Making Android Run on Time

Yin Yan Karthik Dantu Steve Ko Jan Vitek Lukasz Ziarek

***Abstract*—**

I. Introduction

The pervasiveness of cloud services, smartphones, wearables, Internet-of-Things (IoT) devices, and networked embedded devices has lead to the flow of computing, communication, and control towards the *edge* of the Internet, now colloquially referred to as "fog computing." These embedded devices, called edge devices, are typically networked with each other, various sensors, small embedded devices, and periodically with cloud services. These edge devices perform various types of computations, some which are non real-time, like communicating with cloud services (*e.g.* sending biometric data to a health provider) or providing interactivity through GUIs (*e.g.* adjusting settings and sensitivity to customize user experience), while many others have specific timeliness constraints like computing a response to a sensor reading (*e.g.* a fall detector must process accelerometer data at a 20Hz rate [14]) or processing a signal at a specific rate (*e.g.* external processing device for a cochlear implant).

As IoT and fog computing become mainstream, we are left with a fundamental question: *how do we program all these different devices?* A potential starting point is Android, a versatile and attractive environment for developing embedded systems with a set of applications that is growing well beyond the original market segment its designers had envisioned. Indeed, the consumer medical device industry is investing significant resources in exploring Android as a platform for on-demand, personalized health care applications. The sensing community is also investigating leveraging Android for audio-based indoor localization [10], harmonized sound reproduction [9], timely sound delivery [8], and high-rate sensing [13] amongst others. The authors themselves have used Android as a framework for writing software that controls wind farms.

It is not surprising that Android has little support for timeliness guarantees, since it was designed for mobile devices and optimized for device mobility, user experience, and energy efficiency. The goal of our work is to look for non-invasive ways to adopt an Android like system for designing edge devices. By non-invasive, we mean that existing Android applications should continue to work without changes on the platform and new applications can be written in a style that is not too alien for Android developers. Our work builds on previous research that enforced strict isolation between computations [11], studies on how to reduce memory management latencies [6], added priorities to core communication primitives [7], and added priorities in the lower levels of the Android stack [22]. The contribution of our work is an Android system for providing soft real-time guarantees to applications that use it, while allowing legacy code to run as before. More precisely, we propose the following changes to the platform:

- **Declarative timeliness:** A declarative mechanism for programmers to specify the timeliness and resource requirements for their applications, without entangling such specifications into the application.
- **Priority-aware communication:** Specialized communication primitives that preserve the Android communication model, but provide programmers control over how components of differing priority level communicate.
- **Pauseless memory management:** An implementation of our proposed constructs that internally leverages region-based memory management to avoid interference from the garbage collector.
- **Extended APIs:** Extensions of existing constructs to specify requisite real-time behaviors and interactions.

To validate our design and evaluate the quality of our implementation, we report on three applications that we have deployed on our system, both on commodity embedded boards as well as smart phones. They are a cochlear implant, a wind turbine health monitor, and a UAV flight control benchmark. The results of our experiments illustrate that, at least in these three use cases, the modified platform delivers significantly better time predictability than stock Android. We provide software quality metrics to back up our claims that our system not only allows developers to continue writing code in a familiar style and interact with legacy applications, but also avoids mixing real-time configuration code with application logic.

II. An Android-Enabled Cochlear Implant

A cochlear implant restores hearing abilities through an electronic device surgically inserted on the patient's inner ear. The device relies on external components to capture ambient audio, convert it into digital signals, and translate the signals into electrical energy. Recently, there has been interest in leveraging smartphones [3] to provide additional services such as on-the-fly translation or advance noise cancellation. In such a scenario, a smartphone records audio streams and processes them. To provide acceptable performance sound samples should be processed at rate of one ever 8 ms. We now show the limitations and challenges of implementing such an application in Android.

A natural design for an Android version of such an application separates the user interface that controls volume and

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

Fig. 1: Audio Configuration UI (non-realtime)

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

Fig. 2: Audio Processing Service (real-time)

noise reduction, from the sound processing component that converts audio frames into signals. Android provides three software architectural elements, namely *services*, *activities*, and *broadcast receivers*, for, respectively, background computations that do not require user input, foreground computations with user input, and system-wide events. Sound processing is best modeled as a service, while the UI should be an activity as shown in Figure 1 and 2. Of course, Android does not guarantee that the components will not interfere. For example, the audio processing at line 4 could be delayed by the execution of the UI as the Android scheduler is unaware of any notion of priority.

Communication between components is another tricky issue. Android offers two communication mechanisms: messages and intents. Messages are received by Android's `Handler` class which is a unique mailbox for all messages directed at a component. As there is no notion of message priority, the single, first-in first-out message queue attached to a `Handler` will lead to priority inversion with time-critical messages being delayed by messages that have no timeliness requirements. The Android class `Intent` is an event that triggers execution of callbacks in components that have registered for it. This construct is useful for publish-subscribe style interaction but can also lead to priority inversion as callbacks are executed by the receiver which may have different timeliness requirements than the component that raised the intent.

Memory pressure is also a concern. For instance, the message buffers are arbitrary length, this can impact all components. Android provides no mechanism other than garbage collection to manage memory. The Android garbage collector does not have real-time guarantees. What makes matters worse, there is no way to bound memory consumed by different components. Thus a stray non-critical component can compromise the whole system.

Ideally, a programmer should be able to manage application logic and configuration, such as the rate of processing, separately. Android provides such separation through its *manifest*, an XML file that specifies properties of the program. This allows the programmer to declaratively limit the interactions of components. However, the manifest cannot express configuration of parameters that affect the timeliness of an application. For example, there is no mechanism to limit the rate of interaction through message passing or callbacks, nor is there the capability to express component priorities and memory allowances.

Fig. 3 illustrates the architecture of the Cochlear application

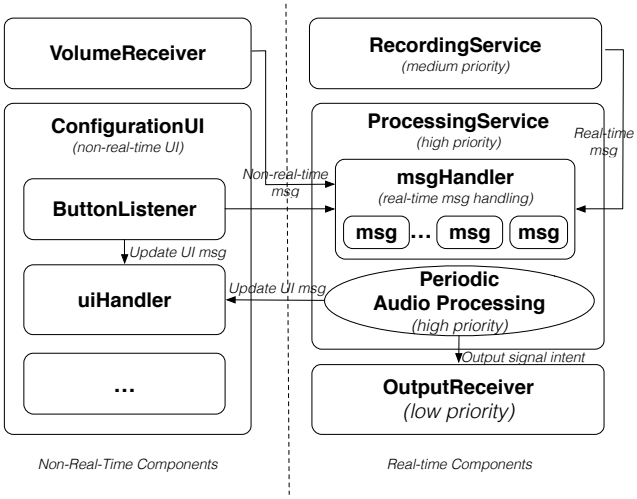


Fig. 3: Architecture of the Cochlear Implant Application

we have implemented in RTDroid. We separate between real-time (`RecordingService`, `ProcessingService`, and `OutputReceiver`) and non-real-time components (`VolumeReceiver` and `ConfigurationUI`). The real-time components have priorities attached and use properly modified communication services that prioritize messages. `ConfigurationUI` has a `Handler` for other components to update the UI, and a non-real-time receiver listens on volume key events. Note that this non real-time component receives messages from both non real-time and real-time components. Similarly, the `ProcessingService` also receives mesages from both non real-time components (`VolumeReceiver` and `ButtonListener`) and a real-tine component (`RecordingService`). Specialized communication services allow for communication between real-time and non real-time components, while preserving memory bounds required by real-time components. To ensure bounded memory for the real-time components, our system utilizes pre-allocated messages. Non real-time components allocate their own message in heap memory, creating isolation between message utilized by real-time components and those which are use by non real-time components. In addition each component is provided with a bounded memory region for its computation. Conceptually this region is divided into two parts, a persistent portion with the lifetime of the component and a release portion which is cleared and made available for each release of a periodic component (*e.g.*`ProcessingService`).

III. Real-Time Android for Edge Devices

To allow a developer to express the real-time computation of an application, we provide three constructs—*real-time services*, *real-time receivers*, and *periodic tasks*. Our real-time service is a counterpart to Android's service, and a programmer can use it to develop a one-shot real-time task, thereby modeling either aperiodic or sporadic computation. The notion of periodic computation is foreign to Android and as such we introduce a new construct (`PeriodicTask`) to model such behavior. For a service that runs periodically, a programmer can use our periodic task internally within the service. A programmer can leverage our real-time receiver (`RealtimeReceiver`) to react to system-wide events delivered via `Intents` For each of these constructs, the programmer can statically assign a priority to a real-time service, a starting time relative to the time when the system started, a deadline by when the computation should complete, as well as the amount of memory it is allowed to consume at run time. If a component exceeds its specified memory bound, an out of memory exception is generated. We enable this with our manifest extension described further in Section III-C.

We do not provide a counterpart to Android's activity, since we do not consider UI to be a real-time component. Instead, we allow for interaction between UI components and real-time components through our message channels. This allows the programmer to create UIs using standard Android mechanisms.

Real-Time Service: Our real-time service is defined as an abstract class, and a programmer needs to implement its callbacks. For example, `onCreate()` is invoked when a real-time service is first created, `onStartCommand()` is invoked when a real-time service is starting, etc. Often times, it is `onStartCommand()` that implements meaningful application logic. Fig. 4 shows a real-time service from our cochlear implant application example, `ProcessingService`. It implements `onStartCommand()` to start a periodic task. Note that real-time service still only performs *one* computation and thus can be considered one-shot, though in this case that computation is periodic. A real-time service executes in its own thread; Android does not provide such semantics to its services (*i.e.*, all services run in the main thread). This change is necessary in order to enable individual priority assignment for each real-time service.

Periodic Task: A periodic task is a new construct that we add to enable periodic computation and is a sub component of our real-time service. In addition to the characteristics of its parent service, a period task requires the programmer to specify its period. Fig. 4 shows an example of this construct, which processes audio input periodically for our cochlear implant application. As shown in the example, a programmer needs only to implement `onRelease()` in the application. The body of `onRelease` represents the computation to be executed periodically.

Real-Time Receiver: Unlike Android, the real-time receiver is a persistent construct, meaning we reuse the real-time receiver object to eliminate the deallocation and reallocation of the real-time receiver for each intent received. As a direct consequence, our real-time receiver is capable of only processing *one* intent at a time. In Android a new broadcast receiver is allocated whenever an `Intent` is received, which results in frequent object allocation and deallocation if many `Intents` are sent from a misbehaving component.

```
1  class ProcessingService extends
       RealtimeService{
2    PeriodicTask task = new PeriodicTask(){
3     public void onRelease(){
4       /* periodic audio processing logic */
5    } }; ...
6    public int onStartCommand(...){
7     /* Each registered task starts after the
8      * onStartCommand() callback. */
9     registerPeriodicTask("processingTask",
          task);
10   }
11 }
```

Fig. 4: Real-Time Service and Periodic Task Example

For a real-time receiver, a programmer needs to implement callbacks to express application logic. The `onReceive()` callback defines logic to react to a system-wide event and is invoked when an `Intent` is received. We introduce a new callback, called `onClean()`, which is used to clean up or reset all class variables in a real-time receiver. This callback is used to cleanup any state between processing intents and is necessary if the programmer wishes to have a *stateless* processing of `Intents`. We note, however, that the `onClean()` callback does not need to be leveraged if the receiver only modifies local variables. If `onClean()` is not used, the programmer can express state that persists between arrivals of `Intents` because the same real-time receiver is used. Referring back to our CI application example, we can implement `OutputReceiver` as a real-time receiver to react to the processed audio output sent by the `ProcessingService`.

Callback Semantics: In all of our real-time constructs, communication causes a construct to execute application logic; either directly via a callback or indirectly as a result of message processing. Thus far we have made the implicit assumption that messages and `Intents` acquire the priority of their sending component. Callback invocation, however, raises an interesting question in terms of priority assignment for all components involved—its caller, its callee, and its `Intent` delivery channel. If we consider the `Intent` and callback invocation priority to be the same, the priority can be either sender's or receiver's priority. If the callback inherits the priority of the receiver, we might lose out on prioritized delivery of messages from high-priority senders. If the callback inherits the priority of the sender and the sender priority is low, high priority processes may impede the execution of the receiver callback invocation.

Thus in our system, we decouple `Intent` delivery from

the callback execution in the triggered component. The Intent delivery portion is based on the properties our real-time channel provides (described later in this section) that prioritizes Intents based on the priority of different senders. The execution of the callback itself, however, uses the priority of the triggered component. Multiple callbacks triggered by a given component sending Intents are serialized, guaranteeing in order execution of the callbacks. For example, in the Cochlear Implant application when ProcessingService sends processed audio output to OutputReceiver through a real-time Intent broadcast channel. The Intent broadcast channel guarantees that the Intent is delivered to the OutputReceiver with the priority of the ProcessingService, and the callback is invoked asynchronously with the priority of the OutputReceiver.

A. Real-time Channels

Our system provides four types of channels for communication: (1) a real-time message passing channel for message passing, (2) a real-time broadcast channel for intent broadcasting and callback invocation, (3) an added bulk data transfer channel that can exchange large data without copying, and (4) a cross-context channel that enables the data exchange between standard Android constructs (non real-time) and our newly introduced constructs. We focus our semantic discussions on real-time message passing and cross-context channels as real-time broadcast and bulk data transfer channels share similar semantics to our real-time message passing channel except that the former allows for multiple recipients and the later is specialized for large messages. We discuss the differences in the underlying implementations of all channels in Section IV.

We adopt Android conventions and require the programmer to declaratively specify for each channel, its name, events associated with the channel, the data type, and size of the channel. To send or receive the message on each channel, the real-time components have to specify the number of messages that they send or receive per release. This specification ensures that an implementation of our system is able to preallocate the messaging objects and enforce memory bounds for all channels. Our system assumes the existence of one primordial cross-context channel to facilitate interaction with other Android applications and services and an implementation must ensure the creation of such a channel at boot time. All other channels are explicitly created by the programmer.

Real-time Message Passing Channel: The real-time message passing channel has three major differences compared to Android: (1) the instance of communication construct (RealtimeHandler) *must* be implemented and registered in a real-time service; (2) only primitive array or fixed length byte buffer can be associated with the real-time message object; and (3) the number of messages in a channel is bounded.

In Android, a construct typically obtains a message object prior to sending the message. If we were to use such a method, the programmer would be required to provide the bound not only on messages actively being sent, but also actively being constructed to ensure bounded memory use during communication. Therefore, instead of obtaining the message directly and holding on to a limited resource, our system provides a mechanism that defers obtaining a message until the send operation itself. This prevents a construct from obtaining a message and waiting an arbitrary amount of time before sending it.

```
1  MessageClosure closure = new
       MessageClosure(){
2      @Override
3      public RTMSG genMsg(RTMSG msg){
4          //msg.obtain() is called in framework
5          Bundle b = msg.getData();
6          b.setInt(index, 3);
7          ...
8          return msg;
9      }
10 };
11 rtmsg.send("channel-name", closure);
```

Fig. 5: Real-time Message Interface

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

Fig. 6: Real-time Component Declaration

Our system provides MessageClosure as an abstract class that encapsulates the sender's message population logic in genMsg(), and gives us the ability to request and populate the message as an atomic operation from the perspective of the sender. This unifies message population and queuing and is shown in Fig. 5. In our cochlear implant application, the message passing logic in ProcessingService can be implemented with the real-time message passing channel. Thus, the high priority messages from RecordingService can be prioritized over the messages from non-real-time ConfigurationUI.

Cross-Context Channel: Our cross-context channel is introduced to allow Android's Activity to communicate with our systems's real-time components. To use the cross-context channel from another Android application, that application must declare an Android Service (RTsProxyService) that subscribes to channels declared in an application that uses our real-time constructs. This will allow activities in Android applications (non real-time) to send an intent to real-time components through the proxy service. For an activity to receive Intents from our real-time constructs, the activity

```
<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. 7: Real-time Channel Declaration

can subscribe to `Intents` defined by our real-time constructs. To preserve memory bounds, the number of `Intents` in a cross-context channel is bounded and each `Intent` has a fixed-length byte buffer payload.

B. Real-Time Memory

For a real-time component, we need a mechanism to enforce memory bounds. We accomplish this by leveraging bounded memory regions for each construct, *Persistent Memory* and *Release Memory*, with specific lifetimes defined by the lifetime and releases of our constructs respectively. The programmer only needs to specify the size of these memory regions in the manifest. Our system, however, must manage these regions.

For every declared component, our system ensures exclusive access to an unused region in memory (Persistent Memory). Similarly, our system ensures an unused region of memory for every release of a construct (*Release Memory*). The total memory bound of a real-time construct is the sum of its persistent memory, release memory, and the release memory of their related internal components.

C. Real-Time Manifest

We extend Android's manifest (Fig. 6) to define real-time properties for our real-time constructs like `priority`, `memSizes` (total, persistent, and release), and `release` have been added. The association between a periodic task and its parent construct is also specified in the manifest through a `periodic-task` tag. Fig. 7 shows an example of a real-time message passing channel declaration with a name attribute as an event identifier. Each channel should define its runtime behavior via: `type` attribute (channel communication type), `order` (message delivery order), `execution` (execution priority of the invoked function), `drop` (message dropping policy), `data size` and `data type`. Components can use `intent-filter` to identify themselves as *publishers* or *subscribers* of a channel and to specify the number of messages sent or processed in each callback function release.

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

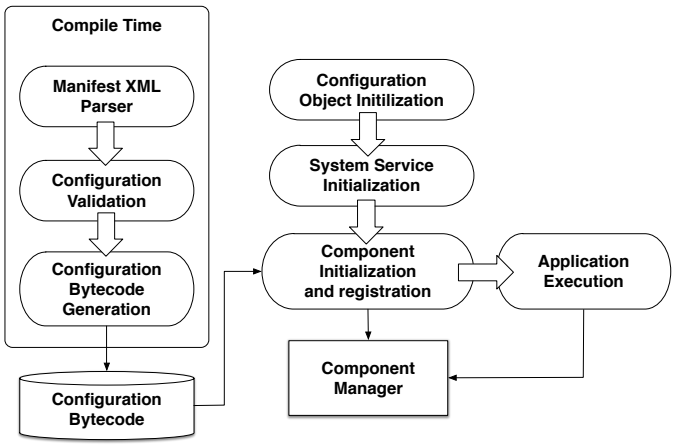


Fig. 8: RTDroid Bootstrap

IV. Implementation

Our implementation is realized in the RTDroid open source framework [22], which leverages an RTOS and an RT JVM (Fiji VM [18]). On top of this base system, we have implemented our system. The system is open source and available at rtdroid.cse.buffalo.edu.

A. Application Bootstrap

The system bootstrap is divided into two stages: compile time and application runtime and is shown in Fig. 8. This two stage bootstrap ensures that all necessary memory regions are pre-allocated and components are correctly configured. This is possible because our manifest extension lists all application components (e.g., real-time services and receivers), the communication channels they use, and the real-time properties they require. During compile time, our framework parses the manifest file of an application, runs verification checks, and emits *configuration object bytecode* for all components. This configuration object 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, as we describe next.

B. Memory Management

For real-time applications, providing memory usage guarantees implies that the underlying system should satisfy the following two properties: (1) when a real-time construct allocates an object, it should be predictable *i.e.*, the object allocation should not be blocked by the memory usage of any other construct, either causing allocation failure or delays; (2) the collection of unused objects should also be predictable, *i.e.*, the underlying memory management scheme should not interfere with the execution of a real-time component.

In order to provide these two properties, we use scoped memory, a region based memory management scheme, as defined by RTSJ. Scoped memory provides fixed amount of

memory for real-time tasks through the usage of memory regions called scopes 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. RTSJ provides three types of memory areas: (1) *heap memory*, which behaves like the traditional heap memory as defined by the Java specification, (2) *immortal memory*, which is never reclaimed, and (3) *scoped memory*, which provides bounded, temporal memory regions, whose lifetimes are determined by the threads using that scope. To guarantee referential integrity, RTSJ imposes a number of rules on how scoped memory must be used, such as (1) the objects in a scope are only reclaimed after all threads in that scope have finished, (2) every thread must enter a scope from the same parent scope – the *single parent rule*, and (3) a scope with a longer lifetime cannot hold a reference to an object allocated in a scope with a shorter lifetime.

Fundamentally, we leverage scoped memory to provide memory bounds corresponding to the lifetime of different computations as well as data across computations (messages). To provide memory boundary for each component, we group the computation and associated allocations performed by the computation into two separate lifetimes: (1) the duration of the *lifetime* of the component (called *persistent scope*), and (2) the duration of *one callback function invocation* of the component, which maybe periodic or aperiodic (called *release scope*). The scopes correspond directly to the types of memory defined by our system; persistent memory and release memory respectively. 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. Similarly channels are allocated in *immortal memory*. Non real-time constructs leverage the heap.

C. Component Lifetime Management

Component lifetime management requires, (1) observing component priorities, (2) guaranteeing the periodicity of a component, (3) automatically managing memory, and (4) guaranteeing per-component memory bounds. For the first two, we extend an existing mechanism in the RTDroid open source framework, which provides priority-aware scheduling. In order to provide the last two, our component manager carefully uses different types of memory for each construct.

Real-time Service: We implement our real-time service using Fiji's real-time thread construct, providing the basic support for priority assignment and scheduling. By default, when a real-time service is initialized, it is assigned a *persistent scope* that shares the same lifetime as the real-time service, allocated when the service starts and deallocated when the service terminates (shown in Fig. 9). All class variables of the real-time service are allocated in this scope. In addition, if the real-time service uses communication channels (which should be declared in its manifest), corresponding `Intent` queues for the channels are allocated in the persistent scope as well.

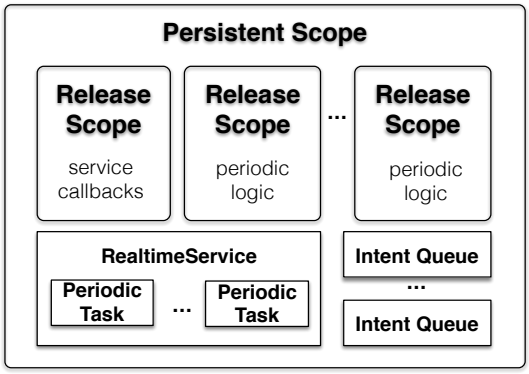


Fig. 9: Scope Structure for `RealtimeService`

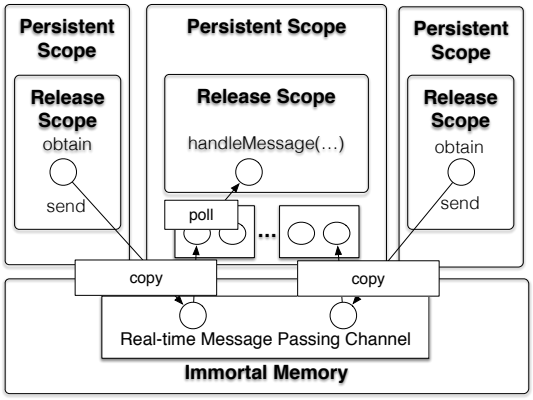


Fig. 10: Real-time Message Passing Channel

Whenever a callback is invoked by the service, it is assigned a *release scope* used for allocating local objects for the callback. This release scope shares the same lifetime as the callback – it is allocated when the callback is invoked, and deallocated when it returns. Similarly, when a periodic task is started in a real-time service, it is also assigned a release scope. This guarantees referential integrity and ensures that all memory uses of the real-time service are contained and isolated.

a) Real-time Receiver: We implement our real-time receiver using Fiji's asynchronous event handler 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. Similar to our real-time service, we allocate a persistent scope for each real-time receiver. Class variables are allocated in the persistent scope and a callback gets assigned a release scope for its local objects. All memory uses of a real-time receiver are contained and isolated within its scope.

D. Real-time Communication Channels

Real-time Message Passing Channel The open-source RTDroid [22] framework already provides priority aware message delivery. We utilize this feature to provide the priority

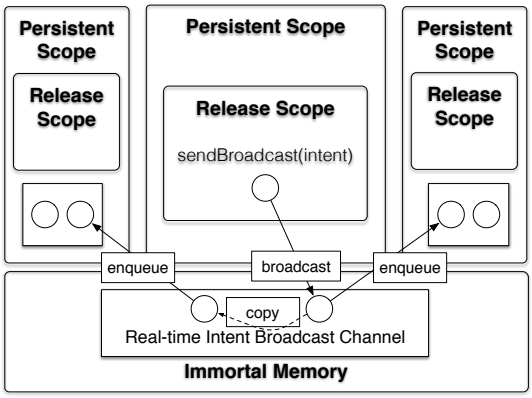


Fig. 11: Real-time Intent Broadcast Channel

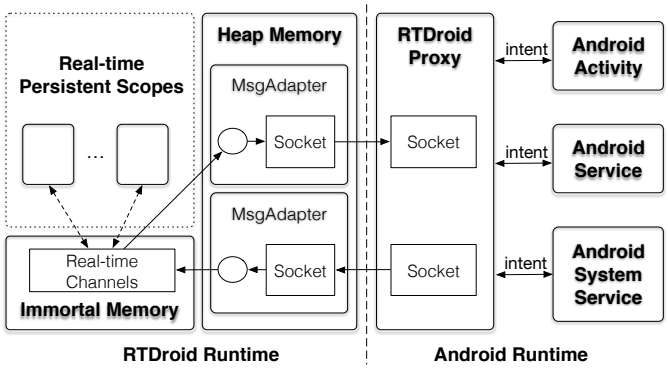


Fig. 12: Cross-Context Channel

aware message delivery required by our message passing channel. In that framework, the real-time handler constructs use priority inheritance to prioritize the incoming messages from senders with different priorities. All messages are queued and handled based on their senders' priority. Consider Fig. 10 that illustrates our design and how it fits into our scope memory hierarchy.

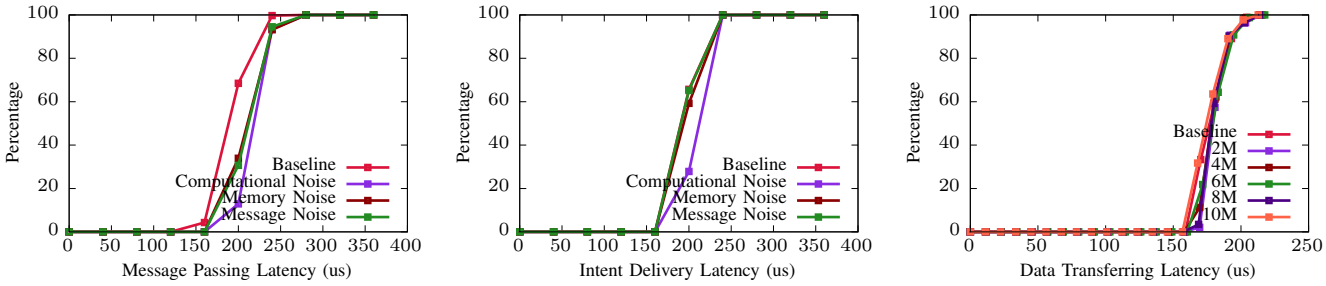
Our RTDroid extension pre-allocates the real-time message objects in the persistent scope of the receiving construct. The message is served based on sender priority as a message pool. As a direct consequence, the obtain operation can fail when no messages are available as they have been given to higher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

Conceptually, once a message has been obtained, the sender must copy the data to be sent from its local allocation context to the message pool of the receiving construct. This ensures that a sender cannot utilize or fill the allocation context of a receiver. The message content will only be copied to the receiver when the receiver is ready to receive and process the message. This strategy keeps the amount of memory dedicated to message passing constant. The sender must utilize its own memory (heap or its release scope) to create the data that it wishes to send and cannot use system resources to store this data unless it is able to obtain a message object.

Real-time Intent Broadcast Channel The real-time intent broadcast channel is designed to trigger the callback function of the real-time component. Unlike the priority inheritance in the real-time message passing channel, we decouple the priority based mechanism for determining message delivery from the execution context of the service which executes the callbacks at its own priority. Fig. 11 shows how an intent object is duplicated in the immortal memory, and copied to the intent queues in multiple subscribers. The memory usage of the broadcast channel is bounded, because our RTDroid extension pre-allocates intent objects in each subscriber's intent queue based on the size and type of data in manifest.

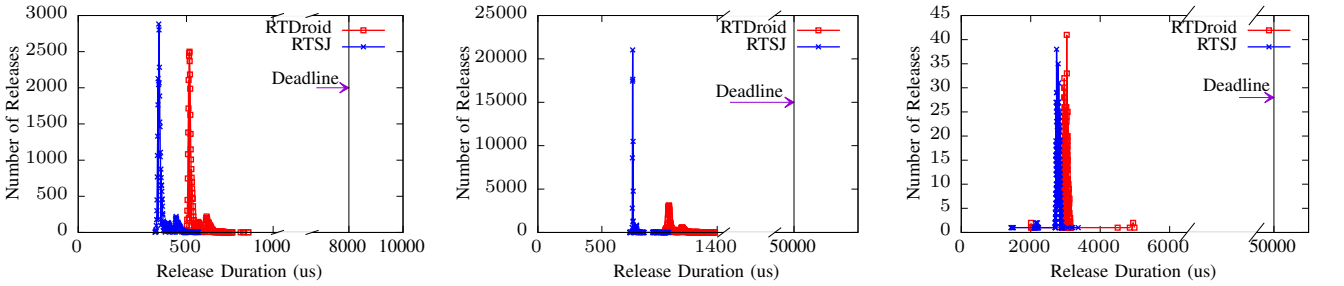
Bulk Data Transfer Channel The bulk data transfer channel shares a number of similarities in implementation with the other channels. To allow zero-copy data transfer between two regions, we modify how scoped memory works to permit ownership transfer. A nested scope, which in this case encapsulates the 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 system does not expose scopes to programmers, the constraints are ensured by the structure of the channel as well as the real-time constructs.

Cross-Context Channel The cross-context channel enables communication between our extended RTDroid runtime and stock Android runtime. Fig. 12 shows how the bi-directional communication is established through sockets. There are two proxy components in each runtime, and they use socket objects to exchange information. To avoid interference, the Android proxy component in our framework is executed in heap memory, and it runs as the lowest priority in real-time. 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, preventing non real-time components from exhausting memory used by real-time constructs. Note that non real-time components can exhaust the heap, but this will not affect real-time components as they leverage pre-allocated memory regions.



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

Fig. 13: Micro-benchmarks for Real-time Communication Channels



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

Fig. 14: RTSJ v.s. RTDroid

V. Evaluation

To evaluate RTDroid we use three application case studies: a cochlear implant application described in Section II, a UAV flight control system, jPapaBench [5] , and a turbine health monitoring application. We use these case studies to compare against Android as well as RTSJ. All results are collected on a Raspberry Pi Model B, which has a single-core ARMv6-based CPU with 512 MB RAM, and runs Debian with Linux preemptive kernel v3.18 and on Google Nexus 5 smartphone, which has a quad-core 2.3 GHz Krait 400 Processor and 2GB RAM, running Android v6.0.1. On both platforms we only enable one core and fix CPU frequency. For the turbine health monitoring application, we use an external audio codec [2] in order to provide high-quality audio playback and capture for vibro-acoustic analysis.

A. Channel Micro Benchmarks

To demonstrate that our channels provide real-time guarantees, we use a micro benchmark that runs two real-time services and one non real-time service. The real-time services act as a sender that sends a message every 100 ms with the highest priority and 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. We use three types of noise-making threads: (1) heap noise that allocates an array of 512 KB in the heap memory every 200 ms, (2) computational noise that computes π every 200 ms, and (3) message noise that sends a low-priority message to the receiving service every 200 ms.

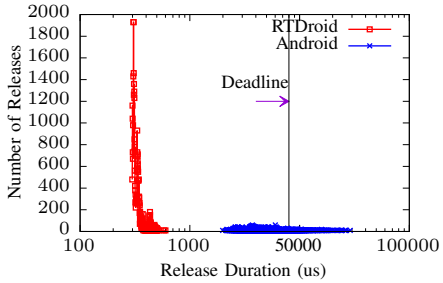
Fig. 13 shows CDF plots comparing the performance of all three types of channels. For basic messaging, shown in Fig. 13a, our implementation effectively provides an unchanged overall latency profile, regardless of the types of background load. We observe in Fig. 13b similar performance characteristics for our `Intent` broadcast channel, though we do notice additional overhead as compared to the message passing channel. This is to be expected as the `Intent` broadcast channel results in the creation of a callback, which adds a fixed amount of overhead. Fig. 13c shows the CDF comparing the transfer latencies with different sizes of data payload for the bulk data transfer channel. The transfer latency is the delivery time of an `Intent` with a bulk data payload. Instead of stressing the system with noise-making threads, we increase the size of data payloads to demonstrate the performance of our *zero-copy* data transfer.

B. Comparison between RTSJ, Android, and RTDroid

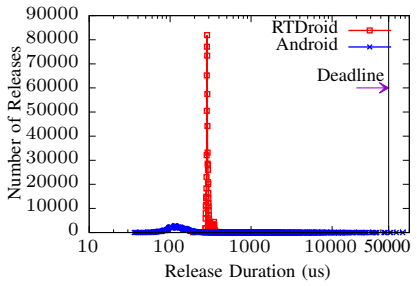
Cochlear Implant: The cochlear implant application has a real-time service for audio processing and a real-time receiver for output error checking. Each run of the audio processing needs to acquire 128 audio samples, process them, and send processed audio output to the output receiver. This process should complete within 8 ms [3], [1]. Our main measurement and comparison point is this audio processing task since it has a strict timing requirement. We collected 40,000 audio

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	740	360	3,000	2,779
Standard Deviation (μs)	16	15	2,831	55	43	1,530	107	103
Deadlines Missed	0	0	5,160	0	0	14	0	0

TABLE I: RTDroid v.s. RTSJ in Task Execution Duration



(a) Cochlear Implant: Audio Processing Duration



(b) jPapaBench: Stabilization Task Duration

Fig. 15: Android v.s. RTDroid

processing task release durations for each execution, and repeat the total experiment 10 times.

jPapaBench: jPapaBench is a real-time Java benchmark that simulates autonomous flight control. We have ported it to RTDroid and Android. Our ports have divided the jPapaBench code into two services: (1) an autopilot service that contains simulated sensing tasks and UAV controlling tasks, (2) a fly-by-wire (FWB) service that handles radio commands and performs safety checking. The original communication is replaced with `Intent` broadcasts. The task we monitor for comparison is the autopilot stabilization task, which runs periodically and needs to finish within 50 ms. As before we measure release durations and repeat the experiment 10 times. Due to the physical simulation variation, each execution takes roughly 9180 releases to complete the same flight path.

Wind Turbine Health Monitoring: The wind turbine health monitoring application was written using a subset of RTSJ and also RTDroid. 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 [15]. It consists of an probing task that imposes a clean sine-wave audio tone at one side of a blade, a recording task that stores the captured audio from the other end of the blade, and an analyzing task that detects cracks by analyzing the stored audio stream. The audio recording task *must* be executed every 50 ms in order to capture meaningful data, and as such is our main point of measurement. We collected release durations of the audio recording task over 2 hours, and only kept releases that perform recording logic.

Fig. 14 shows aggregated task execution durations over each application, and plots the frequency of the execution duration. These results show that the use of scoped memory as well as performing communication over channels does increase the execution duration, but we do not observe deadline missed release during any experiments. Android, not surprisingly, is not very predictable. Fig. 15 shows that there is extreme variance in the duration of the releases. To quantify the overhead, we report the statistical results of each application in Table I. Both RTDroid and RTSJ have similar standard deviations, but RTDroid does induce an overhead from the use of scoped memory and channel based communication. This is particularly visible in the stabilization task of jPapaBench, which reads from shared memory and performs at tight numeric computation. The RTDroid version leverages scoped memory and receives data over channels. However, even with this overhead no deadlines are missed.

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 SLoCCount.
[b]Methods or blocks protected by synchronized statements
[c]Lines of XML cod

TABLE II: Code Complexity for Cochlear Implant

Although RTDroid does induce additional overhead when compared to applications written in RTSJ, it does provide tangible benefits in terms programability. Table II shows code metrics over three types of code—the common code in both versions of implementation (mostly the application logic), RTDroid specific code, and RTSJ specific code, but excludes common libraries (*i.e.* the FFT and signal processing libraries

for the cochlear implant). It shows that RTDroid applications are implemented with fewer lines of code. This occurs because RTSJ requires developers to manually instantiate all tasks, and provide release logic with the multi-threading APIs. In RT-Droid all application components are declared in the manifest and the boot process initiates and starts them. Additionally, since our system provides communication channels as APIs, it removes certain synchronization concerns from the application logic.

VI. Related Work

Previous attempts to make Android amenable to real-time include the work of Maia *et al.* who proposed four different architectures [11], [16], [17], [19], [12] that enforce a strict separation between real-time and non real-time apps. Our work strives to make such interactions safe. Kalkov *et al.* [6] proposed to explicitly trigger the GC to reduce pause time during critical periods. Our work avoids this as choosing when to run the GC is difficult. They also explored how components interact through `Intent` messages, and re-designed it to provide priority awareness [7], but not memory predictability, limiting the solution to soft real-time apps. RTDroid [22] explored how to add priorities to three exemplar constructs in Android: `Looper` and `Handler`, the `Alarm Manager`, and `Sensor Manager`. We adopt the priority mechanisms defined by RTDroid and observe that they are not enough to correctly encode a priority aware `Intent`.

Our work leverages previous results on region-based memory management [21]. Scope memory was introduced in the RTSJ [4] to avoid GC interference. Scope memory allows the system designer to prove properties about the predictability of the overall system including static memory bounds [20]. 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.

VII. Conclusion

Real-time capabilities have the potential of increasing the range of applications that can be written on the Android platform. This paper is a step towards turning Android into a high-level real-time programming environment in which developers can freely mix time-critical code with code that is blissfully unaware of any timing constraints. In this paper we have shown that the changes required to the Android programming model from the programmers perspective are quite modest. Our constructs, which expose familiar Android interfaces, additionally provide statically specified memory bounds and priority awareness.

References

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

[2] Wolfson Audio Card for Raspberry Pi Model B. http://www.element14.com/community/community/raspberry-pi/raspberry-pi-accessories/wolfson_pi. Accessed March 25, 2015.

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

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

[5] Tomas Kalibera, Pavel Parizek, Michal Malohlava, and Martin Schoeberl. Exhaustive Testing of Safety Critical Java. In *Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 164–174, 2010.

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

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

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

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

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

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

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

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

[14] Muhammad Mubashir, Ling Shao, and Luke Seed. A Survey on Fall Detection: Principles and Approaches. *Neurocomput.*, 100:144–152, January 2013.

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

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

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

[18] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-Level Programming of Embedded Hard Real-Time Devices. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 69–82, New York, NY, USA, 2010. ACM.

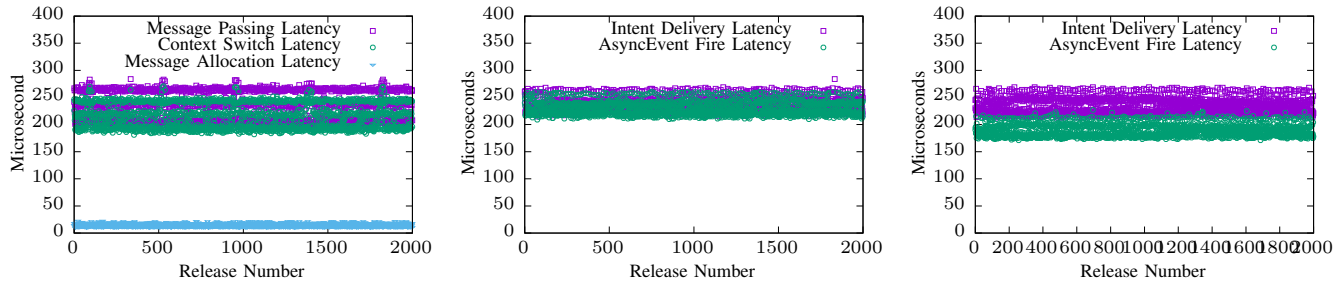
[19] Ganesh Jairam Rajguru. Reliable Real-Time Applications on Android OS. *International Journal of Management, IT and Engineering*, 4(6):192, 2014.

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

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

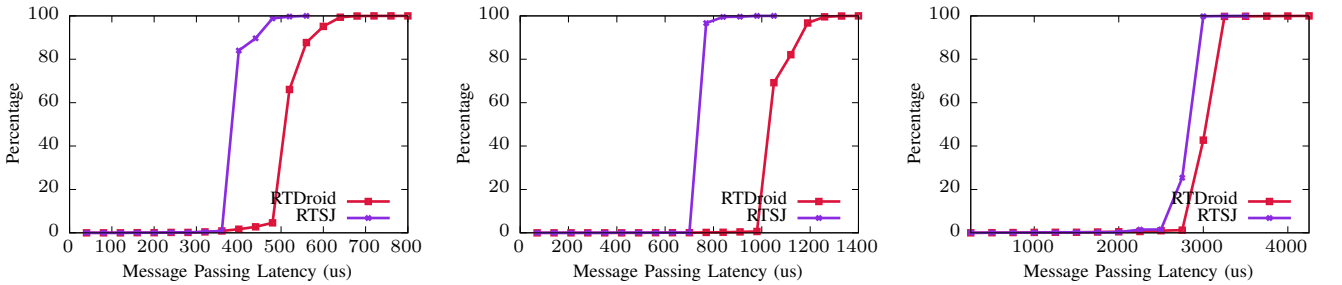
[22] Yin Yan, Sree Harsha Konduri, Amit Kulkarni, Varun Anand, Steve Ko, and Lukasz Ziarek. Real-Time Android with RTDroid. In *The 12th International Conference on Mobile Systems, Applications, and Services*, MOBISYS '14, New York, NY, USA, 2014. ACM.

APPENDIX



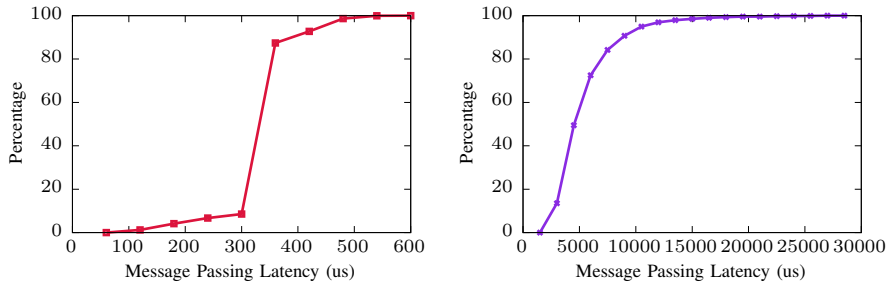
(a) Message Passing Channel (b) Intent Broadcasting Channel (c) Bulk Data Transfer Channel

Fig. 16: Real-time Communication Channels: Baseline Scatter Plot for Micro-benchmarks

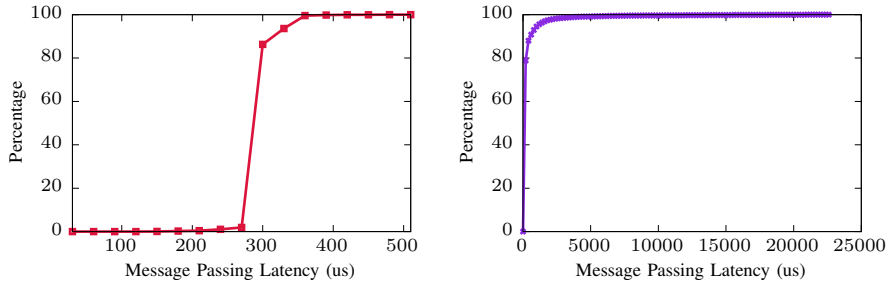


(a) Cochlear Implant: Audio Processing Duration (b) jPapaBench: Stabilization Task Duration (c) Turbine Monitoring: Audio Recording Duratio

Fig. 17: RTDroid v.s. RTSJ in CDF



(a) Cochlear Implant: Audio Processing Duration with RTDroid (b) Cochlear Implant: Audio Processing Duration with Android



(c) jPapaBench: Stabilization Task Duration with RTDroid (d) jPapaBench: Stabilization Task Duration with Android

Fig. 18: Android v.s. RTDroid in CDF