Making Android Run on Time

Yin Yan, University at Buffalo, yinyan@buffalo.edu
Karthik Dantu, University at Buffalo, kdantu@buffalo.edu
Steven Y. Ko, University at Buffalo, stevko@buffalo.edu
Jan Vitek, Northeastern University, j.vitek@northeastern.edu
Lukasz Ziarek, University at Buffalo, lziarek@buffalo.edu

***Abstract*—Time predictability is difficult to achieve in the complex, layered execution environments that are common in modern embedded devices. We consider the possibility of 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 report on the design and implementation of an Android virtual machine with soft-real-time support, and provides experimental data validating feasibility over three case studies.**

I. 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 of smartphones and tablets, but also holds for the Internet of Things and 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 paper investigates the use of 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 programming model and to the libraries. The choice of Android is motivated by its popularity on mobile devices, but 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 health care (*e.g.* a cochlear implant) [2], audio-based indoor localization [12], harmonized sound reproduction [11], timely sound delivery [10], and UAV flight control [3]. We have also deployed our system on a wind turbine for blade health monitoring [15].

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 previous research efforts which included the design and implementation of the Fiji real-time Java virtual machine [19] and our first generation RTDroid system [24], [25] that focused on enhancing integrating the Fiji VM virtual machine and Android system services with support for the Real-Time Specification for Java (RTSJ) [5] on top of a real-time operating system.

978-1-5090-5269-1/17/$31.00 ©2017 IEEE

Our contributions focus on the programming model exposed to developers. To retain a familiar style of development, we make a number of changes to the Android abstractions and how they interact with the underlying system as well as each other. We aim to leave legacy Android code unaffected, and only expose real-time features to components which have timeliness requirements. The changes required to the Android platform 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 Android 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.

To validate our design and evaluate the quality of our implementation, we report on three applications deployed on RTDroid on commodity embedded boards as well as smartphones. They are a cochlear implant, a wind turbine health monitor, and a UAV flight control benchmark. Our results illustrate that, at least in these three use-cases, the modified platform delivers significantly better time predictability than stock Android and reduces the code complexity as compared to the RTSJ. Readers interested in additional use-cases are directed to [4] which includes applications for audio-based indoor localization, harmonized sound reproduction, and timely sound delivery.

While our experience using Android for embedded tasks has been mostly positive, we should mention some limitations. Performance has not been a problem for our target applications, but clearly using a high-level language can come at a cost. 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 degraded performance. Lastly, unlike the RTSJ which was designed with great care to cover many different real-time programming styles, our approach is demand-driven and makes no claims of generality.

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

Fig. 1: Audio Configuration UI written in Android.

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

Fig. 2: Audio Processing Service written in Android.

II. Android-enabled Real-time Applications

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

Even with theses limitations, the health care industry has been studying how to adapt Android for wearable and implantable health devices, like *cochlear implants*. A cochlear implant restores hearing abilities through an electronic device surgically inserted in the inner ear. It relies on external components to capture ambient audio, convert it into digital signals, and translate the signals into electrical energy. There is interest in leveraging smartphones [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 8 ms.

A plausible design for such an application would be to split the user interface that controls volume and noise reduction from sound processing. The UI can be implemented as an activity as shown in Figure 1. It deals 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 will not be delayed by UI processing. When components have to interact through Android-based communication mechanisms, ensuring non-interference becomes even more tricky.

Figure 3 shows the architecture of our solution in RT-Droid. It separates real-time (`RecordingService`, `ProcessingService`, and `OutputReceiver`) and non-real-time components (`VolumeReceiver` and `ConfigurationUI`). The former have priorities attached and use communication services that prioritize messages. `ConfigurationUI` has a `Handler` for other components to update the UI, and a non-real-time receiver listens on volume key events. It also receives messages from real-time components. Similarly, the `ProcessingService` receives messages from non real-time components (`VolumeReceiver` and `ButtonListener`) and a real-tine component (`RecordingService`). RTDroid allows these components to communicate while enforcing memory bounds. Each real-time component is provided a fixed amount of memory for its exclusive use. That memory is divided into two section, one persists for the lifetime of the component, the other is cleared each time the component yields control. Messages are pre-allocated. Non real-time components allocate messages in heap memory. RTDroid extends the Android manifest to enable developers to declare properties of components that include priority, periodicity and memory bounds.

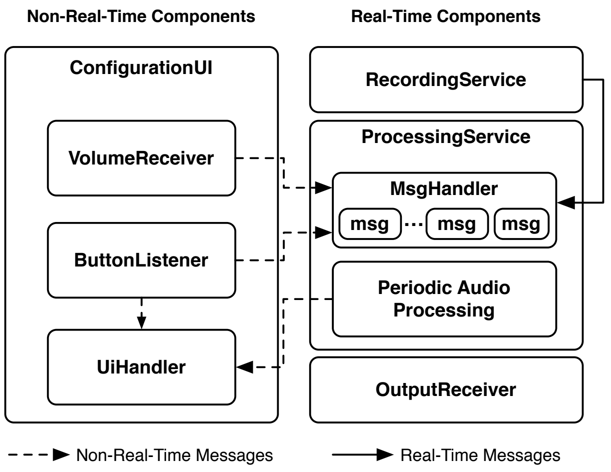


Fig. 3: Architecture of the Cochlear Implant Application.

III. 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 [24] integrated a real-time JVM and a RTOS with

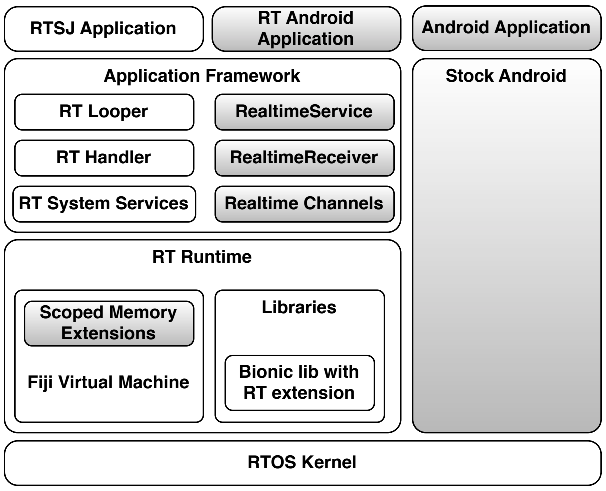


Fig. 4: RTDroid runtime architecture. Gray elements are the extensions introduced in this paper. Notably, we now support multiple, interacting real-time applications, real-time applications written using an Android like programming model, as well as legacy code and stock Android applications.

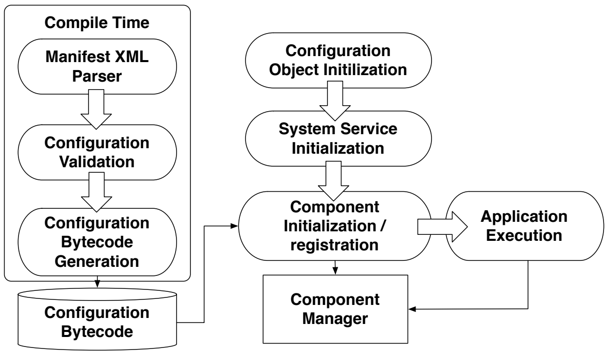

Compile Time
Manifest XML Parser
Configuration Validation
Configuration Bytecode Generation
Configuration Bytecode
Configuration Object Initilization
System Service Initialization
Component Initialization / registration
Application Execution
Component Manager

Fig. 5: Bootstrap sequence.

the Android framework, as well as re-designed some system services for real-time support. It relied on real-time garbage collection and was limited to running a single real-time application. The present 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 5. 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 VM, this is well suited to run user interface components and applications that have no timeliness requirements.

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 runtime, shown in Figure 5. The two stage process ensures that memory can be pre-allocated and components are correctly configured. During compile time, our framework parses the manifest file of an application, runs verification checks (described in more detail in Section sec:rt-manifest, and emits configuration bytecode for all components. 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.

A. Components

RTDroid supports three different real-time components: services, tasks and receivers. A `RealtimeService` is a counterpart to Android's service used for one-shot aperiodic or sporadic computation. As the notion of periodic computation is foreign to Android, we introduce the `PeriodicTask` class to model such behavior. Tasks are used internally within a real-time service. A `RealtimeReceiver` is used to react to system-wide events delivered via intents. We do not provide a counterpart to Android's activities; they are used for UI programming and we have not yet observed real-time requirements for these. We do allow for 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`, etc.). The association between a periodic task and its parent is also specified in the manifest by a `periodic-task` tag. The manifest provides information for boot-time verification and pre-allocation of components. RTDroid ensures that the total memory requested specified for a component equals the objects in its persistent memory, its per-release memory, and that of its sub-components. Figure 6 shows a manifest for the processing service of our running example.

Managing the lifetime of components requires: (1) ensuring

```
<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.

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

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

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

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 in order to allow services to run with different priorities.

Services are bound to real-time threads from the underlying real-time JVM. By default, when a service is initialized, it is

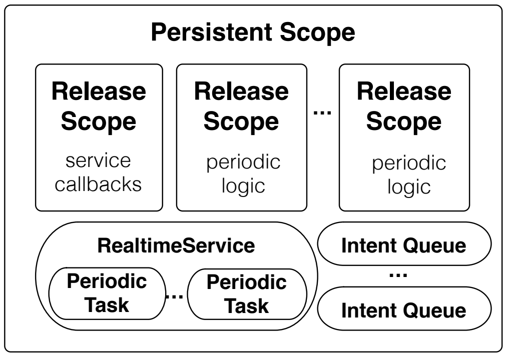


Fig. 7: Scope Structure for a Service.

```
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. */
    registerPeriodicTask("processingTask",
         task);
  }
}
```

Fig. 8: Real-Time Service and Periodic Task.

assigned a persistent memory scope that has the same lifetime as the service. The scope is allocated when the service starts and deallocated when the service terminates. Static initializers for the service are run in this scope. In addition, if the service uses communication channels, intent queues are allocated in persistent memory. Callbacks execute within the scope of release memory for the associated service. The release memory is cleared when the callback returns. Similarly, when a periodic task is started in a service, it is also assigned a release scope. Note that our manifest requires specification of memory bounds for callbacks and periodic tasks, and this information is used to size release scopes appropriately. Figure 7 depicts the scope structure for a service as well as pre-allocated objects during boot.

2) Periodic Task: A periodic task is a sub-component of a service. In addition to the characteristics of its parent service, a task needs a period. Figure 8 shows an example which processes audio input periodically. A programmer needs only to implement `onRelease()` to specify a periodic computation.

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

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

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

B. Communication

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

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

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

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

Our implementation creates a fixed-length message queue for each channel. Along with the message queue, message objects are also pre-allocated. They live in persistent memory of the receiver. Figure 10 illustrates the scope memory hierarchy of our design.

```
<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.

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

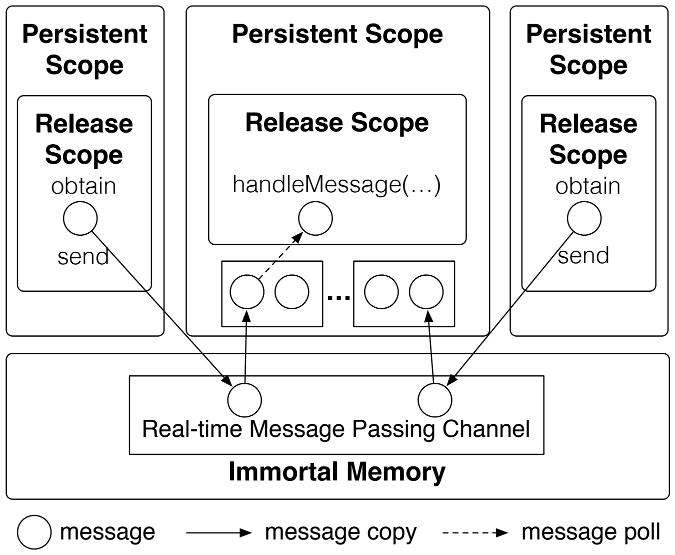


Fig. 10: Real-time Message Passing Channel.

Sending messages is slightly tricky as they are pre-allocated and senders should not retain references messages after the message has been sent. The protocol for sending a message is thus indirect. A `MessageClosure` is allocated by the sender, and the `genMsg()` callback is used to populate the message's payload with data. This unifies message population and queuing and is shown in Figure 11. In our running example, high-priority messages from `RecordingService` can be prioritized over the messages from non-real-time `ConfigurationUI`.

The message is served based on sender priority as a message pool. As a direct consequence, the obtain operation can fail when no messages are available as they have been given to higher priority component. An exception is raised in this case. If a high priority component attempts to obtain a message and all message objects are currently in use, the high priority thread can steal message objects that are currently being used

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

Fig. 11: Message Passing Interface.

by low priority and non real-time senders. Since messages are obtained during the send method of `MessageClosure`, all message objects in use will correspond to messages that have been enqueued, but not yet received. If the message is stolen from a construct, an asynchronous exception is delivered to the construct by utilizing the RTSJ `AsynchronousInterruptedException` mechanism.

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

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

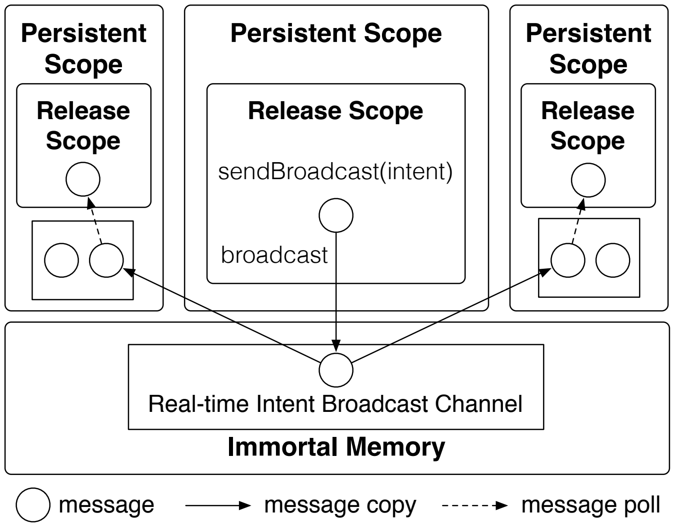


Fig. 12: Real-time Intent Broadcast Channel.

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

4) Cross-Context Channels: Cross-context channels allow Android's activities to communicate with real-time components. In this scenario communication is occurring between two separate VMs, one of which is executing the non real-time application and 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 an real-time application that uses our real-time constructs. For communication in the other direction, a real-time application need only to subscribe to intents the non real-time application has declared in its manifest. Since our manifest is an extension of the Android manifest, no changes are require to the configuration of Android. The proxy service allows non-real-time code to send an intent to real-time components. Communication in the other direction requires the activity can subscribe to intents defined by real-time code. To preserve memory bounds, the number of intents in a cross-context channel is bounded and each intent has a fixed-length payload. Figure 13 shows how the bi-directional communication is established through sockets between RTDroid and Android. To do so, we leverage two proxy components in each runtime, To avoid interference, the Android proxy component is executed in heap memory, and it runs at the lowest real-time priority. The incoming message objects are translated to real-time 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. Non-real-time components can exhaust the heap, but this will not affect real-time components using pre-allocated memory regions.

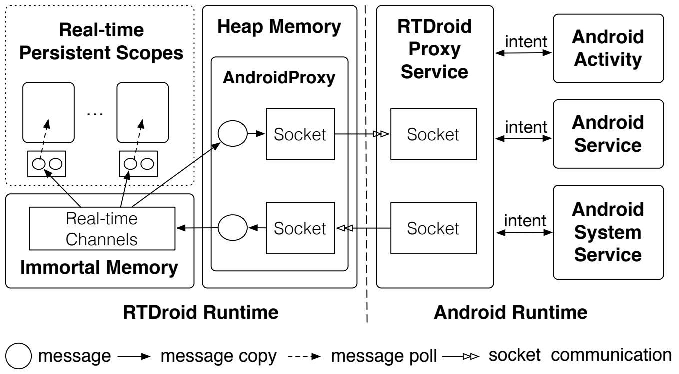


Fig. 13: Cross-Context Channel.

C. Memory Management

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

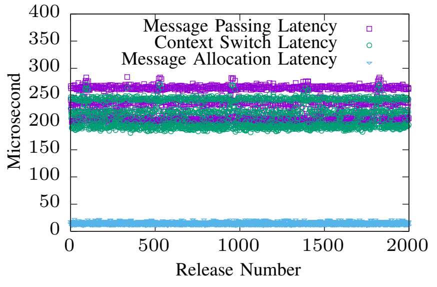
IV. Evaluation

To evaluate our system we use three application case studies: a cochlear implant application described in Section II, a UAV flight control system, jPapaBench [6], 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 a Google Nexus 5 smartphone, which has a quad-core 2.3 GHz Krait 400 Processor and 2GB RAM, running Android v6.0.1. On both platforms we only enable one core and fix CPU frequency. For the turbine health monitoring application, we use an external Wolfson audio codec in order to provide high-quality audio playback and capture for vibro-acoustic analysis. Raw data and plotting scripts can be found under the publications tab and cases study code under the application tab on our website: rtdroid.cse.buffalo.edu.

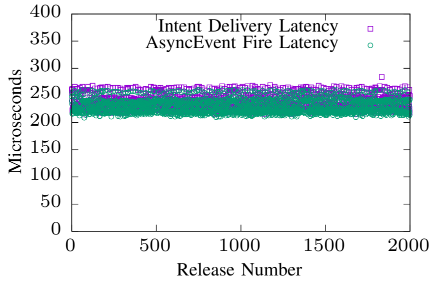
A. Channel Micro Benchmarks

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. 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. 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.

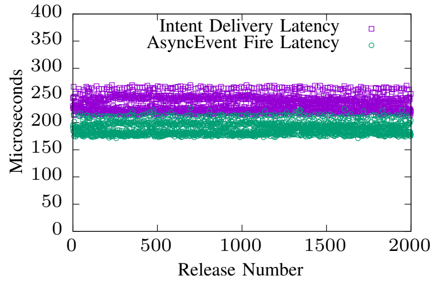
Figure 14 shows raw performance measurements for the baseline performance of our channel implementations. Message passing consists of message allocation by the sender, message delivery, and context switch from the sender to the receiver. Figure 14a shows this breakdown with just the sender and the receiver, and it is our baseline performance. In the figure, we plot the latency of 2000 message passing events.



(a) Message Passing

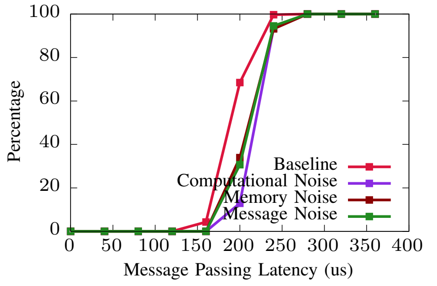


(b) Intent Broadcast

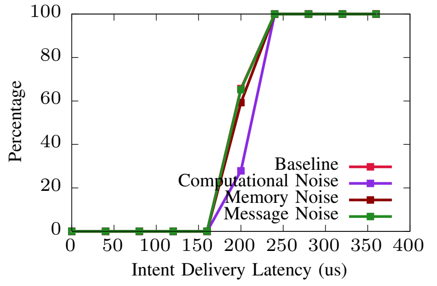


(c) Bulk Data Transfer

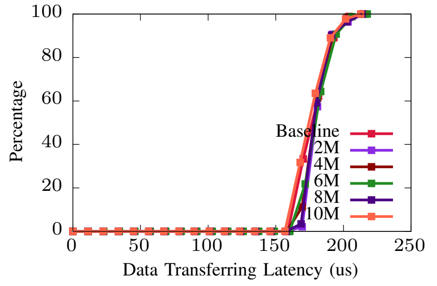
Fig. 14: Real-time Communication Channels: Baseline Scatter Plot for Micro benchmarks.



(a) CDF for Message Passing



(b) CDF for Intent Broadcast



(c) CDF for Bulk Data Transfer

Fig. 15: Micro-benchmarks for Real-time Communication Channels.

For each event, the message allocation latency is the amount of time it takes for a sender to instantiate a message. The message passing latency is the time taken for delivery. The context switch latency is the difference between the time the sender sends a message and before the receiver processes the message. As shown, all three types of latency are tightly bounded across all events, and there is no outlier that takes much more time to process than others. It shows that without any other background load, our implementation provides stable and predictable performance.

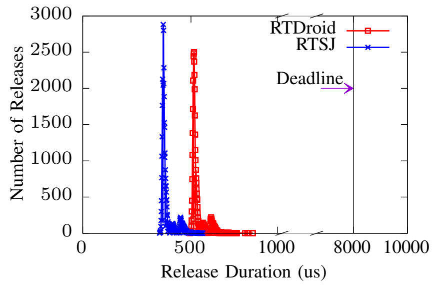
We have conducted a similar experiment to evaluate our `Intent` delivery channel. The experimental configuration is the same except that we use our `Intent` broadcast channel instead of message passing; the sender sends an `Intent` every 100 ms, and the receiver executes a dummy callback that responds to the `Intent`. The `Intent` delivery latency is the overall latency for each `Intent` event, and the callback trigger latency is the amount of time it takes to spawn a new callback. Figure 14b shows the baseline performance. Similarly Figure 14c shows the baseline performance for bulk data transfer, which also leverages the `Intent` mechanism for delivery, but is specialized to use the bulk data transfer channel.

Figure 15 shows cumulative distribution function (CDF) plots comparing the performance of all three types of channels. The CDF illustrates what percent of the total measured points are equal to or less than a given time value. For basic messaging, shown in Figure 15a, our implementation effectively provides an unchanged overall latency profile, regardless of the types of background load. We observe in Figure 15b similar performance characteristics for our intent broadcast channel, though we do notice additional overhead as compared to the message passing channel. This is to be expected as the intent broadcast channel results in the creation of a callback, which adds a fixed amount of overhead. Figure 15c 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.

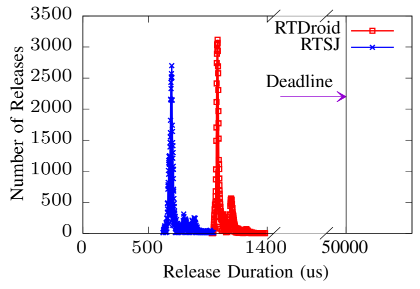
B. Comparison with RTSJ and Android

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.

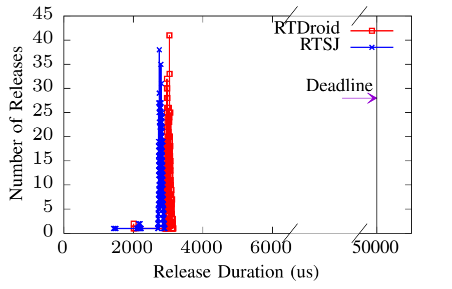
1) Cochlear Implant: The cochlear implant application has a real-time service for audio processing and a real-time receiver for output error checking. Each run of the audio processing needs to acquire 128 audio samples, process them, and send processed audio output to the output receiver. This process should complete within 8 ms [2], [1]. Our main measurement and comparison point is this audio processing task since it has a strict timing requirement. We collected



(a) Cochlear Implant: Audio Processing Duration

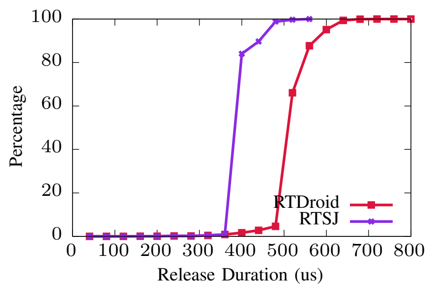


(b) jPapaBench: Stabilization Task Duration

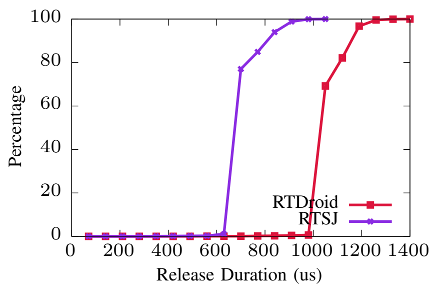


(c) Turbine Monitoring: Audio Recording Duration

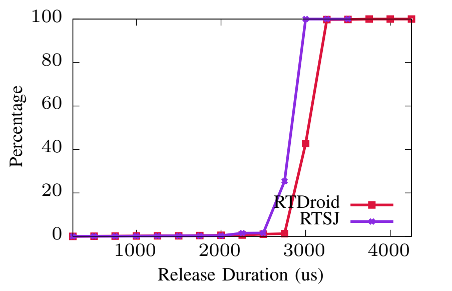
Fig. 16: Performance Measurements on Raspberry Pi.



(a) Cochlear Implant: Audio Processing Duration



(b) jPapaBench: Stabilization Task Duration



(c) Turbine Monitoring: Audio Recording Duration

Fig. 17: CDFs of Performance Measurements on Raspberry Pi.

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

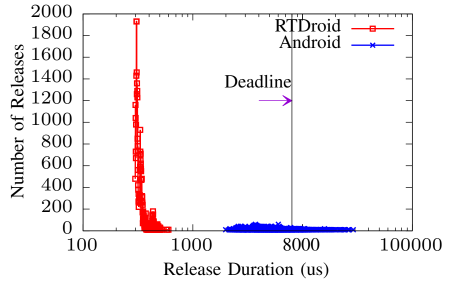
TABLE I: Task Execution Duration Statistics.

40,000 release durations for each execution, and repeat the experiment 10 times.

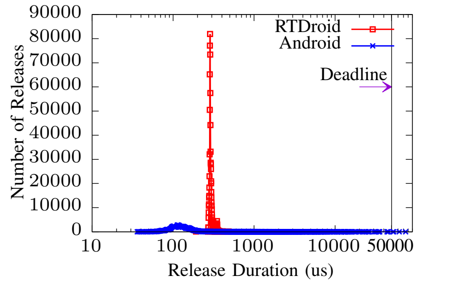
2) jPapaBench: jPapaBench is a real-time Java benchmark that simulates autonomous flight control. We have ported it to our system as well as Android and divided the code into two services: (1) an autopilot service that executes sensing, stabilization, and control tasks, (2) a fly-by-wire (FWB) service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

3) Wind Turbine Health Monitoring: 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 [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. The size of the audio buffer recorded per release is around 2MB and as such we leverage our bulk data transfer channel for communication between the recording and audio processing tasks for the version implemented in our system. The RTSJ version uses a shared memory buffer.

Figure 16 shows aggregated task execution durations over each application, and plots the frequency of the execution duration for each release. These results show that the use of scoped memory as well as performing communication over channels does increase the execution duration for each release, but this overhead is bounded. Android, not surprisingly, is

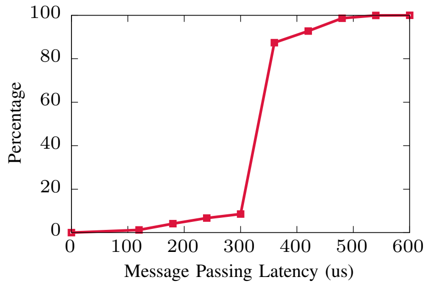


(a) Cochlear Implant: Audio Processing Duration

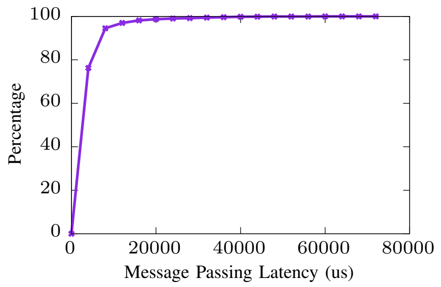


(b) jPapaBench: Stabilization Task Duration

Fig. 18: Performance Measurements on Nexus 5.



(a) Audio Processing Duration with RTDroid



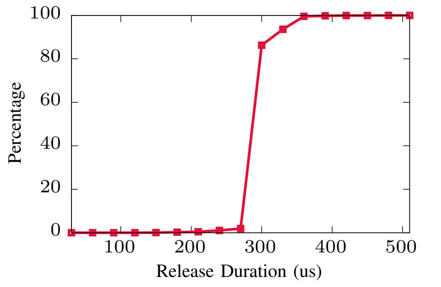
(b) Audio Processing Duration with Android

Fig. 19: CDFs of Performance Measurements of the Cochlear Implant on Nexus 5.

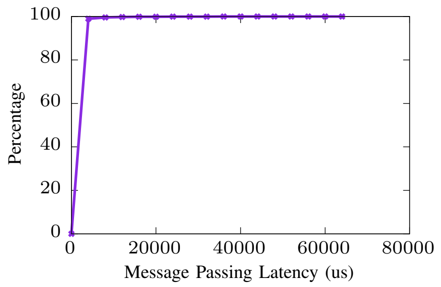
not very predictable. Figure 18 shows that there is extreme variance in the duration for each release. To quantify the overhead imposed by our system, we report the statistical results of each application in Table I. Both our system and RTSJ have similar standard deviations even in the presence of scoped memory and channel based communication. Our system's overhead is particularly visible in the stabilization task of jPapaBench. In the RTSJ version, the stabilization tasks reads sensor data from global shared memory buffers, performs at tight numeric computation, and produces control commands for the motors, which are also stored in global shared memory buffers. The version executing on our system, in comparison, receives sensor readings and sends control commands over channels, instead of reading from global buffers.

Figure 17 show the CDFs of the experiments detailed in Figure 16. We can observe that the curves of the CDFs for RTSJ performance compared to RTDroid performance are similar. Based on this observation as well as similar standard deviations presented in Table I, we can conclude that RTDroid does introduce additional latency, but does not impact the predictability of the code as compared to RTSJ. Figure 19 and Figure 20 show the CDFs of the experiments detailed in Figure 18. Although not surprising, our numbers indicate that a non trivial portion of releases in Android exhibit significant delays, even when not in the presence of a loaded system.

Although our system does induce additional overhead when compared to applications written in RTSJ, it does provide tangible benefits in terms programability. In addition to hiding the complexity of writing code that leverages scoped memory, our system also decouples configuration from application logic and



(a) Stabilization Task Duration with RTDroid



(b) Stabilization Task Duration with Android

Fig. 20: CDFs of Performance Measurements of the jPapaBench Stabilization Task on Nexus 5.

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 Measurements.

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

V. Related Work

Previous attempts to make Android amenable to real-time include the work of Maia *et al.* who proposed four different architectures [13], [16], [18], [20], [14] that enforce a strict separation between real-time and non real-time apps. Kang *et al.* [9] and Ruiz *et al.* [21] 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. Kalkov *et al.* [7] 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 intents, providing a mechanism to prioritize intents [8], 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 observers that only prioritizing intents can induce priority inversion in the callbacks that handle those intents. Other efforts have the left the Android framework unmodified [17], 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 RTOS, RTDroid [24] 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 nor to 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 [23]. Scope memory was introduced in the RTSJ [5] to avoid GC interference. Scope memory allows the system designer to prove properties about the predictability of the overall system including static memory bounds [22]. 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.

VI. 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.

Acknowledgments. This work was partially supported by NSF Grants 1523426, 1513006, 1405614 and 1544542.

References

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

[2] Hamza Ali, Arthur Lobo, and Philipos Loizou. Design and Evaluation of A Personal Digital Assistant-Based Research Platform for Cochlear Implants. *IEEE Transactions on Biomedical Engineering*, 60(11):3060–3073, 2013. doi:10.1109/TBME.2013.2262712

[3] Adam Czerniejewski, Shaun Cosgrove, Yin Yan, Karthik Dantu, Steven Ko, and Lukasz Ziarek. jUAV: A Java Based System for Unmanned Aerial Vehicles. In *International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES, 2016. doi:10.1145/2990509.2990511

[4] Girish Gokul, Yin Yan, Karthik Dantu, Steven Ko, and Lukasz Ziarek. Real Time Sound Processing on Android. In *International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES, 2016. doi:10.1145/2990509.2990512

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

[6] 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, 2010. doi:10.1145/1850771.1850794

[7] Igor Kalkov, Dominik Franke, John Schommer, and Stefan Kowalewski. A Real-Time Extension to The Android Platform. In *International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES, 2012. doi:10.1145/2388936.2388955

[8] Igor Kalkov, Alexandru Gurghian, and Stefan Kowalewski. Predictable Broadcasting of Parallel Intents in Real-Time Android. In *International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES, 2014. doi:10.1145/2661020.2661023

[9] Hyeongseok Kang, Dohyeon Kim, Jeongnam Kang, and Kanghee Kim. Real-Time Motion Control on Android platform. *The Journal of Supercomputing*, 72(1):196–213, 2016. doi:10.1007/s11227-015-1542-5

[10] 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*, RTSS, 2015. doi:10.1109/RTSS.2015.34

[11] 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 *International Conference on Pervasive and Ubiquitous Computing*, UbiComp, 2014. doi:10.1145/2632048.2636077

[12] 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. doi:10.1145/2462456.2464450

[13] Cláudio Maia, Luís Nogueira, and Luis Miguel Pinho. Evaluating Android OS for Embedded Real-Time Systems. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, OSPERT, 2010.

[14] 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, 2012.

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

[16] Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi, and Soo-Mook Moon. Evaluation of Android Dalvik Virtual Machine. In *International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES, 2012. doi:10.1145/2388936.2388956

[17] E. Peguero, M. Labrador, and B. Cook. Assessing Jitter in Sensor Time Series From Android Mobile Devices. In *International Conference on Smart Computing*, SMARTCOMP, 2016. doi:10.1109/SMARTCOMP.2016.7501679

[18] Luc Perneel, Hasan Fayyad-Kazan, and Martin Timmerman. Can Android Be Used for Real-Time Purposes? In *Computer Systems and Industrial Informatics* ICCSII, 2012. doi:10.1109/ICCSII.2012.6454350

[19] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-Level Programming of Embedded Hard Real-Time Devices. In *European conference on Computer Systems*, EuroSys, 2010. doi:10.1145/1755913.1755922

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

[21] Alejandro Pérez Ruiz, Mario Aldea Rivas, and Michael González Harbour. CPU Isolation on the Android OS for Running Real-Time Applications. In *Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES, 2015. doi:10.1145/2822304.2822317

[22] Daniel Tang, Ales Plsek, and Jan Vitek. Static Checking of Safety Critical Java Annotations. In *International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES, 2010. doi:10.1145/1850771.1850792

[23] Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-value λ-calculus Using a Stack of Regions. In *Symposium on Principles of Programming Languages*, POPL, 1994. doi:10.1145/174675.177855

[24] Yin Yan, Sree Harsha Konduri, Amit Kulkarni, Varun Anand, Steve Ko, and Lukasz Ziarek. Real-Time Android with RTDroid. In *International Conference on Mobile Systems, Applications, and Services*, MOBISYS, 2014. doi:10.1145/2594368.2594381

[25] Yin Yan, Shaun Cosgrove, Varun Anand, Amit Kulkarni, Sree Harsha Konduri, Steve Ko, and Lukasz Ziarek. RTDroid: A Design for Real-Time Android. In *IEEE Transactions on Mobile Computing*, 15(10):2564–2584, 2016. doi:10.1109/TMC.2015.2499187