `MediaPlayer` is most commonly used to play audio from compressed audio files (like MP3, AAC etc.).

2.1.2 *MediaRecorder*

`MediaRecorder` works similarly to `MediaPlayer` but it can only record audio data from the specified source to the file whose path is specified in the `setAudioSource()` and the `setOutputFile()` APIs.

`MediaPlayer` and `MediaRecorder` cannot be used if additional sound processing is to be done on the audio data played or recorded. Applications may frequently want to process the recorded data before saving, or apply filters like equalization, echo, reverb or custom filters. For this, they need control over individual buffers of data before they are saved or played. We discuss two classes that provide this functionality.

2.1.3 *AudioTrack*

An application can directly write raw PCM audio data using the `AudioTrack` object. The application can create multiple instances of `AudioTrack` to simultaneously play multiple audio streams. Figure 1 shows an Android application using `AudioTrack` to play audio. Each instance of `AudioTrack` must specify the buffer size on initialization. This buffer size determines the minimum frequency at which data should be written to the `AudioTrack` object to avoid under-runs which in turn lead to glitches and stutters in the playback. As the data is buffered in the Audio framework, the buffer size also affects the playback latency.

2.1.4 *AudioRecord*

Similar to `AudioTrack`, the `AudioRecord` can read raw PCM data from the audio hardware. The application must periodically poll the `AudioRecord` object and read audio data.

In the remainder of this section we discuss how the Audio framework handles the playback and recording, and the various sources of latency and unpredictability.

2.2 Audio Latency and Unpredictability

Although Android provides a mechanism to handle audio input and output, there is inherent unpredictability that can result in high

```
1  public void play(){
2    AudioTrack audioTrack = new
         AudioTrack(AudioManager.STREAM_MUSIC,
         SAMPLE_RATE, AudioFormat.CHANNEL_OUT_STEREO,
         AudioFormat.ENCODING_PCM_16BIT, bufferSize,
         AudioTrack.MODE_STREAM);
3    audioTrack.play(); // Start playback session
4    while(isPlaying){
5      ...
6      //Application generates audio data
7      audioTrack.write(buffer, 0, bufferSize);
           //Write audio data stored in buffer
8    }
9    audioTrack.stop();
10   audioTrack.release();
11 }
```

Figure 1: Enabling Audio Playback on Android

latency in the input and output paths due to the best-effort nature of the platform. As shown in Figure 2, there are many components involved in performing audio playback—this creates communication and buffering delays between the different components. We examine each component and explain the sources of unpredictability below.

2.2.1 *Android Audio Framework*

The first layer that handles audio in Android is the application framework. Writing audio data to the `android.media` APIs discussed earlier is a *push* operation, which means that data written is pushed down by the audio framework and buffered in the native layer. The Android audio framework employs a *push*-based operation for playback where the audio data is pushed down and buffered in the native layer, rather than the audio driver *pulling* audio data when it is ready to play more data. This is a design decision made in Android which results in significant latency. Unfortunately, the Android framework in general, including the audio framework, does not provide any guarantees for predictability [26]. This means that the push operations can also result in a high degree of latency variation. This is especially true when there are many threads competing against each other for computational and memory resources. This could result in audio glitches since buffer data delivery might not happen in a timely fashion.

2.2.2 *Native Libraries*

The next layer that handles audio is the audio native library. Among other components in the library, `AudioFlinger` provides the most important functionality as it is the native audio server. The `AudioFlinger` service

- receives audio buffers from various processes/threads playing data simultaneously
- buffers data if an application writes a buffer bigger than the native period size,
- performs resampling on the buffer if the application data sampling rate differs from the sample rate of the native audio chip
- mixes the audio data from different processes
- and writes the mixed and resampled data to the audio driver

All these operations introduce unpredictable latency into the audio path. Typically, it adds multiple periods to the latency depending on the buffer size, and is difficult to precisely quantify, making it unpredictable in terms of latency.

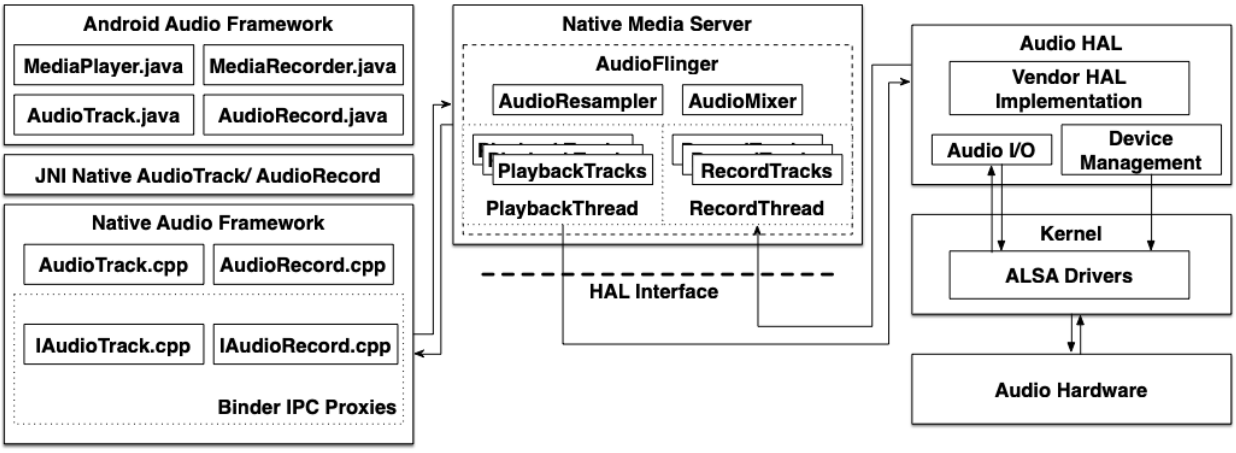


Figure 2: Audio Playback on Android

2.2.3 *Audio HAL*

The next layer that handles audio is Android's Audio Hardware abstraction layer that connects the generic framework APIs defined in `android.media` to device specific implementation of the audio driver and hardware. Communication between the media server and the audio HAL happens using standard interfaces that each HAL implementation must provide. Vendors are free to implement their own HAL code. Vendors usually implement closed-source functionality in the HAL like noise suppression or plugins to improve audio clarity on their devices. Ideally, the audio HAL should not be unpredictable in terms of latency, but Android does not have any mechanism to guarantee that they behave in a predictable fashion.

2.2.4 *Audio Driver*

The last layer for audio processing in Android is the popular audio driver, ALSA (Advanced Linux Sound Architecture). ALSA provides an interrupt driven mechanism to play or consume data from the audio hardware. A *period* is defined as the number of frames played or recorded between two sound interrupts. If the system is configured with a long *period* size, the kernel worker thread that reads/writes audio data can write a larger buffer and wait for a longer period before waking up again, this reduces unpredictability related to scheduling of the *kworker* thread but increases latency as applications must wait longer to read/write audio data. If the system is configured with a shorter *period* size, the *kworker* thread is frequently invoked to read/write data. Any unpredictability related to scheduling the *kworker* process will result in poor audio quality for the end user. The vanilla kernel used in Android does not provide any predictability guarantees in the scheduling of the *kworker* thread.

2.3 Timeliness via A New Audio Framework

The current Audio architecture for Android does not provide predictable audio performance or real-time guarantees. Previous systems like RTDroid [26] explore fixed-priority scheduling, which gives an opportunity to reduce unpredictability in the system. But we cannot use RTDroid for audio applications as:

- It does not provide a higher level API for applications to implement multimedia capabilities and functionality for the applications to directly access the sound driver to play/record audio.
- Applications cannot concurrently use the audio resources.
- Applications need to manually configure the hardware mixers to enable the audio devices to play or record data.

```
1  <audio name=S0>
2  <source type=AUDIO_SOURCE_PERIODIC_TASK
       value="Alert0" />
3  <sink type=AUDIO_SOURCE_DEVICE
       value="AUDIO_DEVICE_OUT_SPK" />
4  <streamconfig CHANNEL="1" SAMPLERATE="48000" >
5  <stream type= STREAM_TYPE_ALERT >
6  </audio>
```

Figure 3: Declaring an Audio Session in the Manifest

The following section describes an audio framework designed to allow real-time sound applications to achieve their latency bounds with minimal configuration.

3. DESIGN

In this section we propose a real-time audio framework which offers predictable, low-latency audio playback/capture and a declarative mechanism for application developers to express the real time audio requirements in their applications. First we discuss the real-time audio manifest and how developers can express their audio sessions in their applications, then we discuss the components of the audio framework that enable playback and record with predictable low latency.

3.1 Audio Manifest

Our audio manifest is an extension to the Android application manifest, where an application developer can express configuration requirements related to audio sessions in an application. Figure 3 shows an example manifest. The audio manifest decouples complex device management and configuration from actual sound processing/synthesis code. We provide the following manifest components to fully describe an audio session.

Source/Sink.

The input and output of the audio path for a session are described in the `source` and `sink` tag respectively.

We support the following source/sink types.

- **AUDIO_FILE:** This type should be used when a developer needs to play some fixed audio type from a file present in the device memory.
- **AUDIO_PERIODIC_TASK:** For applications that process audio or synthesize audio data, we can specify a periodic task that writes audio data to a buffer. On each release, the periodic task must write a single buffer of audio data of a specified size. When specified as a sink, the specified periodic task receives recorded buffers.
- **AUDIO_DEVICE:** An application can specify the hardware device from which it wants to record audio or play audio out of. Our Audio Manager provides support for the following input/output devices:

  Input:

  - `AUDIO_DEVICE_IN_BUILTIN_MIC`
  - `AUDIO_DEVICE_IN_WIRED_HEADSET`

  Output:

  - `AUDIO_DEVICE_OUT_EARPIECE`
  - `AUDIO_DEVICE_OUT_SPEAKER`
  - `AUDIO_DEVICE_OUT_WIRED_HEADSET`
  - `AUDIO_DEVICE_OUT_WIRED_HEADPHONE`
  - `AUDIO_DEVICE_OUT_LINE`

The audio framework routes input or output to a source or a sink specified in the manifest, which decouples any device management code from the audio application. Any change in the desired output path can be directly changed in the manifest itself without any logic change.

Stream Configuration.

In case either the sink or the source specified is a hardware device, there might be some hardware related configuration that a developer would need to specify. These configurations include a sample rate, the number of channels in the audio data, and a PCM data format. For these configuration parameters, we support the `streamconfig` tag in our manifest. Currently, our framework only supports a system-wide sample rate. It is our future work to provide application-specific sample rates. Section 3.3.1 describes how the stream configuration information is used to calculate the buffer sizes and the periodicity of various components to reduce the latency in the system.

Stream.

We support the `stream` tag that defines the stream type of an audio session. The rationale behind a stream is that some audio sessions can co-exist and can be played at the same time (a user should be able to hear a notification while listening to music), while other kinds of streams may require exclusive and non-preemptive use of the audio resources. This requires distinguishing different sessions based on stream types and scheduling their playback according to the following rules:

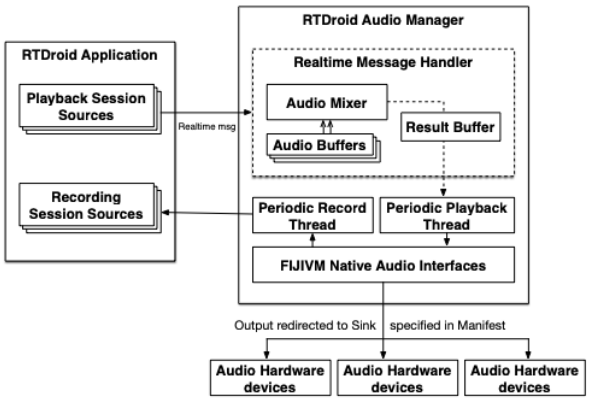


Figure 4: RTDroid Audio Framework

- **STREAM_TYPE_MUSIC:** These are regular non-mixable audio streams, only one stream of this type at a time can be played. Any new request of the same type will be queued and can only be played when the playback of the existing session of this type ends.
- **STREAM_TYPE_ALERT:** These should be used for notifications/alerts. These streams can be mixed with any stream that is currently playing.
- **STREAM_TYPE_HIGH_PRIORITY:** These should be used only for audio streams that need to be played immediately. Any session with this stream type would preempt any existing audio session. After `STREAM_TYPE_HIGH_PRIORITY` ends, the preempted audio session is resumed.

3.2 Audio Manager

The audio manager is our implementation of the sound server. We have modeled the audio manager to exist within a VM instance. Thus we will be considering only a single application execution where an application can implement multiple services which can implement multiple audio sessions. On each application launch, an instance of the audio manager is created. Figure 4 shows the overall architecture of our Audio Manager.

Throughout the lifecycle of the application the audio manager is responsible for the following.

- Audio session management: In case of multiple audio sessions, the Audio Manager makes dispatching decisions based on the stream type of the new requests.
- Mixing of audio data from different sessions in case multiple audio sessions exist at the same time.
- Providing a playback thread that periodically writes aggregated audio frames from the applications to the PCM card.
- Input/output device management: The audio manager preactivates the audio hardware and sets the configuration as specified in the audio manifest

3.3 Playback Thread

A playback thread periodically writes audio data to an output sink. If any audio data is not available for playback, the playback thread pads the buffer and writes silent frames. This is done to keep the output device active and reduce latency for playback when

$$Buffer_{Result} = \sum_{i-1}^{NumberofBuffers} \{Buffer_{result} + Buffer_i\} \quad (1)$$

$$Buffer_{Result} = Frame_i + Frame_j \forall Frame_i \in Buffer_i, Frame_j \in Buffer_j \quad (2)$$

$$Frame_{Result} = \begin{cases} \frac{Frame_i * Frame_j}{(PCM\_MAX\_VALUE/2)}, \text{if } Frame_i < \frac{PCM\_MAX\_VALUE}{2} \text{and } Frame_j < \frac{PCM\_MAX\_VALUE}{2} \\ 2 * (Frame_i + Frame_j) - \frac{Frame_i * Frame_j}{(PCM\_MAX\_VALUE/2)} - PCM\_MAX\_VALUE, \text{otherwise} \end{cases} \quad (3)$$

Figure 5: Mixing Audio Buffers

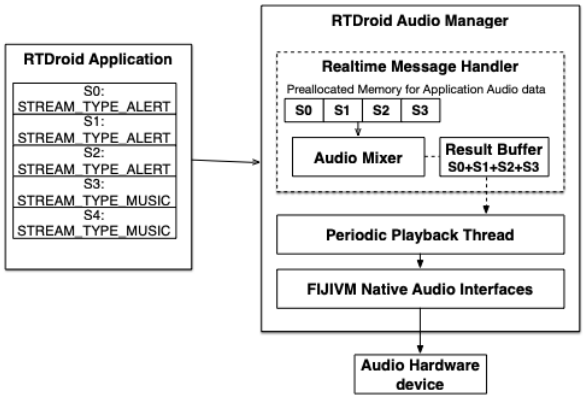


Figure 6: Mixing with Multiple Audio Sessions

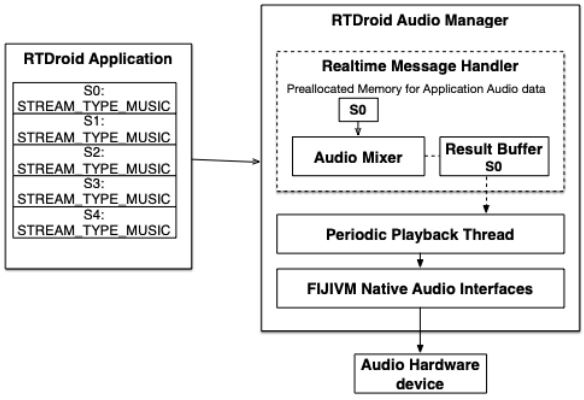


Figure 7: Multiple Audio Sessions without Mixing

audio data is ready. When an audio buffer is received by the audio manager, the playback thread plays the buffer on its next release. The playback thread is launched on application bootup.

3.3.1 *Reducing Buffering Latency*

As discussed in Section 2.2.4, we want to configure the audio system to use a shorter *period* to decrease the buffering latency in the system. The native audio chip defines a minimum *period* size depending on the device configuration specified in the `streamconfig` tag in the audio manifest. To reduce the latency we configure the system to use the lowest possible *period* size (T_{per}), by doing this we limit the maximum perceivable latency in the system to T_{per}. To ensure a glitch free playback we must ensure that the audio manager provides audio data to the hardware in a timely manner. This is done by setting the periodicity of the Playback thread and all active sessions to T_{per}. On each release, all the active audio sessions will write a buffer of size $Buffer_{min}$ bytes containing T_{per} seconds of data. The buffer size is calculated as:

$$Buffer_{min} = 2 \times T_{per} \times (SampleRate) \times (Numberofchannels)$$

Next we discuss how the buffers are mixed to achieve overlayed playback of multiple streams.

3.3.2 *Stream Mixing*

To provide overlayed playback when multiple PCM buffers are written to the audio manager they are mixed using equation 1 in Figure 5 to get a result buffer that is played by the Playback Thread. Each buffer is mixed frame by frame as shown in equations 2 and 3 in Figure 5. With these, we can achieve mixing of PCM streams without any clipping. We need to ensure all the audio sessions have the same configuration; if the configrations specified in `streamconfig` differ we cannot mix or play the audio sessions in the application.

3.4 Application Example

Considering an example application with 5 audio playback sessions as shown in Figure 6. We consider an execution scenario where session S3 has already been started, session S4 cannot preempt S3 and if Sessions S0-S2 are ready to run, their audio frames can be mixed with S3's audio frames. Thus on the next release sessions S0-S3 write a buffer containing T_{per} seconds of data to the audio manager buffer. When the audio manager receives the first buffer it copies the entire buffer to its own local buffer, whenever any new buffer is received it mixes the contents of the local buffer and the newly arrived buffer. On the next release of the playback thread, it writes the local buffer to the audio driver.

Consider a different scenario as shown in Figure 7 where all audio sessions are of type Music. On bootup only $Buffer_{min}$ bytes of memory is allocated, as there is no possibility of mixing in this application. If session S3 has already been started, on the next release S3 writes frames to the shared memory, and on playback thread's next release it finds only a single buffer which it directly plays on the PCM card.

4. IMPLEMENTATION

The sound framework is built as an extension to the RTDroid framework, which uses Fiji real-time JVM. Our audio manager and playback thread discussed in 3.2 and 3.3 are implemented within RTDroid using the constructs provided. For device/PCM card management, we implement a native audio layer.

4.1 Audio Manager

The audio manager is implemented as a real-time service on RTDroid. On application bootup an instance of the audio manger is

Session Name	Type	Start Time(s)	Duration (s)
Music session	MUSIC	0	30
Alert-tones 1	ALERT	10	10
Alert-tones 2	ALERT	15	8

Table 1: Description of Audio Sessions Invoked

spawned. On service start the audio manager sets up the device configuration and opens up the PCM card. The device configuration parameters specified for each session are first verified and checked to make sure of the following.

- All audio sessions use the same sample rate. The periodicity of the playback thread and the size of a shared buffer is calculated according to the sample rate. If there are multiple sessions with different sample rates, we will have to buffer data, incurring additional latency.
- The sample rate specified is natively supported by the PCM card. If the sample rate is not natively supported, we may have to resample the frames causing additional latency.

After configuration validation, the audio manager opens the PCM card using a native API implemented in the Fiji VM. This preactivates the audio path for any future session and helps reduce the device activation latency associated with setting up mixer paths for the hardware.

As discussed in Section 3, there is an optimal buffer size corresponding to each sample rate specified by the audio hardware. Each application needs to write this buffer on every release. To facilitate faster transfer of audio data from applications to the audio manager, the audio manager exposes a real-time message passing channel. This message passing channel is the real-time equivalent of Android's `Handler` and `Message` interface. The real time message passing channel can contain a fixed number of messages of a fixed size. Applications use this message passing channel to send their audio data on each release. The audio manager implements a `RealTimeHandler` which receives the buffers. This `RealTimeHandler` is where the buffers are mixed in case of multiple applications playing mixable audio. The buffer mixing logic is implemented in `handleMessage` which is invoked each time the `RealTimeHandler` receives a buffer from an application.

4.2 Playback Thread

RTDroid provides a periodic task construct used for any periodic operation. As audio playback/record is a periodic activity, we chose to model the playback thread as a periodic task. The periodicity of the playback thread has been discussed in Section 3.3.1. On each release audio frames are written to the PCM card. If there are no applications writing audio data, the playback thread pads the PCM card with silence frames. By this we ensure a bound latency as the device is still enabled and ready to play any audio frames generated by an application.

The audio manager utilizes the native audio APIs for opening, closing, reading from, and writing to the PCM card. These native APIs were implemented in the Fiji VM using the tinyALSA PCM libraries.

5. EVALUATION

To evaluate the predictability of our audio manager, we have conducted three sets of experiments that compare the timing performance of our audio manager to Android's audio framework. The first set of experiments is a set of micro-benchmarks that measure audio streaming latency. The second experiment is a synthetic application that simulates a "Mobile Theater" application, for surround sound, described in SounDroid [9, 10]. The last experiment is a real-world acoustic application for wind turbine health monitoring.

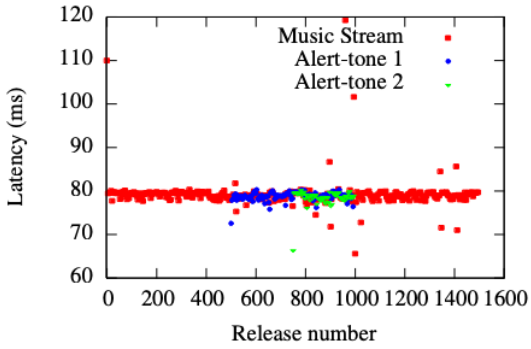


Figure 8: Baseline Measurements of Audio Streaming Latency: Android shows a higher variation in latency

The experimental results of the micro-benchmark and the simulated "Mobile Theater" application are collected on a Nexus 5 smartphone, which has a quad-core 2.3 GHz Krait 400 Processor with 2GB RAM, and runs Android M (Marshmallow). For precise timing, we enable only one core with the CPU frequency fixed. The wind turbine application runs on Raspberry Pi Model B, as an external codec is required to enable high-quality audio playback and capture for vibro-acoustic analysis.

5.1 Micro-benchmarks

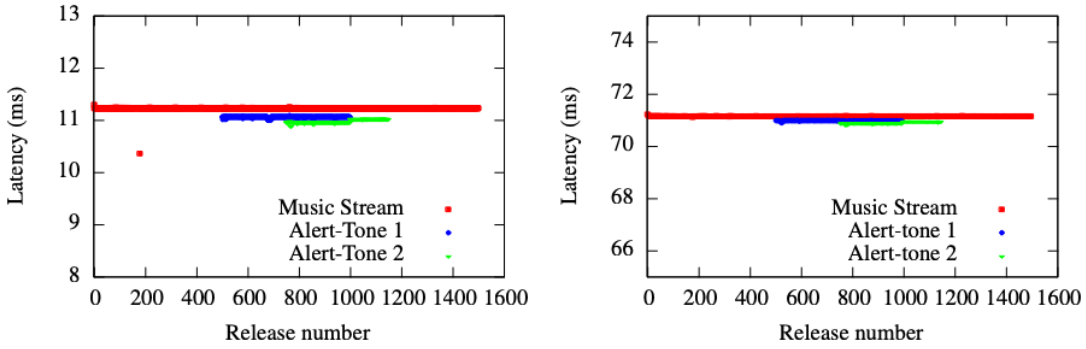
To benchmark the audio streaming latency with stream mixing from multiple sessions, our micro-benchmark application consists of three audio streaming sessions with different lifetimes, as listed in Table 1. The music session shares the same lifetime as the application, the other alert-tone sessions start at 10 seconds and 15 seconds after the application starts, respectively. As mentioned in Section 3, all the applications write to the audio framework, where the audio data from multiple sessions is mixed and then played on sound devices.

We implement this micro-benchmark application to use the audio manager in both Android and RTDroid, and measure the audio streaming latency by taking the difference of two timestamps: one is the time when each audio stream data is written to the audio framework, another one is the time when the mixed buffer is delivered to the hardware device.

Ideally, the second timestamp should have been taken within the sound card driver before the mixed audio data is written to the hardware registers. However, we do not have access to the source code of the sound card on the Nexus 5. Instead, we instrument the HAL interfaces and record the timestamp when the HAL interface is used to write audio data to the audio hardware driver.

5.1.1 Comparison of Base Line Tests

Figure 9 shows the audio streaming latency of each stream session on Android and RTDroid. The x-axis is the delivered sequence number of the mixed buffer in the playback thread in our audio manager, the y-axis is the streaming latency in milliseconds. Figure 8 shows the streaming latency of Android's audio manager ranging from 66 ms to 110 ms. The wider variation of the streaming latency on Android is caused by the expensive IPC communi-



(a) RTDroid (20 ms Playback Thread)

(b) RTDroid (80 ms Playback Thread)

Figure 9: Baseline Measurements of Audio Streaming Latency for Different Streams: RTDroid has almost no variation, regardless of the periodicity.

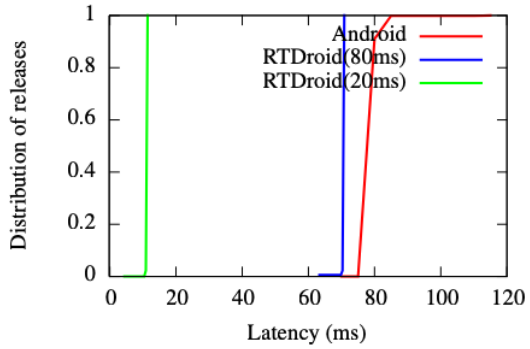


Figure 10: Audio Streaming Latency Comparison of Android and RTDroid: Android has much variation, while RTDroid does not.

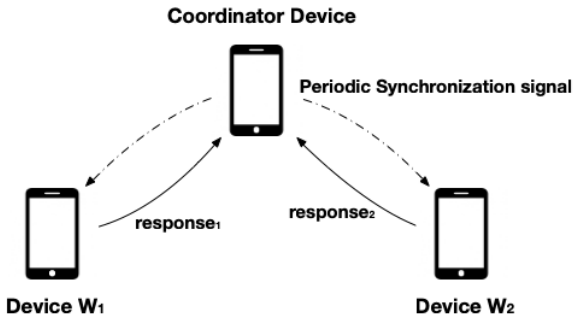


Figure 11: Surround Sound Application

cation of Android and a shared buffer between different sessions. Notice that the minimum period size supported by Android's audio framework is 80 ms. However, the audio driver on the Nexus 5 in fact provides native support for a 20 ms period size.

Thus, we run the micro-benchmark on RTDroid with both 20 ms and 80 ms periodicities, as shown in Figure 9a and Figure 9b. Since Android's audio framework does not support the 20 ms periodicity, we only run our micro-benchmark using the 80 ms periodicity.

As shown, the latency in Figure 9a is lower, as a lower period size results in lower buffering delays. During the execution of these two experiments, we do not observe any missed deadline for streaming delivery. This means there are sufficient time slots during our experiment to perform stream buffering and mixing.

Figure 10 shows an overall comparison between the CDF of streaming latency on Android and RTDroid with different configurations. Once again, Android's audio framework shows a large variation in latency. Our audio manager shows very little variation in latency for both 20 ms and 80 ms periodicities.

5.1.2 *Stress Tests*

To evaluate the predictability of our audio manager, we measure the streaming latency under the following three types of noisy payloads.

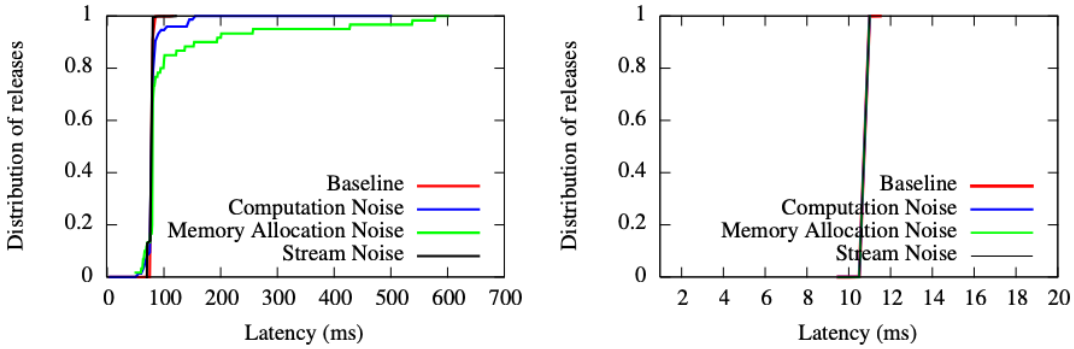
- Memory noise that allocates an array of 512KB in the heap memory
- Computational noise that computes π for 5000 iterations
- Multiple session noise that creates additional stream sessions, which can complicate the audio mixing computation

The first two noisy payloads are implemented to run in independent threads with lowest priority and 20 ms periodicity. The multiple session noise is implemented by declaring them in RTDroid's manifest.

We run the micro-benchmark with each type of payload on Android and RTDroid. For memory and computation payloads, we run the experiments with 20 noise making threads. For multiple session payloads, we run the experiment with up to 20 additional sessions. Figure 12a is a CDF of the latencies on Android. The figure shows that the presence of system load directly effects the streaming latency on Android with the original audio framework. However, Figure 12b, which presents the CDF of the streaming latencies on RTDroid with a 20 ms periodicity, shows that the system performs consistently under various noisy conditions with our audio manager.

5.2 Mobile Theater

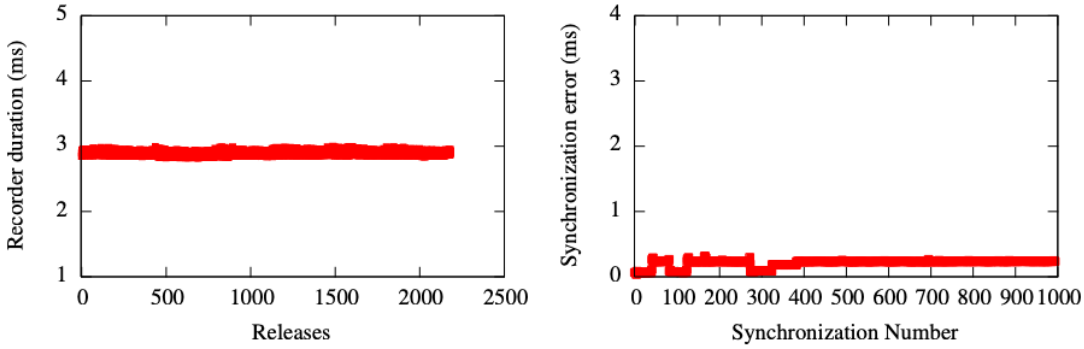
We demonstrate the suitability of using our real-time audio manager for time constrained acoustic applications by implementing a



(a) CDF of Streaming Latency on Android

(b) CDF of Streaming Latency on RTDroid

Figure 12: Variation of Audio Streaming Latency Under Various Noisy Payloads



(a) Turbine Monitoring: Audio Recording Duration

(b) Surround Sound: Synchronization Error in Arrival Time Measurement

Figure 13: Results for Real-time Audio Applications

surround sound application similar to "Mobile Theater" described by Kim *et al.* [9]. The "Mobile Theater" application and other indoor localization and coordination applications require the ability of sending multiple overlayed inaudible streams to multiple devices for coordination. Our surround sound application uses a Nexus 5 smartphone as the coordinator, and two other Nexus 5 smartphones as worker devices (W_1, W_2). As Figure 11 shows, the coordinator plays an inaudible tone for synchronization every 200 ms. In response to the synchronization tone, W_1 and W_2 play different inaudible tones with different frequencies as response signals ($response_1$ and $response_2$). The coordinator collects the arrival times of $response_1$ and $response_2$. Then, the coordinator receives the signal spectrum of the recorded response signal and measures the round-trip time between the time when the synchronization signal is sent and the time when the response signal is received. It then performs triangulation for localization. For accurate localization on such applications, a high degree of synchronization accuracy (i.e., 1 ms) has to be guaranteed [10, 11, 16, 19].

In our implementation of the surround sound application, we measure the synchronization error as the difference between the arrival time of two response signals from W_1 and W_2. Figure 13b presents the synchronization errors over 1000 synchronization signals. All of our synchronization errors are lower than 1 ms, ranging from 0.03 ms to 0.32 ms. Thus, we are confident that our real-time audio manager is suitable for implementing audio applications with timing constraints.

5.3 Wind Turbine Health Monitoring

The final artifact we evaluate is a port of a wind turbine health monitoring application developed for the Robust Distributed Wind Power Engineering project [23]. The original application is implemented in a subset of the Real-Time Specification for Java (RTSJ) [6]. This application implements a crack detection algorithm for turbine blades based on vibro-acoustic modulation. It consists of imposing a clean audio tone of several kHz in frequency on a turbine blade via a transducer mounted on the blade, recording audio from the blade via a transducer mounted at a different point, and analyzing the spectrum of the induced tone [14]. It also performs other tasks of minimal complexity, such as configuring the related audio codecs and external hardware demultiplexers.

The original RTSJ application consists of three periodic real-time threads (a controller, an audio prober, and an audio recorder) and an analysis thread that is not directly mapped to a periodic release schedule. The audio prober and recorder directly utilize PCM devices for audio probing and recording via native helper functions. It additionally requires a real-time controlled-blocking communication class for passing large batches of sample data between the recording thread and analysis thread. To port this application to RT-

Droid, we map the first three periodic real-time threads to RTDroid periodic `Services` and the analysis thread to a real-time `BroadcastReceiver`. Instead of directly interfacing with PCM device, RTDroid application defines two STREAM_TYPE_MUSIC sessions for audio playback and audio record using RTDroid's audio manager. The prober service associates with a playback session that plays a generated sine wave every 50 ms. Similarly, the recorder services associates with a recording session and recording buffer. The prober service can access its recording buffer and perform analysis algorithms, when it is periodically released. Communication between these services and broadcast receiver uses real-time `Intents`, and the real-time communication class is no longer required due to the communication constructs provided directly by RTDroid.

The wind turbine health monitoring application requires tight timing requirements not only from the system for computation but also from the sound system to perform the audio recording task. The application has an audio recording module that should operate within a strict deadline (50 ms). We run the wind farm application using our audio manager for 2 hours and measure the execution duration of the task. Figure 13a shows the frequencies of the execution durations. We can see that there are no deadline misses and the durations are well within 50 ms.

6. RELATED WORK

Android applications with real-time timing contraints and low latency audio processing require significant development effort. `OpenSL ES` [3] has been customized and integrated into Android's NDK, which provides low latency audio playback, recording, and other high-performance audio features. It mainly serves as an alternative to the `MediaPlayer` and `MediaRecord`. However, such integration of `OpenSL ES` [3] only provides native interfaces to the Android application developer. Use of these native APIs can complicate an applications deployment, since it may require root privilege for some advanced features. In academia, researchers are also interested in using the audio resources provided by Android based devices for various applications with timing constraints. For instance, indoor localization and sensing applications [16, 19, 20, 22, 28] require the emission of high-frequency acoustic signals to achieve acceptable accuracy; Surround sound play [10] requires less than 1 ms synchronization. This work provides a mechanism to realize such applications with time sensitive audio requirements on the Android platform.

SounDroid [9] outlined two main challenges of real-time audio management in Android: 1) tight timing requirement of audio requests; 2) unpredictable dispatching latency for audio playback and record. It addressed these two problems via SounDroid framework APIs, which provides customized earliest deadline first scheduling and acoustic frequency division scheduling for audio requests, and device buffer padding for unpredictable device latency. While SounDroid makes significant contributions in terms of scheduling requirements for Real-time sound, it does not directly address the inherent latency and unpredictability of the system. Our work aims to address this issue, and also provides a lower latency implementation of the sound system with session management and a declarative manifest which application developers can leverage to use our proposed sound system.

Previous attempts to make Android amenable to real-time include the work of Maia *et al.* who proposed different system architectures for Android with real-time usage [12, 13, 15, 17, 21]. Kalkov *et al.* [7, 8] proposed to extend the garbage collection of Android's runtime that avoid GC pauses during the execution of critical tasks, and explored how components interact through `Intent` messages, and re-designed it to provide priority awareness. RTDroid [24, 25, 26] utilized an existing real-time OS kernel and an off-the-shelf real-time JVM, and redesigned the core components in its framework layer with real-time features, including `Looper` and `Handler`, `Alarm Manager`, and `Sensor Manager`. These works while providing a step forward in making Android suitable for real-time applications, do not reason about the requirements of the audio subsystem. This work enables the audio ability in RTDroid with bounded audio processing latency.

7. CONCLUSION

In this paper we discuss the shortcomings of the Android audio framework and its unsuitability for real-time audio applications. We introduce the RTDroid audio architecture for managing sound playback. RTDroid leverages a declarative mechanism for describing audio sessions in the real-time manifest. We evaluate our system using a series of micro-benchmarks as well as two real-time applications: a surround sound application and a wind farm monitoring application. Our results show that unlike Android our audio manager provides predictability and timing guarantees that are suitable for use in real-time audio applications.

References

[1] Garage Band: Music creation Studio for Apple. `http://www.apple.com/mac/garageband/`.

[2] Group Play: Shared media content playback on Android devices. `http://developer.samsung.com/group-play`.

[3] OpenSL SE for Android. `https://developer.android.com/ndk/guides/audio/opensl-for-android.html`.

[4] Hamza Ali, Arthur P Lobo, and Philipos 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.

[5] J. Fillwalk. Chromachord: A virtual musical instrument. In *3D User Interfaces (3DUI), 2015 IEEE Symposium on*, pages 201–202, March 2015.

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

[7] 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.

[8] Igor Kalkov, Alexandru Gurghian, and Stefan 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.

[9] 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.

[10] Hyosu Kim, SangJeong Lee, Jung-Woo Choi, Hwidong Bae, Jiyeon Lee, Junehwa Song, and Insik Shin. Mobile Maestro: Enabling Immersive Multi-speaker Audio Applications on Commodity Mobile Devices. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '14, pages 277–288, New York, NY, USA, 2014. ACM.

[11] Kaikai Liu, Xinxin Liu, and Xiaolin Li. Guoguo: Enabling Fine-grained Indoor Localization via Smartphone. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 235–248, New York, NY, USA, 2013. ACM.

[12] 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.

[13] Wolfgang Mauerer, Gernot Hillier, Jan Sawallisch, Stefan Hönick, and Simon Oberthür. Real-time Android: Deterministic Ease of Use. In *Proceedings of Embedded Linux Conference Europe, ELCE*, volume 12, 2012.

[14] Noah J Myrent, Douglas E Adams, Gustavo Rodriguez-Rivera, Denis A Ulybyshev, Tomas Kalibera, Jan Vitek, and Ethan Blanton. A Robust Algorithm for Detecting Wind Turbine Blade Health Using Vibro-Acoustic Modulation and Sideband Spectral Analysis.

[15] 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.

[16] Chunyi Peng, Guobin Shen, Yongguang Zhang, Yanlin Li, and Kun Tan. Beepbeep: A high accuracy acoustic ranging system using cots mobile devices. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 1–14, New York, NY, USA, 2007. ACM.

[17] Luc Perneel, Hasan Fayyad-Kazan, and Martin 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.

[18] 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.

[19] Jian Qiu, David Chu, Xiangying Meng, and Thomas Moscibroda. On the feasibility of real-time phone-to-phone 3d localization. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 190–203, New York, NY, USA, 2011. ACM.

[20] 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.

[21] Ganesh Jairam Rajgurn. Reliable real-time applications on Android OS. *International Journal of Management, IT and Engineering*, 4(6):192–201, 2014.

[22] Stephen P. Tarzia, Peter A. Dinda, Robert P. Dick, and Gokhan Memik. Indoor localization without infrastructure using the acoustic background spectrum. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 155–168, New York, NY, USA, 2011. ACM.

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

[24] Yin Yan, Shaun Cosgrove, Ethan Blanton, Steven Y. Ko, and Lukasz 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.

[25] Yin Yan, Sree Harsha Konduri, Amit Kulkarni, Varun Anand, Steve 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, New York, NY, USA, 2013. ACM.

[26] Yin Yan, Sree Harsha Konduri, Amit Kulkarni, Varun Anand, Steve Ko, and Lukasz 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.

[27] Li Zhang, Parth H. Pathak, Muchen Wu, Yixin Zhao, and Prasant Mohapatra. Accelword: Energy efficient hotword detection through accelerometer. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 301–315, New York, NY, USA, 2015. ACM.

[28] Zengbin Zhang, David Chu, Xiaomeng Chen, and Thomas Moscibroda. SwordFight: Enabling a New Class of Phone-to-phone Action Games on Commodity Phones. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 1–14, New York, NY, USA, 2012. ACM.