Can Android Run on Time? Extending and Measuring the Android Platform's Timeliness

YIN YAN, GIRISH GOKUL, KARTHIK DANTU, STEVEN Y. KO, and LUKASZ ZIAREK,
University at Buffalo, The State University of New York
JAN VITEK, Northeastern University and Czech Technical University

Time predictability is difficult to achieve in the complex, layered execution environments that are common in modern embedded devices such as smartphones. We explore adopting the Android programming model for a range of embedded applications that extends beyond mobile devices, under the constraint that changes to widely used libraries should be minimized. The challenges we explore include the interplay between real-time activities and the rest of the system, how to express the timeliness requirements of components, and how well those requirements can be met on stock embedded platforms. We detail the design and implementation of our modifications to the Android framework along with a real-time VM and OS, and we provide experimental data validating feasibility over five applications.

CCS Concepts: • **Computer systems organization → Real-time systems**; • **Software and its engineering → Real-time systems software**; **Specialized application languages**; **Specification languages**; • **Human-centered computing** → *Ubiquitous and mobile computing*;

Additional Key Words and Phrases: Real-time, Android, mobile devices

ACM Reference format:
Yin Yan, Girish Gokul, Karthik Dantu, Steven Y. Ko, Lukasz Ziarek, and Jan Vitek. 2019. Can Android Run on Time? *ACM Trans. Embed. Comput. Syst.* 17, 6, Article 97 (January 2019), 26 pages.
https://doi.org/10.1145/3289257

1 INTRODUCTION

Embedded devices are being used in contexts that require increasingly complex software stacks that often combine real-time components with timing-oblivious software elements. This is certainly the case with smartphones and tablets, but it also holds for the Internet of Things and other edge devices. While many applications in these fields have some timeliness requirements, they are typically not written using best practices for real-time systems. More often than not, developers forsake predictability in favor of ease of programming. Thus one may see applications written in dynamic languages such as Java or Python running on stock operating systems.

This article investigates the viability of using the Android programming model to write software systems that mix real-time and non–real-time components. In particular, we are interested in minimizing the changes to the Android programming model and to its libraries. While the choice

Authors' addresses: Y. Yan, G. Gokul, K. Dantu, S. Y. Ko, L. Ziarek, and J. Vitek, 338E Davis Hall, Buffalo, NY, 142600-2500, USA; emails: {yinyan, g8, kdantu, stevko, lziarek}@buffalo.edu, j.vitek@northeastern.edu.

1539-9087/2019/01-ART97 $15.00
https://doi.org/10.1145/3289257

of Android is motivated by its popularity on mobile devices, our goal is to broaden its applicability to a larger class of embedded applications. Our approach is pragmatic—we have identified a small number of target applications and extend the platform to support those well. These applications include personalized healthcare (e.g., a cochlear implant) [2], audio-based indoor localization [15], harmonized sound reproduction [13], timely sound delivery [14], and UAV flight control [3]. We have also deployed our system on a wind turbine for blade health monitoring [18].

Android has a programming model based on the Java programming language, with libraries designed and optimized for mobile devices. The system presented here is a natural extension of our first-generation RTDroid system [31, 33] that focused on integrating the Fiji VM virtual machine and Android system services with support for the Real-Time Specification for Java (RTSJ) [7] on top of a real-time operating system.

To retain a familiar style of development, we make only a small number of changes to the Android abstractions and how they interact with the underlying system as well as with each other. We aim to leave legacy Android code unaffected and expose real-time features to components which have timeliness requirements. The changes to Android fall in three categories:

- *Components:* We introduce real-time services, tasks, and receivers to represent timing-aware software elements. The timeliness and resource requirements of these components are defined declaratively in a manifest.
- *Communication:* We extend the platform's communication primitives to provide control over how components of different priority communicate and mechanisms to interact with legacy code.
- *Memory Management:* We expose a limited form of region-based memory that allows programmers to circumvent the garbage collector and to ensure isolated regions of memory for each component.

While our experience using Android for embedded tasks has been mostly positive, we should mention some limitations of this work. Performance has not been a problem for our target applications, but clearly using a high-level language can come at a cost in throughput for some compute-bound tasks. In addition, we have not optimized our system for small devices; there are Java-based virtual machines for tiny devices, but this usually comes with further degradation in performance. Last, unlike the RTSJ, which was designed with great care to cover many different real-time programming styles, our design was driven by the use-cases at hand and we make no claims of generality.

This article extends our previous work [6, 32] and presents a comprehensive overview of the RTDroid programming model. The main additions are the implementation of a real-time sound processing framework (Section 4), two additional applications (acoustic ranging and augmented reality), the reproduction of results from [6] (surround sound), and two new microbenchmarks measuring sound latency estimation and acoustic ranging performance.

The article is organized as follows. Section 2 introduces a smartphone-powered cochlear implant application to motivate the design of our programming model. Section 3 details the programming model and its implementation. Section 4 provides a case study highlighting the integration of the real-time audio framework of [6] into this article's programming model. We discuss two applications and their implementation. Section 5 evaluates RTDroid. To validate our design and evaluate the predictability of our implementation, we conduct stress-test micro-benchmarks. We also report on a series of applications developed with RTDroid. Our results illustrate that, at least in these use-cases, the modified platform delivers significantly better time predictability than stock Android and reduces code complexity as compared to the RTSJ.

```
1  class ConfigurationUI extends Activity{
2    ClickListener l = new ClickListener() {
3      public void onClick(View v) {
4        //change processing config
5    } };
6    public void onStart(){
7      button.setOnClickListener(l);
8   } ...
9  }
```

Fig. 1. Audio configuration UI written in Android.

```
1  class ProcessingService extends Service {
2    public void onStartCommand() {
3      /* periodic audio processing */
4      while (true) {
5        //process every 8 ms
6      }
7    }
8    ...
9  }
```

Fig. 2. Audio processing service written in Android.

2 AN ANDROID-ENABLED REAL-TIME APPLICATION

Using Android for real-time computing is challenging for several reasons. Android provides four software architectural elements, *services*, *activities*, *broadcast receivers,* and *content providers* for, respectively, background computation, foreground computation with user input, handling system-wide events, and data storage. The Android framework that uses the Linux scheduler is not priority-aware, and there is no mechanism to assign priorities to threads. All of these components are executed with Android's event-driven model, which offers two primary event types: *messages* and *intents*. Messages are received by a `Handler`, which is a unique mailbox for all messages directed to a component. An `Intent` is an event that triggers execution of callbacks in components that have registered for it. Application priority can be set with `Thread.setPriority()` and computations can be assigned to the background, but these priorities are only effective in the current thread. Callbacks and events posted to other components do not inherit their sender's priority. There is no notion of priority for messages, and the FIFO queue associated with each handler can lead to arbitrary latency in processing. The underlying scheduling mechanism does not run a priority inheritance protocol to avoid priority inversion. Memory pressure is also a concern. Android provides no mechanism other than garbage collection to manage memory, and its garbage collector does not have real-time guarantees. To makes matters worse, there is no way to bound memory consumed by different components. Thus, a stray noncritical component can affect the whole system.

Even with these limitations, the healthcare industry has been studying how to adapt Android for wearable and implantable health devices, like *cochlear implants*. A cochlear implant restores hearing abilities through an electronic device surgically inserted into the inner ear. It relies on external components to capture ambient audio, convert it into digital signals, and translate the signals into electrical energy. There is interest in leveraging smartphones [2] to provide additional services such as on-the-fly translation or noise cancellation. In such a scenario, a smartphone records audio streams and processes them. To provide acceptable performance sound samples must be handled at rate of one every 8ms.

A plausible design for such an application would be to split the user interface that controls volume and noise reduction from sound processing. The UI can be implemented as an activity, as shown in Figure 1. That activity could deal with configuration parameters set by the user. On the other hand, sound processing is best modeled as a service (Figure 2), which repeatedly processes sound samples. Even in such a simple use-case, it is important to ensure that sound processing is not delayed by UI processing. When components have to interact through Android-based communication mechanisms, ensuring noninterference can become quite tricky.

Figure 3 shows the architecture of our solution in RTDroid. It separates real-time (`RecordingService`, `ProcessingService`, and `OutputReceiver`) and non–real-time components (`VolumeReceiver` and `ConfigurationUI`). The former have priorities attached and use communication services that prioritize messages. `ConfigurationUI` has a `Handler` for other components

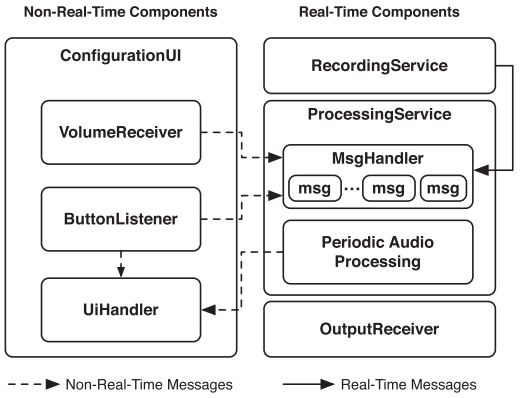


Fig. 3. Architecture of cochlear implant application.

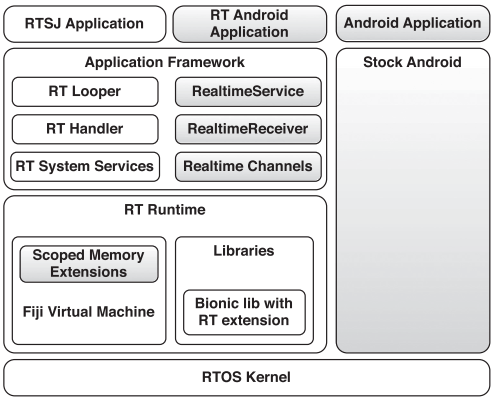


Fig. 4. RTDroid runtime architecture.

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

to update the UI, and a non–real-time receiver listens on volume key events. It also receives messages from real-time components. Similarly, the `ProcessingService` receives messages from non–real-time components (`VolumeReceiver` and `ButtonListener`) and a real-tine component (`RecordingService`). RTDroid allows these components to communicate while enforcing memory bounds. Each real-time component is provided a fixed amount of memory for its exclusive use. That memory is divided into two sections: one persists for the lifetime of the component, the other is cleared each time the component yields control. Both memory sections are implemented as RTSJ scoped memory areas supported in Fiji VM [23] and managed in the RTDroid framework. Messages are preallocated. Non–real-time components allocate messages in heap memory. RTDroid extends the Android manifest to enable developers to declare properties of components that include priority, periodicity, and memory bounds.

3 REAL-TIME ANDROID

We now review the design and implementation of RTDroid, our real-time aware version of Android. RTDroid is available in open source at https://rtdroid.cse.buffalo.edu. Our first release [33] integrated a real-time Java virtual machine and a real-time operating system with the Android framework, as well as redesigned some Android system services to support real-time tasks. That release relied on real-time garbage collection and was limited to running a single real-time application. The current version introduces high-level, real-time constructs with concrete memory bounds (*RealtimeService* and *RealtimeReceiver*), low-level constructs for communication (*Realtime Channels*), and a mechanism for specifying the real-time properties of the constructs shown in Figure 4. These new constructs allow for programming real-time applications in an Android-like manner and for communication between applications. We also support running stock Android applications in a separate virtual machine.

It is important to realize that, while RTDroid supports dynamic loading of code, our system has a dedicated bootstrap sequence divided into two stages: compile-time and application run-time, shown in Figure 5. The two-stage process ensures that memory can be preallocated and components are correctly configured. At compile-time, our framework parses the manifest file of an application, runs verification checks, and emits configuration bytecode for all components.

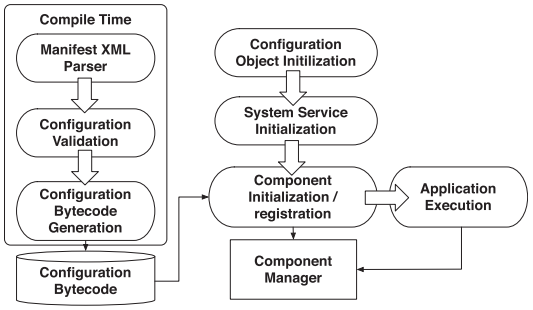


Fig. 5. RTDroid bootstrap sequence.

Verification checks include validating that the specified set of components is schedulable [34], ensuring that the memory necessary for each component is available, and checking the inference of channel sizes based on specified communication patterns and scheduling parameters of communicating components. The compile-time validation is performed for a single app. This configuration bytecode provides a unique handler for each application component. At boot-time, the system goes through the list of handlers and calls each handler to instantiate its corresponding application component. After instantiation, a handler registers its component with our component manager. This component manager manages the lifetime of each component.

3.1 Components

RTDroid supports three different real-time components: services, tasks, and receivers. A `RealtimeService` is a counterpart to an Android service and is used for one-shot aperiodic or sporadic computation. As the notion of periodic computation (a computation that runs at fixed and predictable intervals) is foreign to Android, we introduce the `PeriodicTask` class to model such behavior. Tasks are used internally within a real-time service. A `RealtimeReceiver` is used to react to system-wide events delivered via intents. We do not provide a counterpart to Android's *activities* and *content providers*. They are used for UI programming and data persistence, and we have not observed real-time requirements for those in the applications we designed. However, we do allow for interaction between UI components and real-time components through message channels.

Real-time components are statically assigned the following: a priority, a starting time, a deadline, and a memory limit. This is done declaratively by extending Android's manifest with properties (`priority`, `memSizes`, `release`). The association between a periodic task and its parent is also specified in the manifest by a `periodic-task` tag. The manifest provides information for boot-time verification and preallocation of components. RTDroid ensures that the total memory requested specified for a component equals the objects in its persistent memory, its per-release memory, and that of its subcomponents. Figure 6 shows a manifest for the processing service of our running example.

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

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

Fig. 6. An extended Android manifest.

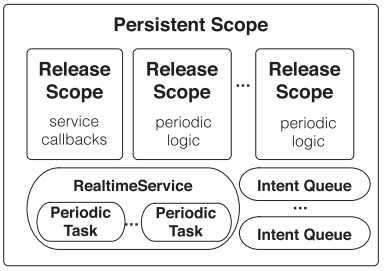


Fig. 7. Scope structure for a service.

The concept of region-based memory allocation comes from the Real-time Specification for Java (RTSJ) [7]. The idea is to avoid having to manage individual objects; instead, objects are allocated to regions, which can be deallocated in one fell swoop. Furthermore, regions can be nested, thus creating a stack of scopes within bounded lifetimes. The RTSJ introduced this idea to Java to provide an alternative to garbage collection. In the RTSJ, each thread was associated with a particular scope, and scopes were nested to form a cactus stack. For RTDroid, region-based allocations has two important benefits: threads that are using it need not be paused during garbage collection, and they make it possible to bound the amount of memory allocated by any thread. RTDroid supports a much simpler form of scoped memory than the RTSJ. Each component has access to two scopes: one is *Persistent Memory* for data that live as long as the component and the other is *Release Memory,* which is cleared before each release of a periodic task. The size of these scopes is given in the manifest. The total memory of a component is the sum of its persistent memory, release memory, and the memory of internal components.

3.1.1 Service. A real-time service is an abstract class; a programmer needs to implement its callbacks. These callbacks are directly inherited from Android's service, and they are invoked at different points in the lifetime of a service. The `onCreate()` callback is invoked at service creation. The `onStartCommand()` method is called at startup and usually implements application logic. Figure 8 shows a service that starts a periodic task. Unlike Android services which run in the main thread, RTDroid services execute in dedicated threads. This change is necessary to allow services to run with different priorities.

Services are bound to real-time threads from the underlying real-time virtual machine. By default, when a service is initialized, it is assigned a persistent memory scope that has the same lifetime as that service. The scope is allocated when the service starts and deallocated when the service terminates. Static initializers for the service are run in this scope; thus, all persistent data for the service will be allocated in that scope and deallocated when the scope is freed. In addition, if the service uses communication channels, intent queues are allocated in persistent memory. Callbacks execute within the scope of release memory for the associated service. The release memory is cleared when the callback returns. Similarly, when a periodic task is started in a service, it is also assigned a release scope. Note that our manifest requires specification of memory bounds for callbacks and periodic tasks, and this information is used to size release scopes appropriately. Figure 7 depicts the scope structure for a service consisting of several tasks, as well as the preallocated objects during boot.

3.1.2 Periodic Task. A periodic task is a subcomponent of a service. In addition to the characteristics of its parent service, a task needs a period and a start time, both of which are specified in the manifest. The period determines when the tasks will be executed by the RTDroid scheduler.

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

Fig. 8. Real-time service and periodic task.

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

Fig. 9. Real-time channel declaration.

Figure 8 shows an example which processes audio input periodically. The `onRelease()` callback is implemented by the developer and contains the application logic that is run at each period.

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

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

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

3.2 Communication

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

Figure 9 shows a real-time message-passing channel declaration with a name attribute as an event identifier. Each channel should define its runtime behavior via `type` attribute (channel

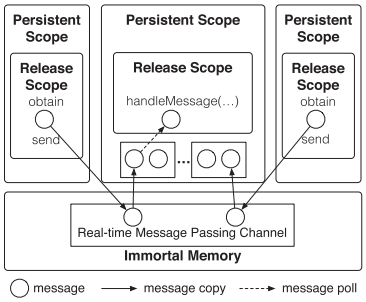


Fig. 10. Real-time message passing channel.

```
1  MessageClosure c =
2    new MessageClosure(){
3      @Override
4      public RTMSG genMsg(RTMSG m){
5          Bundle b = m.getData();
6          b.setInt(idx, 3);
7          ...
8          return m;
9      }
10 };
11
12 rmsg.send("channel", c);
```

Fig. 11. Message passing interface.

communication type), `order` (message delivery order), `execution` (execution priority of the invoked function), `drop` (message dropping policy), `data size`, and `data type`. Components can use `intent-filter` to identify themselves as *publishers* or *subscribers* of a channel and to specify the number of messages sent or read in each callback release.

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

3.2.1 Message Channels. A real-time message-passing channel allows components to communicate with statically bounded memory usage while enabling dynamic message passing with best effort at run-time. It has three distinctive characteristics: (i) there is only one receiving component per channel, and each `RealtimeHandler` must be registered in its associated real-time service; (ii) only primitive arrays (or fixed-length byte-buffers) can be exchanged; and (iii) the number of in-flight messages is bounded.

To enforce the memory usage bounds of components' communication, the implementation creates a fixed-length message pool for each channel. Message objects are preallocated in the persistent memory of the channel's receiver. Figure 10 illustrates the scope memory hierarchy. When a message is sent, the sender creates an instance of `MessageClosure`. As shown in Figure 11, the message closure is allocated in the sender's release scope. When the method `send()` is called, it triggers the actual enqueuing operation and transfers the control of message enqueuing to the receiver. The receiver copies the message from the sender's allocation context to its message pool. This ensures that a sender cannot utilize or fill the allocation context of a receiver directly. The receiver must choose to receive the message. Since each channel is itself bounded, non–real-time senders cannot overflow a channel. A message will only be copied to the receiver when that receiver is ready to process the message. After the message is processed, the message object is returned to the message object pool.

The memory dedicated to messages is constant in the message-passing channel. The sender utilizes its own memory (heap or its release scope) to create the data that it wishes to send and cannot use the receiver's resources to store these data unless it is able to obtain a message object. Queuing of messages is handled based on the sender's priority. If the pool is empty, high-priority components can steal a message from a low-priority sender (i.e., the receiver can choose to drop a low-priority message to satisfy an enqueuing request from a higher priority sender). In such cases, the system notifies the low-priority sender of the message being

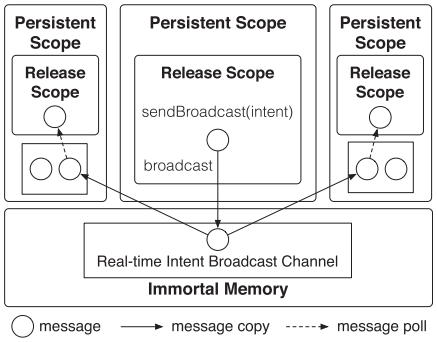


Fig. 12. Real-time intent broadcast channel.

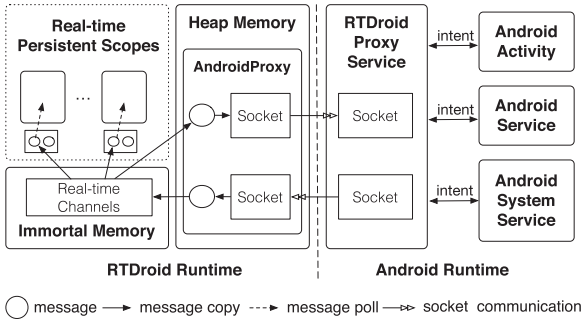


Fig. 13. Cross-context channel.

dropped by delivering an `AsynchronousInterruptedException`. This exception, defined in RTSJ, allows notification across threads. RTDroid's components check and handle associated `AsynchronousInterruptedExceptions` before and after each invocation of the execution of component callbacks.

3.2.2 Broadcast Channels. Real-time broadcast channels are used to invoke callbacks of real-time components. We decouple the priority of intent delivery from invocation of callbacks which execute at their own priority. Intents are, by default, delivered in priority order in the same way as messages over a message channel. The main difference between intents and messages is the number of recipients. For messages, this is always one, while for intents this is the number of subscribers. Subscription to an intent must be declared in the component's manifest. Figure 12 shows how an intent object persists in immortal memory until it is copied to the intent queues in multiple subscribers. Although the message will be replicated for each subscriber, only one message is stored in the channel itself. A count is associated with the message identifying the number of recipients subscribed to the intent. On receipt, when the message is copied to the receivers intent queue, the count is decremented. The last recipient releases the message back to the message pool in exactly the same fashion as the message-passing channel. The memory usage of the broadcast channel is bounded because intent objects are preallocated in each subscriber's intent queue based on the size and type of data in the manifest as well as a bounded number of intent messages.

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

3.2.4 Cross-Context Channels. Cross-context channels allow Android's activities to communicate with real-time components. In this scenario, communication is occurring between two separate VMs, one of which is executing the non–real-time application while RTDroid executing a real-time application. This allows us to support interaction with both legacy Android code as well

as other Android applications. We note that cross-context channels are not required for communication between multiple real-time applications as the Fiji VM supports multiple VMs in the same address space.

To enable such communication an Android application must declare a service (`RTsProxyService`) that subscribes to channels declared in a real-time application that uses our real-time constructs. For communication in the other direction, a real-time application need only to subscribe to intents that the non–real-time application has declared in its manifest. Since our manifest is an extension of the Android manifest, no changes are required to the configuration of Android. The proxy service allows non–real-time code to send an intent to real-time components. Communication in the other direction requires that the activity can subscribe to intents defined by real-time code. To preserve memory bounds, the number of intents in a cross-context channel is bounded, and each intent has a fixed-length payload. Figure 13 shows how the bidirectional communication is established through sockets between RTDroid and Android. To do so, we leverage two proxy components in each runtime, To avoid interference, the Android proxy component is executed in heap memory, and it runs at the lowest real-time priority. The incoming message objects are translated to real-time intents or messages with the lowest priority and sent to the subscribing real-time components via real-time channels. Only one message is deposited into a real-time channel at a time, thus preventing non–real-time components from exhausting memory used by real-time constructs. Non–real-time components can exhaust the heap, but this will not affect real-time components using preallocated memory regions.

3.3 Memory Management

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

To achieve predictable object allocation and reclamation, we leverage scoped memory to provide memory bounds corresponding to the lifetime of different computations as well as data across computations. To provide a memory boundary for each component, we group the computation and associated allocations performed by the computation into two separate lifetimes: (i) the duration of the *lifetime* of the component (persistent scope) and (ii) the duration of one callback invocation (release scope). The scopes correspond directly to the types of memory defined by our system: persistent memory and release memory, respectively.

The complexity of using scoped memory is hidden in RTDroid's framework. From the programming perspective, a developer only needs to specify memory sizes in RTDroid's application manifest for each real-time component as well as the number of messaging objects for the application communication. For a given component, the component instance and its class variables are

allocated in its persistent memory; any other local variables are stored in the release memory. To support the need of "persisting" objects, there is an extra API call that allows allocating objects from the release memory to the persistent memory in the same component.

To enforce memory guarantees for real-time components, the immortal memory is used as a substrate to allocate and keep track of references to various memories of real-time components and broadcast channels. The size of immortal memory is currently statically set in the RTDroid manifest. This size can be calculated at compile time. Each component run is bound to its own thread of control that starts in the *immortal memory*. This assures that the memory necessary for creating the execution context for the thread is always available, even if the construct has to be terminated and restarted.

4 CASE STUDY: A REAL-TIME AUDIO FRAMEWORK

This section introduces a real-time audio service as a case study to show how we redesigned and implemented real-time counterparts of Android's built-in audio framework in RTDroid. Before discussing the design of the new real-time audio service, we start with an overview of the existing audio framework and explain the difficulties of using that framework in the context of a real-time application. The remainder of this section illustrates real-time components in the real-time audio framework and discusses how we overcame the challenges.

4.1 Android Audio Framework

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

4.1.1 Application APIs. The `MediaPlayer` and `MediaRecorder` are the primary interfaces that allow developers to play back and record multimedia streams, like video and audio, from a file or a specified URI. They present higher level abstractions of an audio stream and only accept compressed audio streams that are ready for playback. In contrast, `AudioTrack` or `AudioRecord` are lower level abstraction that have adjustable data frames associated with raw data buffers. Figure 16 shows an application using `AudioTrack` to play audio. Each instance of `AudioTrack` must specify its buffer size on initialization. This size determines the minimum frequency at which to write to the `AudioTrack` to avoid under-runs, which are undesirable as they lead to playback glitches and stutter. As the data are buffered in the framework, the buffer size also affects playback latency. Similar to `AudioTrack`, the `AudioRecord` can read the data from a PCM buffer. Notice that multiple application may create multiple instances of `AudioTrack` or `AudioRecord` at the same time, so the design of the framework should be able to handle device multiplexing.

4.1.2 System Services. As in most sound frameworks, Android provides a sound daemon to handle multiclient requests. This design does introduce unpredictability, and that unpredictability can result in higher latency on the input and output paths due to the best-effort nature of the platform. As shown in Figure 14, there are many components involved in performing audio playback—this creates opportunities for communication and buffering delays. We examine each component and explain sources of unpredictability next.

The first layer that handles audio is the application framework. Writing audio data to the `android.media` APIs discussed earlier is a *push* operation, which means that data written are pushed down by the audio framework and buffered in the native layer. The framework is push-based for playback; the audio data are buffered in the native layer rather than pulled by the audio driver when it is ready to play more data. This design decision can result in significant latency. The Android framework in general does not provide any guarantees for predictability [33].

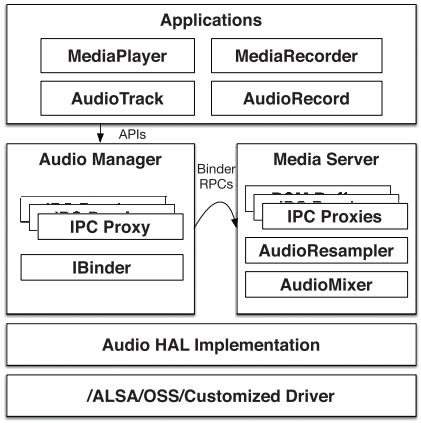


Fig. 14. Architecture of the default audio framework.

Push operations can experience latency variations, especially when many threads compete for computational and memory resources.

The next layer includes a centralized `AudioManager` that is used to configure the sound device and a native `Media Server` which implements the most important functionalities:

- It receives audio data from various applications.
- It buffers audio data if an application writes a buffer bigger than the native buffer on devices.
- It resamples on the recording buffer if the application data sampling rate differs from the sample rate of the native audio chip.
- It mixes the audio data from different processes.
- It writes the mixed and resampled data to the audio driver.

All these operations introduce unpredictable latency into the audio path. Typically, they require interprocess communication between applications and system services via Android's Binder calls and add multiple periods to the latency depending on the buffer size.

The next layer that handles audio is Android's Audio Hardware abstraction layer that connects the generic framework APIs to device-specific audio drivers and the underlying 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 be predictable in terms of latency, but Android does not have any mechanism to guarantee that this is the case.

The last layer is the popular Advanced Linux Sound Architecture (ALSA). 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, 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 worker thread but increases latency as applications must wait longer to read/write audio data. If the system is configured with a shorter period, the kworker thread is frequently invoked. Any variation due to scheduling will result in poor audio quality. The vanilla kernel used in Android does not provide any predictability guarantees in the scheduling of worker threads.

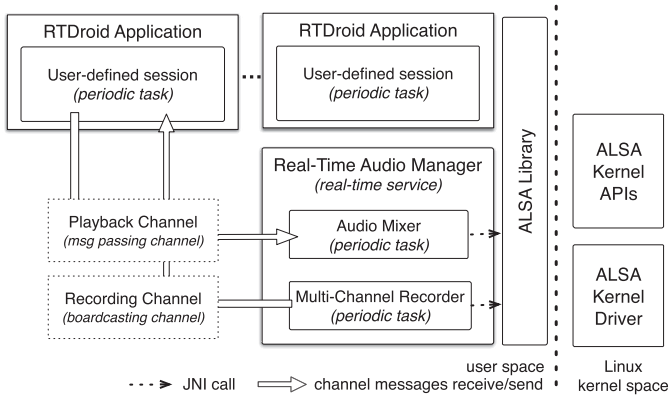


Fig. 15. Architecture of real-time audio framework.

4.2 Real-Time Audio Framework in RTDroid

Overview: To support multiplexing, RTDroid inherits Android's centralized design. For better predictability, it provides a real-time audio manager using the real-time constructs introduced in RTDroid. The new framework introduces the notion of an `audio session` that can be defined by developers. A session is similar to Android's `AudioTrack` and `AudioRecord`. However, instead of manually configuring them at run-time, the new framework requires that developers specify sessions statically in what we call the *audio manifest* so that the booting process can validate the compatibility of sessions and configure the real-time audio manager with precise timing guarantees. As mentioned in Section 3, our prior work discusses manifest validation in detail [34]; briefly, manifest validation checks whether or not the requirements of an application can be met at run-time (e.g., the schedulability, the amount of memory necessary, channel sizes, etc.).

The *audio manifest* allows programmers to express the requirements of their audio sessions *statically* in an XML file. Static audio session properties such as channel, sample rate, periodicity, source (e.g., MIC), sink (e.g., speaker), and the like, are all expressed in an audio manifest. At run-time, our audio manager guarantees that all the requirements expressed in an audio manifest are met. Figure 17 shows an example manifest, which we describe in more detail later in this section.

Figure 15 shows an overview of the real-time audio framework, where the real-time audio manager is implemented as *a real-time service*. It provides APIs to application components for audio recording and playback and interacts with the native `tinyALSA` library to control the on-device sound card. Software mixing and demultiplexing is performed within the audio manager. Our audio framework introduces the notion of a user-defined audio session. Each audio session specifies its sampling rate and local buffer size as part of its component declaration in its application manifest. Due to the limited range of audio capabilities on a modern smartphone, our audio framework provides an abstraction, *a user-defined audio session*, with a frame size and a buffer size. During the application bootstrap, the RTDroid framework checks device capabilities and subsequently allocates the user-defined session.

User-Defined Audio Sessions: The user-defined audio session is a subclass of `PeriodicTask` class, and a developer can implement the acoustic computation within the `onRelease()` function. To create an audio session, the developer must declare a periodic task with an `<audio-session>` sub-element as listed in Figure 17. For example, when an MP3 playback session is created, the type of `<audio-session>` is specified as *audio-tracker* and the sampling rate and buffer size are given. Then, the framework initializes an instance of `AudioTracker` and attaches it to the

```
/* Initiate audio track object with
 * sampling rate and buf size.*/
AudioTrack track =
          new AudioTrack(...);
track.play();
while(isPlaying){
  //play audio data
  track.write(buf, 0, bufSize);
}
track.stop();
track.release();
```

Fig. 16. Audio playback code in Android.

```
<periodic-task
     name="pkgname.Audio">
  <priority priority="51"/>
  <memSizes release="1M"/>
  <release start="0ms"
          periodic="50ms" />
  <!-- declares a playback session -->
  <audio-session type="audio-tracker"
    frame-size="44100Hz"
    buffer-size="128kb" />
</periodic-task>
```

Fig. 17. Audio session declaration in RTDroid.

corresponding session. Audio functionalities are accessed as shown in Figure 16. Rather than manually creating objects, object allocation is hidden from developers. For recording, the session type is set to `audio-recorder`.

When multiple sessions are defined, the framework performs a compatibility check for the sound device configuration at compile-time. During bootstrap, protected objects are instantiated in each session as audio I/O APIs. An instance of `AudioRecorder` is provided for receiving data from each input device and an instance of `AudioTracker` is provided for sending data to output devices.

Real-Time Audio Manager: The real-time audio manager serves as a sound daemon and provides audio functionalities to application components via the `Audio Mixer` and the `Multi-Channel Recorder`. The audio manager is implemented as a real-time service, and the `Audio Mixer` and the `Multi-Channel Recorder` are implemented as two periodic tasks for audio mixing and recording for multiple clients. They communicate with application components via real-time communication channels. The basic operation of the audio mixer/recorder is writing/reading audio frames to/from the PCM card, respectively. If there are no applications writing audio data, the playback thread pads the PCM card with silent 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.

Mixing and resampling is implemented in two periodic tasks configured during application compilation. There are two parameters: the periodicity of the task and a bytes buffer for up-scaling and down-scaling audio samples. The periodicity of a task is simply chosen as the smallest `periodic` amount of all tracking sessions or recording sessions in an application. The buffer size in a mixing or resampling task is calculated with the sampling rate and the session buffer of each session as well.

There are two real-time communication channels declared, one for audio sample subscription and the other for sample delivery. Both channels are created in RTDroid's framework, which provides constructs such as `RealTimeHandler` and `handleMessage` for real-time communication. The `Audio Mixer` and `Multi-Channel Recorder` leverage them to transfer audio data between application components and the audio manager. The communication channel ensures data integrity and prevents message overflow during audio data transfer. A message-passing channel is used for application components to send audio data to the mixing task. A broadcast channel is used for the resampling task to deliver recorded samples to application components. Both channels only accept data bytes, and the buffer size of each audio session is used to preallocate messages for both channels.

Additionally, our audio manager performs the following audio-related tasks. First, an instance of the audio manager is spawned on application bootup. 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 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, a real-time Java VM that RTDroid uses. 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.

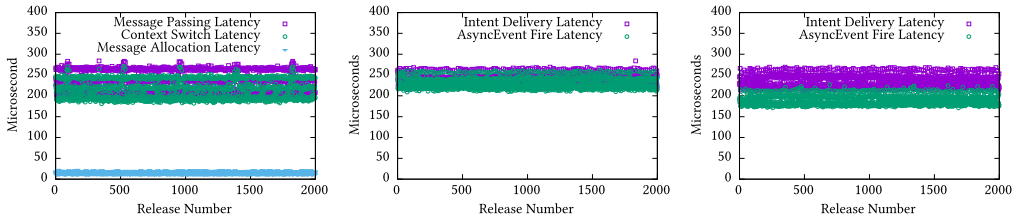
The audio manager utilizes the native audio APIs for opening, closing, reading from, and writing to the PCM card. To implement these APIs, we leveraged `tinyALSA` libraries to interface with the kernel. `tinyALSA` libraries provide APIs for directly accessing the audio PCM card and changing hardware mixer parameters which the native audio layer uses to set correct configuration parameters according to the requirements specified in the audio manifest. The library also provides the API `pcm_get_buffer_size()` to determine the lowest possible buffer size directly supported by the hardware; this is required to enable low latency as we aim to reduce data buffered between layers. In our native audio layer, we use the lowest buffer size to transfer the required amount of audio data to/from the audio manager, and `pcm_write()` and `pcm_read()` APIs provided by the `tinyALSA` library are used to write/read data from the hardware.

Use Cases: There are many use cases where real-time audio processing becomes necessary. For example, latency estimation is necessary for *coordinated playback* across multiple mobile devices as envisioned previously [13, 14]. Similarly, directional audio in museum exhibitions [4, 8] allows for a better user experience for users. For such applications, latency estimation is done by a device sending an estimation signal to another device, which returns an acknowledgment. The initial sender then measures the round-trip time from the initial signal release to response signal detection. However, uncertainties in actual sending times and receiving times caused by latency in device response could contribute to errors in round-trip time measurement. Thus, real-time predictability is necessary on both the sender's device as well as the receiver's device. Similarly, latency estimation is necessary for acoustic ranging. Knowing the speed of sound, we can estimate the distance between a sender and a receiver by sending and receiving an audio signal. Such distance measurements have been used as a primitive to design applications which perform peer-peer localization, device synchronization, and more. Ranging using sound is an attractive solution as it does not require sophisticated hardware and can be deployed on devices with a speaker and microphone. In Section 5, we evaluate RTDroid for latency estimation and acoustic ranging, and we further discuss the applications we implement to evaluate these techniques.

5 EVALUATION

This section shows experimental results for micro-benchmarks and applications in RTDroid. To evaluate the predictability and efficiency of our programming model, we conduct a series of stress tests for message delivery latency and three case studies which compare task processing duration and code efficiency between RTDroid and Android. The applications are the cochlear implant application of Section 2, an UAV flight control system, jPapaBench [9], and a turbine health monitoring application. We use these case studies to compare RTDroid with Android and the RTSJ. To explore the correctness of our audio framework, we run stress tests as well as evaluate it on applications that compute device coordination and distance estimation.

Results are collected on a Raspberry Pi Model B, which has a single-core ARMv6-based CPU with 512 MB RAM and runs Debian with Linux preemptive kernel v3.18 and a Google Nexus 5



(a) Message Passing (b) Intent Broadcast (c) Bulk Data Transfer

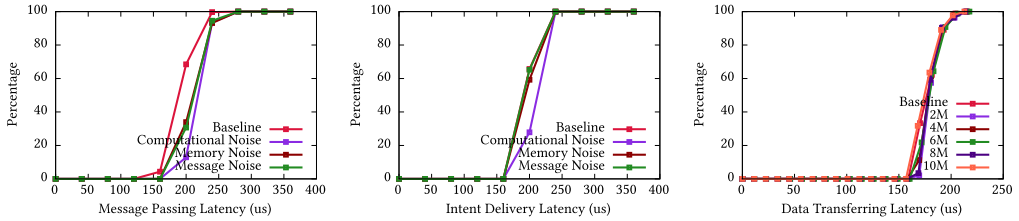
Fig. 18. Real-time communication channels: baseline scatter plot for micro-benchmarks.

smartphone, which has a quad-core 2.3GHz Krait 400 Processor and 2GB RAM, running Android v6.0.1. On both platforms we only enable one core and fix CPU frequency. For the turbine health monitoring application, we use an external Wolfson audio codec to provide high-quality audio playback and capture for vibro-acoustic analysis. Raw data and plotting scripts can be found under the publications tab and cases study code under the application tab on our website: http://rtdroid.cse.buffalo.edu.

5.1 Micro Benchmarks

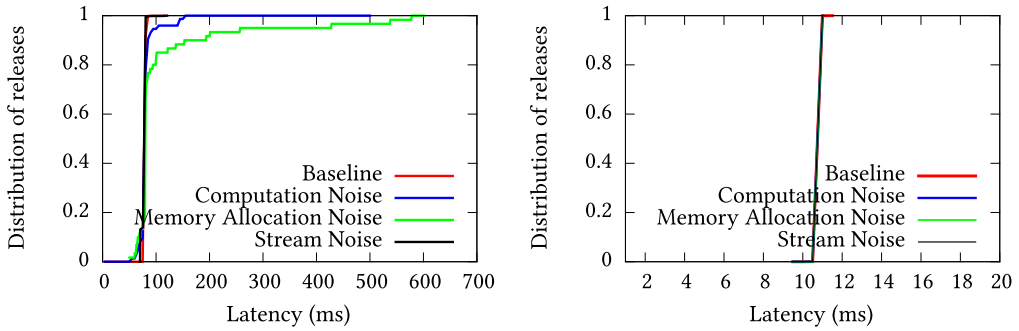
The benchmark for communication runs two real-time services and one non–real-time service. One real-time service acts as a sender that sends a message every 100 *ms* with the highest priority and one as a receiver of the message. The third service, executing in heap memory, starts 30 noise-making threads with the lowest priority to inject noise into the system. Similarly, the audio micro-benchmark has a real-time service which contains a user-defined audio session and plays a fixed number of audio data from a MP3 file. To stress the system, we use three types of noise-making threads: (i) heap noise that allocates an array of 512 KB in the heap memory every 200 *ms*, (ii) computational noise that computes π every 200 *ms*, and (iii) message noise for the message latency measurement, which sends a low-priority message to the receiving service every 200 *ms*, or stream noise for the audio playback measurement, which delivers a fixed-number of audio samples to play every 200 *ms*.

5.1.1 Message Delivery Latency in Communication Channels. Figure 18 shows the performance of our channel implementations. Message passing consists of message allocation by the sender, message delivery, and context switch from sender to receiver. Figure 18(a) shows this breakdown with just the sender and the receiver. This is the baseline performance. The figure plots the latency of 2000 message-passing events. For each event, the message allocation latency is the amount of time it takes for a sender to instantiate a message. The message-passing latency is the time taken for delivery. The context switch latency is the difference between the time the sender sends a message before the receiver processes the message. As shown, all three types of latency are tightly bounded across all events, and there is no outlier that takes much more time to process than others. It shows that, without any other background load, our implementation provides stable and predictable performance. We have conducted a similar experiment to evaluate our `Intent` delivery channel. The experimental configuration is the same except that we use our `Intent` broadcast channel instead of message passing; the sender sends an `Intent` every 100 *ms*, and the receiver executes a dummy callback that responds to the `Intent`. The `Intent` delivery latency is the overall latency for each `Intent` event, and the callback trigger latency is the amount of time it takes to spawn a new callback. Figure 18(b) shows the baseline performance. Similarly, Figure 18(c) shows the baseline performance for bulk data transfer, which also leverages the `Intent` mechanism for delivery but is specialized to use the bulk data transfer channel.



(a) CDF for Message Passing (b) CDF for Intent Broadcast (c) CDF for Bulk Data Transfer

Fig. 19. Micro-benchmarks for real-time communication channels.



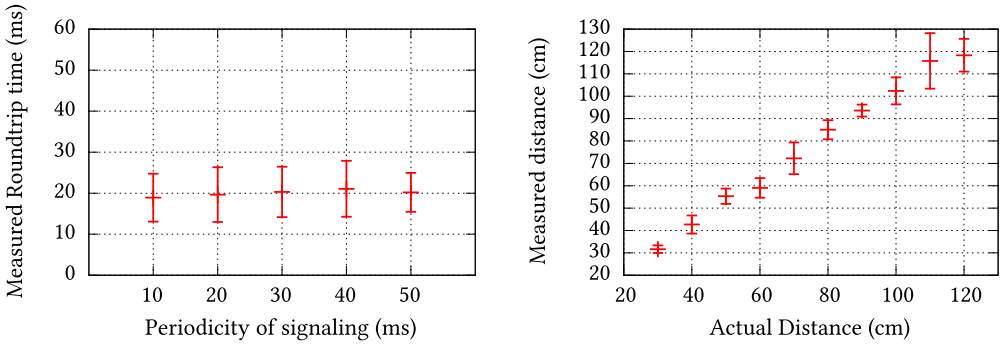
(a) CDF on Android (b) CDF on RTDroid

Fig. 20. Audio playback latency.

Figure 19 shows Cumulative Distribution Function (CDF) plots comparing the performance of all three types of channels. The CDF illustrates what percent of the total measured points is equal to or less than a given value. For basic messaging, Figure 19(a), our implementation provides a predictable latency profile regardless of the types of background load. Figure 19(b) reveals similar characteristics for broadcast channels, though there is additional overhead as compared to message passing. This is to be expected as the intent broadcast creates a callback, which adds fixed overhead. Figure 19(c) shows the CDF comparing the transfer latency with different sizes of data payload for the bulk data transfer channel. The transfer latency is the delivery time of an intent with a bulk data payload. Instead of noise-making threads, we increase the size of data payloads to demonstrate the performance of *zero-copy* data transfer.

5.1.2 Playback Latency in Real-time Audio. Figure 20 shows the CDF of audio playback latency calculated as a time variance between the time when a byte buffer is sent from the session to the audio manager and the time when the buffer is transferred to the native library and ready to deliver to the kernel. Figure 20(a) is the CDF of the latency on Android. The figure shows that memory and computation noise induce delays. In the 2000 collected latencies, we observed between 10% and 20% of samples had increased latency due to noise. Notice that stream noise is almost imperceptible since the audio manager mixes streams before sending them to the native tinyALSA library. With RTDroid, Figure 20(b) shows that performance is much better (faster) and more predictable (sharper step in the CDF and few outliers).

5.1.3 Latency Estimation. Kim et al. [13] suggest that sound spatialization is a key acoustic technique to enrich sound reproduction. Several multichannel standards have been proposed for this purpose. Key to spatial reproductions is maintaining accurate playback timing across multiple speakers. Traditionally, all speakers are wired to a central mixer that carefully controls playback



(a) Latency estimation for different periodicities of signaling. RTDroid provides consistent latency estimates irrespective of periodicity

(b) Acoustic ranging: Actual vs measured distances

Fig. 21. Latency estimation and acoustic ranging.

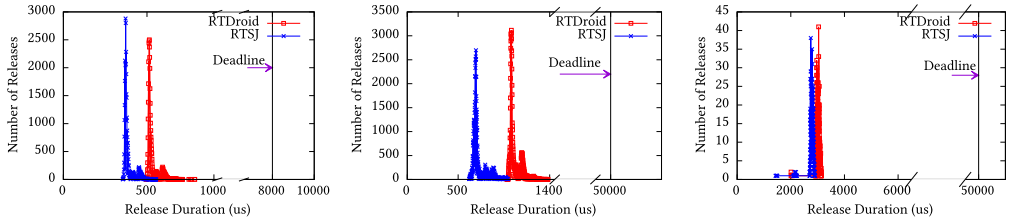
timing. However, in a mobile audio scenario, identifying latency between devices precisely will allow us to recreate the surround sound experience without the need for wiring. Latency estimation targets this suite of applications and allows estimation of latency between speakers in the background. Shown in Figure 21(a) are round-trip estimates between static devices with varying periodicity. Irrespective of periodicity, our estimator accurately estimates the latency. This latency is mostly dependent on the distance between the sender and the receiver.

5.1.4 Acoustic Ranging. Performing acoustic ranging on commodity devices has been a challenge, and previous work builds ranging/distance measurement using sound [15, 21]. The inherent latency associated with sound processing on smartphones and the unpredictability under varying system loads are the major challenges. We use two devices running our application. Ranging is initiated on one device that sends out an audio signal. On receiving this audio signal, the second device plays a response signal. When the first device hears the response, the round-trip time between the ranging and response is recorded. The distance between the two devices is estimated using this round-trip time. We performed ranging between two devices placed at different distances. We captured hundreds of measurements at each distance. Figure 21(b) shows the actual distances between the devices and the distance measured by the application, with the average and standard deviation of the estimated distance. The results show that the system is able to accurately estimate the range between two devices with a resolution of under 10 *cm*.

5.2 Comparison to Android and RTSJ

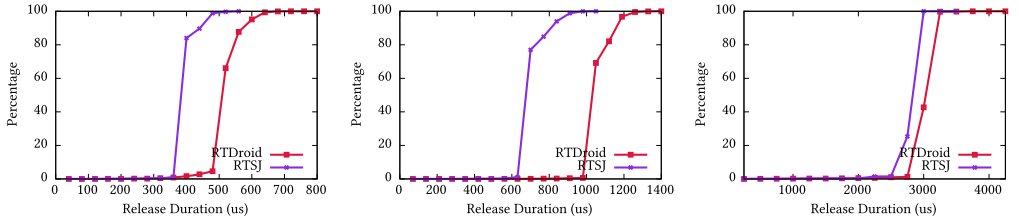
We conduct three case studies consisting of a cochlear implant application, a UAV flight control system, and a turbine health monitoring application to compare RTDroid in realistic settings against Android as well as RTSJ.

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



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

Fig. 22. Performance measurements on Raspberry Pi.



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

Fig. 23. CDFs of performance measurements on Raspberry Pi.

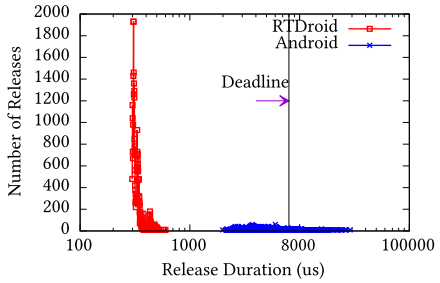
5.2.2 jPapaBench. A real-time Java benchmark simulates autonomous flight control. We have ported it to our system as well as Android and divided the code into two services: (i) an autopilot service that executes sensing, stabilization, and control tasks; and (ii) a Fly-by-Wire (FWB) service that handles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

5.2.3 Wind Turbine Health Monitor. The wind turbine health monitoring application was developed originally using RTSJ. We have also created a version to execute on our system. Since this application requires specialized hardware, we did not implement an Android version. The application performs crack detection on turbine blades based on vibro-acoustic modulation [18]. It consists of a probing task that imposes a clean sine-wave audio tone at one side of a blade, a recording task that stores the captured audio from the other end of the blade, and an analyzing task that detects cracks by analyzing the stored audio stream. The audio recording task *must* be executed every 50 *ms* in order to capture meaningful data and, as such, is our main point of measurement. We collected release durations of the audio recording task over 2 hours and only kept releases that perform recording logic. The size of the audio buffer recorded per release is around 2MB, and, as such, we leverage our bulk data transfer channel for communication between the recording and audio processing tasks for the version implemented in our system. The RTSJ version uses a shared memory buffer.

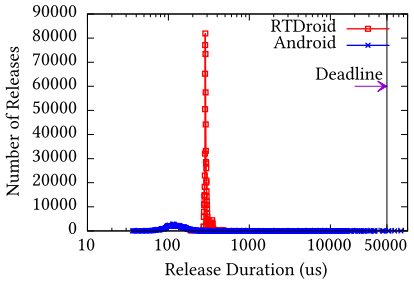
5.2.4 Results. Figure 22 shows aggregated task execution durations over each application and plots the frequency of the execution for each release. In all three applications, both RTDroid and the RTSJ easily meet the application-specific requirement (i.e. they finish well ahead of the deadline). In Figure 22(a) and (b) there is a clear difference in the execution time: The RTSJ is faster but the performance penalty of using RTDroid is not prohibitive. This observation is confirmed by the CDFs of Figure 23. The curves of the CDFs for RTSJ compared to RTDroid have the same shape.

Table 1. Task Execution Duration Statistics

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



(a) Cochlear Implant: Audio Processing



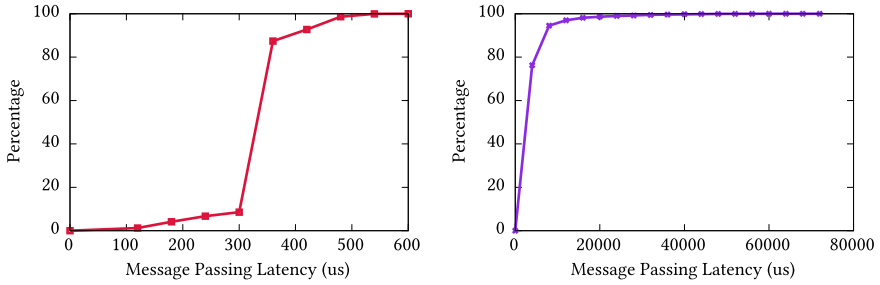
(b) jPapaBench: Stabilization Task

Fig. 24. Performance measurements on Nexus 5.

Based on this observation, as well as similar standard deviations presented in Table 1, we can conclude that RTDroid does introduce additional latency but does not impact the predictability of the code as compared to RTSJ.

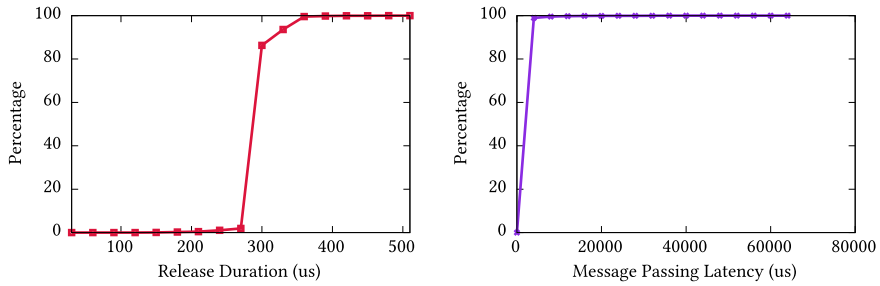
Figure 24 compares the execution time of RTDroid and plain Android. This clearly shows that there is significant variance in execution times on the Android platform. Android can, and does, miss deadlines. In Figure 24(a), we observe that the cochlear implant application runs faster in RTDroid. In Figure 24(b), one can observe that while Android can sometimes be faster than RTDroid, it can also be several orders of magnitude slower (note that the x-axis is in log scale). To quantify the overhead imposed by our system, we report the statistical results of each application in Table 1. Both our system and RTSJ have similar standard deviations even in the presence of scoped memory and channel-based communication. Our system's overhead is particularly visible in the stabilization task of jPapaBench. In the RTSJ version, the stabilization tasks reads sensor data from global shared memory buffers, performs a tight numeric computation, and produces control commands for the motors, which are also stored in global shared memory buffers. The version executing on our system, in comparison, receives sensor readings and sends control commands over channels, instead of reading from global buffers. Figures 25 and 26 show the CDFs of the experiments detailed in Figure 24. Note the difference in the x-axis; the latency in Android is much larger with much more variability. Although not surprising, our numbers indicate that a nontrivial portion of releases in Android exhibit significant delays even when not in the presence of a loaded system.

Although our system does induce additional overhead when compared to applications written in RTSJ, it does provide tangible benefits in terms of programmability. In addition to hiding the complexity of writing code that leverages scoped memory, our system also decouples configuration from application logic and simplifies interactions between components via Android-like communication over channels. Table 2 shows code metrics over three types of code—the common code in both versions of implementation (mostly the application logic), code specific to our system, and RTSJ-specific code—but excludes common libraries (i.e., the FFT and signal processing libraries for the cochlear implant). It shows that applications written for our system are implemented with



(a) Audio Processing with RTDroid (b) Audio Processing with Android

Fig. 25. CDFs of performance measurements of the Cochlear Implant on Nexus 5.



(a) Stabilization Task Duration with RTDroid (b) Stabilization Task Duration with Android

Fig. 26. CDFs of performance measurements of the jPapaBench stabilization task on Nexus 5.

fewer lines of code. This occurs because RTSJ requires developers to manually instantiate all tasks and provide release logic with the multithreading APIs. In our system, all application components are declared in the manifest and the boot process initiates and starts them. Additionally, since our system uses message passing, it removes explicit programmer-written synchronization between interacting components. There are two important consequences to the reduction of source code in favor of manifest complexity: (i) our system can automate reasoning about manifest files [34], and (ii) removal of synchronization (critical regions) reduces interference costs and simplifies validation of system schedulability.

5.3 Multiplexed Sound Applications

We evaluate the correctness of our audio framework by using two applications: device coordination and distance estimation. We discuss the applications and the results in the remainder of this section.

5.3.1 Coordinator with Surrounding Sound Playback. This application is inspired by existing work that proposes a class of audio mobile applications [13, 14]. One example shows surround sound playback from a set of mobile devices, recreating the surround sound experience in mobile scenarios and others. Such applications require timeliness and the ability of the devices to communicate using audio beacons for synchronization. Our application is designed to work as follows. We build the surround sound system using one smartphone as a coordinator which sends out an inaudible tone for synchronization every 200 *ms*. Other smartphones in the vicinity (worker devices) running the system respond to this synchronization signal with a signal of a certain frequency associated with the device. The coordinator collects the arrival times of the responses from

Table 2. Code Complexity Measurements

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

[a]Source Lines of Code as counted by David A.Wheeler's SLoC-Count.
[b]Methods or blocks protected by synchronized statements.
[c]Lines of XML cod.

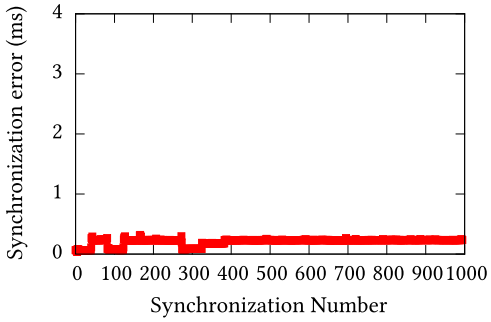


Fig. 27. Surround sound coordination system: Synchronization error in arrival time measurement.

the worker devices by analyzing the frequency response of the received audio data. Using these arrival times, we can estimate latency in coordination between devices. This information is then used to carefully schedule multichannel playback from the speakers. For accurate localization, such systems need a high synchronization accuracy, as mentioned in previous works [13, 15, 21, 24]. Figure 27 presents the synchronization errors over 1,000 synchronization beacons. We can see that the synchronization error is always well below 1 *ms*. This gives us confidence that our system can be further used to build more complex systems using sound.

5.3.2 Simulated Interactive Museum Display. The application simulates a large museum display with both audio and visual components. The display has several speakers located at different locations of the display. Our objective is to determine the speaker that is closest to the visitor and use that speaker for audio playback to produce an enhanced user experience [4, 8]. Such systems can also be used for enhancing the indoor gaming experience, where the system detects if the player is near an obstacle/checkpoint and can play sound effects, narratives for the story, instructions, and the like. We implement a simplified version of this application using our framework. We use three smartphones as speakers (with fixed location) and the user's smartphone, which can change its position with respect to the speakers. The smartphone has our app and periodically sends out inaudible audio signals. When the fixed devices detect the signals, the fixed devices initiate distance measurement with the mobile device. Upon determining the range of each of the speakers from the mobile phone, it signals the nearest speaker to start playback. Our application setup is shown in Figure 28. Points 1–8 in the figure show different listener locations. We evaluate the system by measuring how accurately the speaker devices are able to calculate the location of the listener. Figure 29 shows the distances measured by the speaker devices at each point. We can see that the fixed speaker devices can accurately range if the listener is close and even triangulate his position if required (given that there are three speakers). Using separate frequencies for different speakers, the smart phone is able to determine how far it is from each of them. It then informs the nearest speaker to engage for audio playback. Shown in Figure 29 is the timeline diagram for the interaction between the user's smartphone and the speakers at a particular display.

6 RELATED WORK

Previous attempts to make Android amenable to real-time include the work of Maia et al. who proposed four different architectures [16, 17, 19, 22, 26] that enforce a strict separation between

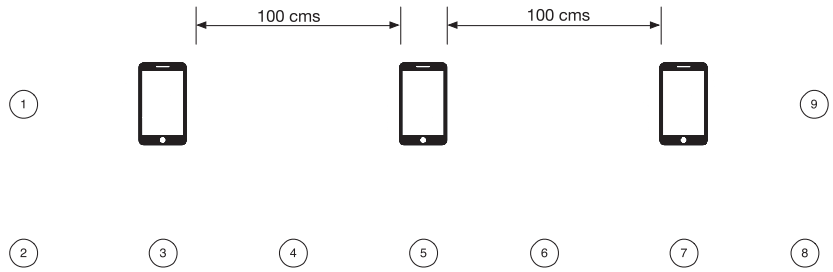


Fig. 28. Experimental setup for augmented reality application: Circles denote location of listener during the experiment. Device 1 is on the left, Device 2 in the middle, and Device 3 on the right.

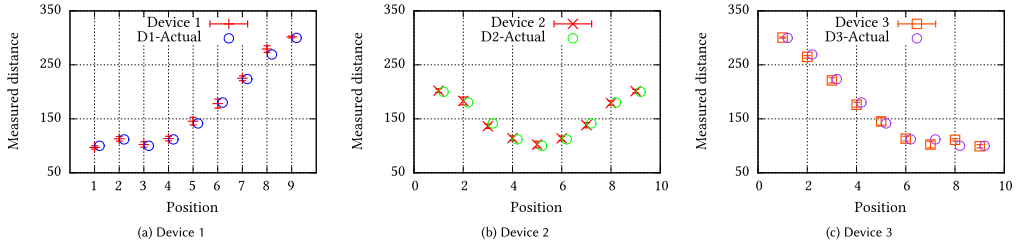


Fig. 29. Actual vs. estimated distance from speakers.

real-time and non–real-time apps. Kang et al. [12] and Ruiz et al. [27] implemented such separation in the standard Linux kernel, assigning one or more cores for real-time tasks and isolating those cores from the rest of the system. Our work strives to make such interactions safe. Prior work [5, 10] on real-time garbage collection for Android focused on reducing the latency the GC could induce on computation. This was accomplished by modifying the Dalvik VM to explicitly trigger the GC based on heuristics to reduce pause time during critical periods. Choosing when to run the GC is unfortunately difficult and requires reasoning about the schedule of tasks in the system as well as when those tasks allocate memory. This is exacerbated when communication between applications is permitted. Prior work has also explored how components interact through intents, providing a mechanism to prioritize intents [11], but did not provide any memory bounds on communication. Our work provides static bounds on memory consumption of communication between tasks, allows communication between real-time and non–real-time tasks, and observes that only prioritizing intents can induce priority inversion in the callbacks that handle those intents. Other efforts have the left the Android framework unmodified [20], instead focusing on exposing the degree of jitter present in sensor data in the system so that applications can make necessary adjustments. Our system strives to eliminate such jitter.

In addition to integrating a real-time–capable VM and a real-time operating system, RTDroid [33] explored how to add priorities to three exemplar constructs in Android and to study the feasibility of adding guarantees within the internals of the Android framework. We adopt the priority mechanisms provided by RTDroid for communication with framework services such as sensors, but observe that they are not enough to correctly encode intents or provide memory bounds. The channels provided by our system replace the communication mechanism of the original RTDroid. The original RTDroid did not provide a mechanisms for interacting with legacy code and did not provide support for Android APIs; programmers were stuck using libraries provide by the VM and the RTSJ. Our work provides an Android-like programming model that hides the complexities of

the RTSJ, allows for interaction with legacy code, and disentangles configuration from application logic.

Our work leverages previous results on region-based memory management [30]. Scope memory was introduced in the RTSJ [7] to avoid GC interference. Scope memory allows the system designer to prove properties about the predictability of the overall system, including static memory bounds [28]. In our system, scopes are mostly *hidden* from the programmer. The developer needs to configure the system to specify necessary bounds but does not need to worry about adhering to the scope memory rules enforced by RTSJ. Bounds are specified declaratively through our manifest extensions, instead of programmatically, thereby abstracting out configuration from function. Since services communicate through message passing, the complexity of reasoning about cross-scope references and scope nesting levels (scope stacks) is handled seamlessly by our underlying system. This largely removes the cognitive burden from the programmer of using scope memory in application development.

6.1 Sound Processing on Android

Android applications with real-time timing constraints and low-latency audio processing require significant development effort. `OpenSL ES` [1] 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` [1] 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 [21, 24, 25, 29, 35] require the emission of high-frequency acoustic signals to achieve acceptable accuracy; surround sound play [13] 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 [14] outlined two main challenges of real-time audio management in Android: (i) tight timing requirement of audio requests and (ii) unpredictable dispatching latency for audio playback and record. It addressed these two problems via SounDroid framework APIs, which provide 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.

7 CONCLUSION

Real-time capabilities have the potential of increasing the range of applications that can be written on the Android platform. This article is a step toward turning Android into a high-level real-time programming environment in which developers can freely mix time-critical code with code that is unaware of any timing constraints. In this article, we have shown that the changes required to the Android programming model from the programmer's perspective are quite modest. Our constructs, which expose familiar Android interfaces, additionally provide statically specified memory bounds and priority awareness. We report on a number of experiments to validate our system. The micro-benchmarks present some evidence that the implementations of our extensions are robust to memory and computational stress. The application benchmarks show that our system outperforms

the Android platform on real-time metrics and is easier to program than the Real-time Specification for Java. Overall, we argue that RTDroid is a viable point in the design space of embedded development environments.

ACKNOWLEDGMENTS

We thank the reviewers for their constructive feedback. This work received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (award 1823230, 1523426, 1513006, 1405614, 1749539, 1544542 and 1518844) as well as ONR (award 503353).

REFERENCES

[1] [n. d.]. OpenSL SE for Android. https://developer.android.com/ndk/guides/audio/opensl-for-android.html.

[2] Hamza Ali, Arthur P. Lobo, and Philipos C. Loizou. 2013. Design and evaluation of a personal digital assistant-based research platform for cochlear implants. *IEEE Transactions on Biomedical Engineering* 60, 11 (2013). DOI: https://doi.org/10.1109/TBME.2013.2262712

[3] Adam Czerniejewski, Shaun Cosgrove, Yin Yan, Karthik Dantu, Steven Y. Ko, and Lukasz Ziarek. 2016. jUAV: A Java based system for unmanned aerial vehicles. In *Proceedings of the International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*. DOI: https://doi.org/10.1145/2990509.2990511

[4] Lesley Fosh, Steve Benford, Stuart Reeves, Boriana Koleva, and Patrick Brundell. 2013. See me, feel me, touch me, hear me: Trajectories and interpretation in a sculpture garden. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. DOI: https://doi.org/10.1145/2470654.2470675

[5] Thomas Gerlitz, Igor Kalkov, John Schommer, Dominik Franke, and Stefan Kowalewski. 2013. Non-blocking garbage collection for real-time Android. In *Proceedings of the International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*. DOI: https://doi.org/10.1145/2512989.2512999

[6] Girish Gokul, Yin Yan, Karthik Dantu, Steven Y. Ko, and Lukasz Ziarek. 2016. Real time sound processing on Android. In *Proceedings of the International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*. DOI: https://doi.org/10.1145/2990509.2990512

[7] James Gosling and Greg Bollella. 2000. *The Real-Time Specification for Java*. Addison-Wesley.

[8] Florian Heller and Jan Borchers. 2014. AudioTorch: Using a smartphone as directional microphone in virtual audio spaces. In *Proceedings of the Conference on Human-computer Interaction with Mobile Devices & Services (MobileHCI)*. DOI: https://doi.org/10.1145/2628363.2634220

[9] Tomas Kalibera, Pavel Parizek, Michal Malohlava, and Martin Schoeberl. 2010. Exhaustive testing of safety critical Java. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*. DOI: https://doi.org/10.1145/1850771.1850794

[10] Igor Kalkov, Dominik Franke, John F. Schommer, and Stefan Kowalewski. 2012. A real-time extension to the Android platform. In *Proceedings of the International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*. DOI: https://doi.org/10.1145/2388936.2388955

[11] Igor Kalkov, Alexandru Gurghian, and Stefan Kowalewski. 2014. Predictable broadcasting of parallel intents in real-time Android. In *Proceedings of the International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*. DOI: https://doi.org/10.1145/2661020.2661023

[12] Hyeongseok Kang, Dohyeon Kim, Jeongnam Kang, and Kanghee Kim. 2016. Real-time motion control on Android platform. *The Journal of Supercomputing* 72, 1 (2016). DOI: https://doi.org/10.1007/s11227-015-1542-5

[13] Hyosu Kim, SangJeong Lee, Jung-Woo Choi, Hwidong Bae, Jiyeon Lee, Junehwa Song, and Insik Shin. 2014. Mobile maestro: Enabling immersive multi-speaker audio applications on commodity mobile devices. In *Proceedings of the International Joint Conference on Pervasive and Ubiquitous Computing (Ubicomp)*. DOI: https://doi.org/10.1145/2632048.2636077

[14] H. Kim, S. Lee, W. Han, D. Kim, and I. Shin. 2015. SounDroid: Supporting real-time sound applications on commodity mobile devices. In *Proceedings of the Real-Time Systems Symposium (RTAS)*. DOI: https://doi.org/10.1109/RTSS.2015.34

[15] Kaikai Liu, Xinxin Liu, and Xiaolin Li. 2013. Guoguo: Enabling fine-grained indoor localization via smartphone. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*. DOI: https://doi.org/10.1145/2462456.2464450

[16] Cláudio Maia, Luís Nogueira, and Luis Miguel Pinho. 2010. Evaluating Android OS for embedded real-time systems. In *Proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*.

[17] Wolfgang Mauerer, Gernot Hillier, Jan Sawallisch, Stefan Hönick, and Simon Oberthür. 2012. Real-time Android: Deterministic ease of use. In *Proceedings of the Embedded Linux Conference Europe (ELCE)*.

[18] Noah J. Myrent, Douglas E. Adams, Gustavo Rodriguez-Rivera, Denis A. Ulybyshev, Jan Vitek, Ethan Blanton, and Tomas Kalibera. 2015. A robust algorithm to detecting wind turbine blade health using vibro-acoustic modulation and sideband spectral analysis. In *Proceedings of the Wind Energy Symposium*. DOI: https://doi.org/10.2514/6.2015-1001

[19] Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi, and Soo-Mook Moon. 2012. Evaluation of Android Dalvik virtual machine. In *Proceedings of the International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*. DOI: https://doi.org/10.1145/2388936.2388956

[20] E. Peguero, M. Labrador, and B. Cook. 2016. Assessing jitter in sensor time series from Android mobile devices. In *Proceedings of the International Conference on Smart Computing (SMARTCOMP)*. DOI: https://doi.org/10.1109/SMARTCOMP.2016.7501679

[21] Chunyi Peng, Guobin Shen, Yongguang Zhang, Yanlin Li, and Kun Tan. 2007. BeepBeep: A high accuracy acoustic ranging system using COTS mobile devices. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*. DOI: https://doi.org/10.1145/1322263.1322265

[22] Luc Perneel, Hasan Fayyad-Kazan, and Martin Timmerman. 2012. Can Android be used for real-time purposes?. In *Proceedings of the Computer Systems and Industrial Informatics (ICCSII)*. DOI: https://doi.org/10.1109/ICCSII.2012.6454350

[23] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. 2010. High-level programming of embedded hard real-time devices. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. DOI: https://doi.org/10.1145/1755913.1755922

[24] Jian Qiu, David Chu, Xiangying Meng, and Thomas Moscibroda. 2011. On the feasibility of real-time phone-to-phone 3D localization. In *Proceedings of the Conference on Embedded Networked Sensor Systems (SenSys)*. DOI: https://doi.org/10.1145/2070942.2070962

[25] Tauhidur Rahman, Alexander T. Adams, Mi Zhang, Erin Cherry, Bobby Zhou, Huaishu Peng, and Tanzeem Choudhury. 2014. BodyBeat: A mobile system for sensing non-speech body sounds. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MOBISYS)*. DOI: https://doi.org/10.1145/2594368.2594386

[26] Ganesh Jairam Rajguru. 2014. Reliable real-time applications on Android OS. *International Journal of Management, IT and Engineering* 4, 6 (2014).

[27] Alejandro Pérez Ruiz, Mario Aldea Rivas, and Michael González Harbour. 2015. CPU isolation on the Android OS for running real-time applications. In *Proceedings of the International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*. DOI: https://doi.org/10.1145/2822304.2822317

[28] Daniel Tang, Ales Plsek, and Jan Vitek. 2010. Static checking of safety critical Java annotations. In *Proceedings of the International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*. DOI: https://doi.org/10.1145/1850771.1850792

[29] Stephen P. Tarzia, Peter A. Dinda, Robert P. Dick, and Gokhan Memik. 2011. Indoor localization without infrastructure using the acoustic background spectrum. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MOBISYS)*. DOI: https://doi.org/10.1145/1999995.2000011

[30] Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. DOI: https://doi.org/10.1145/174675.177855

[31] Yin Yan, Shaun Cosgrove, Varun Anand, Amit Kulkarni, Sree Harsha Konduri, Steven Y. Ko, and Lukasz Ziarek. 2016. RTDroid: A design for real-time Android. *IEEE Transactions on Mobile Computing* 15, 10 (2016). DOI: https://doi.org/10.1109/TMC.2015.2499187

[32] Yin Yan, Karthik Dantu, Steve Ko, Jan Vitek, and Lukasz Ziarek. 2017. Making Android run on time. In *Proceedings of the Real-Time and Embedded Technology and Application Symposium (RTAS)*. DOI: https://doi.org/10.1109/RTAS.2017.38

[33] Yin Yan, Sree Harsha Konduri, Amit Kulkarni, Varun Anand, Steve Ko, and Lukasz Ziarek. 2014. Real-time Android with RTDroid. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MOBISYS)*. DOI: https://doi.org/10.1145/2594368.2594381

[34] Yin Yan and Lukasz Ziarek. 2018. Application validation on RTDroid. *SIGBED Review* 15, 4 (2018). DOI: https://doi.org/10.1145/3269482.3269484

[35] Zengbin Zhang, David Chu, Xiaomeng Chen, and Thomas Moscibroda. 2012. SwordFight: Enabling a new class of phone-to-phone action games on commodity phones. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MOBISYS)*. DOI: https://doi.org/10.1145/2307636.2307638

Received December 2017; revised September 2018; accepted October 2018