SounDroid: Supporting Real-Time Sound Applications on Commodity Mobile Devices

Hyosu Kim†, SangJeong Lee‡, Wookhyun Han†, Daehyeok Kim†, Insik Shin*
†*School of Computing, KAIST, South Korea
‡Frontier CS Lab., Software R&D Center, Samsung Electronics, South Korea
hyosu.kim@kaist.ac.kr, sj94.lee@samsung.com, insik.shin@cs.kaist.ac.kr

***Abstract*—A variety of advantages from sounds such as measurement and accessibility introduces a new opportunity for mobile applications to offer broad types of interesting, valuable functionalities, supporting a richer user experience. However, in spite of the growing interests on mobile sound applications, few or no works have been done in focusing on managing an audio device effectively. More specifically, their low level of real-time capability for audio resources makes it challenging to satisfy tight timing requirements of mobile sound applications, e.g., a high sensing rate of acoustic sensing applications. To address this problem, this work presents the SounDroid framework, an audio device management framework for real-time audio requests from mobile sound applications. The design of SounDroid is based on the requirement analysis of audio requests as well as an understanding of the audio playback procedure including the audio request scheduling and dispatching on Android. It then incorporates both real-time audio request scheduling algorithms, called EDF-V and AFDS, and dispatching optimization techniques into mobile platforms, and thus improves the quality-of-service of mobile sound applications. Our experimental results with the prototype implementation of SounDroid demonstrate that it is able to enhance scheduling performance for audio requests, compared to traditional mechanisms (by up to 40% improvement), while allowing deterministic dispatching latency.**

I. Introduction

Various real-time applications using audio resources have emerged recently in mobile environments. They not only simply play music or sound effects, but also perform many functions including indoor localization [1], gesture recognition [2], [3], [4], [5], inter-device distance measurement [6], [7], [8], and harmonized sound reproduction [9]. Unlike traditional sound applications, they have quite sophisticated real-time constraints. For example, sensing applications should emit acoustic signals within about tens of milliseconds for acceptable accuracy. Surround sound play in [9] requires extremely tight synchronization in timing, i.e., less than 1ms.

Yet, there is a scarcity of research on supporting emerging mobile sound applications with a focus on handling their real-time characteristics. While mobile sound applications impose various timing requirements, many of such requirements are not properly supported on Android, one of the most popular mobile platforms. There is a recent work, called RTDroid [10], designed to enable fixed-priority scheduling for real-time support on Android. However, the fixed-priority scheduling is not enough to satisfy well the tight individual timing requirements imposed by diverse sound applications.

In this paper, we report the development of a novel real-time audio device management framework, named SounDroid, for Android sound applications. We first conduct a survey on how applications make use of an audio device (i.e., a loudspeaker) to deliver their own services. We then analyze timing requirements associated with the quality-of-service (QoS) that they aim to provide. Key aspects of such timing constraints include the followings: low and bounded playback latency, accurate on-time playbacks, and high sensing rates. With a proper understanding of the Android audio management system, we identify several sources that make it difficult to support such various timing requirements on Android. The current audio device management typically consists of two steps: (1) *scheduling* that selects one of the audio requests waiting in a ready queue and (2) *dispatching* that gives control of the audio device to the selected request. Each step faces several challenges to be addressed as follows.

- **Diverse requirements for audio request scheduling.** Mobile sound applications impose tight timing constraints from various viewpoints. One is from a high degree of predictability. For example, some applications require to play sounds punctually at the appointed time (with an error of a few milliseconds) for various purposes, including synchronized playbacks among multiple devices. Another is from a performance viewpoint. Sensing applications may also require low playback latency to have high sensing rates and/or demand tight deadlines to obtain smaller jitter bounds for improved sensing accuracy. It becomes even more challenging to meet such stringent timing requirements since applications demand to use an audio device in a non-preemptive fashion. In addition, to further increase scheduling performance, it is essential to consider frequency ranges that each application mostly use.
- **Unpredictable and large dispatching latency.** Even though Android has tried to optimize its audio playback procedure, we observe that there still exists unexpected long latency, especially during the audio request dispatching. The dispatching latency is often so large (i.e., a few to a couple of hundreds of milliseconds) that, even for properly scheduled requests, the platform could not emit sounds on time. More importantly, it is almost unpredictable. Thus, it is very difficult to design an efficient scheduling mechanism without optimizing the dispatching latency.

From the above two sources of challenges, SounDroid develops two main solutions accordingly. First, for the scheduling challenges, we design a novel real-time audio scheduling framework that is based on our new modeling of audio playback requests for mobile sound applications. The model is able to specify various requirements of audio requests, such as atomicity, on-time playback, and acoustic frequency requirements. In order to support the tight timing constraints of various audio requests under non-preemptive scheduling, we propose a new scheduling algorithm, Earliest Deadline First with Virtual-scheduling (EDF-V), that is an extension of EDF-

*A corresponding author

1052-8725/15 $31.00 © 2015 IEEE
DOI 10.1109/RTSS.2015.34

based algorithms [11], [12] with a more elaborated support of the distinctive timing constraints of audio requests. Leveraging the characteristic of audio requests in a frequency domain, we also propose Acoustic Frequency Division Scheduling (AFDS). Second, to deal with the audio request dispatching latency, we optimize the overall audio playback procedure significantly on Android. Especially, we understand and tackle the sources of unpredictable long dispatching latency, which have not been explored sufficiently so far for tight real-time support.

We have implemented the prototype of SounDroid as an extension to Android. We evaluate the real-time capability of SounDroid for mobile sound applications using both simulation and experiments with Android devices. Our evaluation results show that, compared to existing scheduling algorithms and mobile platforms, SounDroid offers much improved scheduling performance for audio requests, lowering and stabilizing the dispatching latency. We also measure how SounDroid affects performance of real-world mobile sound applications such as surround sound reproduction and sensing applications. As a result, for each application, SounDroid allows a high degree of QoS by meeting its requirements (1ms of synchronization accuracy for immersive sounds and 33Hz of measurement rate for sensing), even when multiple sound applications run on SounDroid simultaneously.

Contribution. The main contribution of this paper can be summarized as follows.

- To the best of our knowledge, this work makes the first attempt to explore the real-time issues of currently-emerging mobile sound applications. It then designs a novel mobile platform architecture, SounDroid, which manages an audio device with a high degree of real-time capability, while addressing challenges on the audio playback procedure.
- It introduces the EDF-V and AFDS scheduling algorithms and dispatching latency optimization techniques for real-time support on an audio device. Our experimental results show that EDF-V outperforms the traditional schemes (i.e, CEDF [12]) in terms of schedulability with a marginal increase on average in computation overhead. SounDroid also provides much more deterministic dispatching latency with an acceptable amount of the worst-case latency.
- It demonstrates the effectiveness of SounDroid with real-world mobile sound applications. In contrast to the poor experience on Android, our experiment with SounDroid shows a significant improvement in the QoS of applications. To this end, we expect that with SounDroid, many interesting mobile sound applications could be easily developed.

II. Mobile Sound Applications

Building upon the useful features of sounds in mobile environments, a variety of interesting services has increasingly emerged. We categorize them as follows:

- *Localization.* Sounds can be used to identify the location of objects. Multiple mobile devices emit acoustic signals in turn, and calculate the distance to each source device using the time of arrival. With this information, the location can be derived by applying computational geometry such as triangulation. There are some representative works such as BeepBeep [6], GuoGuo [1], and Phone-to-Phone 3D localization [8].
- *Gesture Recognition.* We can recognize the gesture of users with sounds. A device plays an acoustic signal and records it back, measuring changes in the signal, e.g. Doppler Effect, during the transmission. This approach makes use of machine learning techniques to train possible acoustic changes for gestures, and recognizes them based on the training set. It enables in-air gesture recognition without any specialized devices. SoundWave [2], DopLink [3], AirLink [4], and Spartacus [5] are representative works in this category.
- *Mobile Motion Game.* Acoustic distance measurement technology enables a new kind of mobile game, i.e., a mobile peer-to-peer motion game. For example, SwordFight [7] makes each of two mobile devices become a virtual sword. Users can play a fencing game which determines the success of attacks based on the distance between two virtual swords.
- *Audio Device Collaboration.* There are a couple of mobile applications that coordinate multiple devices to produce high-quality harmonized sound reproduction. For example, in Mobile Maestro [9], six mobile devices play the different audio channels of 5.1 ch surround sound, enabling immersive sound experience everywhere. More simply, multiple devices can play the same music much louder, known as GroupPlay [13].

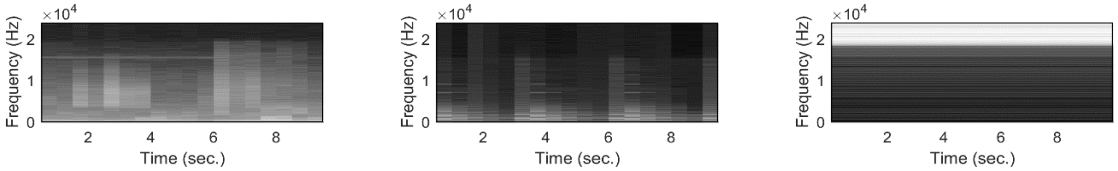
Basically, the proliferation of mobile sound applications is based on the interesting features of sounds: 1) sound propagation can help to capture in-air and/or device-to-device changes, 2) sounds in the inaudible frequency range makes unnoticeable sensing possible, and 3) most mobile devices, even low cost devices, include speakers and microphones. We expect that a more number of mobile sound applications will emerge with the advance of audio device performance and form factors. We will thus have more applications running simultaneously, resulting in more contention on audio resources. Therefore, it is imperative to provide a system solution that optimizes the audio device management in mobile platforms.

A. Requirement Analysis

From the above mobile sound applications, we have identified three major requirements of audio playback requests.

R1: Various stringent timing constraints. Many mobile sound applications impose diverse timing requirements related to the use of an audio device. Such requirements can be broadly characterized as follows.

- *Bounded latency*. Some sounds should be played no later than a specific time delay limit. For example, users expect (audible) feedback for their inputs within 100ms for their QoS satisfaction [14]. Thus, user-interactive applications, such as mobile games, request that the playback latency should be less than 100ms.
- *On-time playback*. Some sounds should be played as closely as possible to a specific time point. For instance, in Mobile Maestro [9], multiple mobile devices are arranged to play sounds at the same time for immersive sound reproduction. According to human auditory perception [15], such multi-device audio collaboration requires tight timing synchronization (i.e., 1ms) for accurate sound reproduction. If not, an auditory image would be localized in a wrong direction. This requires each mobile device to play sounds at its own designated time point with an error range of 1ms.
- *Tight and regular sensing intervals*. Sensing applications monitor dynamic surrounding environments continuously by emitting acoustic signals periodically. In this case, the sensing



(a) Movie (Les Miserables). (b) Phone ring. (c) Acoustic signal used in Mobile Maestro [9].

Fig. 1: The spectrogram of different types of audio streams. It presents the spectrum of frequencies as they vary with time. The brighter the spectrum is, the louder the sound is at the corresponding frequency.

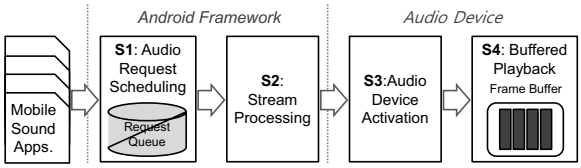


Fig. 2: Audio Playback Procedures on Android.

rate has a critical impact on their QoS. For example, as the sensing rate of mobile motion games increases, users might experience much better game quality due to the increased sensing accuracy. Typical sensing rates for mobile motion games range from 10Hz to 30Hz [7], [16].

R2: Atomic playback. We observe that the sounds of mobile sound applications should not be interrupted in the middle of emission. For example, users might feel very annoyed if the surround sound playback is interfered by sensing audio requests. Also, even between sensing applications, signal interference might cause the difficulty in signal detection and finally result in an unacceptable decrease in sensing accuracy.

R3: Use of different frequency ranges. We also find that mobile sound applications make use of different acoustic frequency ranges. Figure 1 shows the spectrogram of three representative audio streams. As shown in the figure, conventional sounds of music or phone ring utilize the audible frequency range, i.e., between 0 and 18k/20kHz. In contrast, acoustic signals used for sensing applications are usually modulated in the inaudible frequency range (above 18kHz). Moreover, individual applications presume that they can use their frequency ranges exclusively. For example, user-interactive applications aim to provide clear sounds for quality of sound. Similarly, sensing applications request that signals do not overlap with each other for high sensing accuracy.

III. SoundDroid Framework

This section presents the overview of SounDroid that supports real-time characteristics of various audio playback requests made by multiple mobile sound applications. To design the framework carefully, we now investigate the audio playback procedure on Android, then identify major technical challenges, and discuss the overall architecture of SounDroid.

A. Understanding Playback Procedure

Audio playback requests undergo the following four steps to produce their sound. As shown in Figure 2, the first two steps belong to the service layer in Android, while the later two are related to the audio device hardware.

S1: Audio request scheduling. The audio requests are scheduled to utilize the audio resource according to a scheduling policy. For example, Android schedules requests according to the LIFO (Last-In-First-Out) policy to handle the most recent audio request first.

S2: Stream processing. The audio source of the requests is decoded to the raw data playable by the audio device.

S3: Audio device activation. Before the raw data is given to the audio device, the mobile platform checks the status of the device. If it is inactive, the platform delays the process until it becomes active.

S4: Buffered playback. The raw data is split into several chunks, called *frame*, which is a basic unit for interaction with an audio device. Then, each frame is inserted into the frame buffer, and the audio device plays it frame-by-frame sequentially.

Since this internal procedure in Android does not properly handle timing and frequency constraints of real-time audio requests, it cannot satisfy the QoS requirements of real-time mobile sound applications. In the next subsection, we figure out the two major technical challenges in the audio request scheduling and dispatching.

B. Challenges in Audio Request Scheduling and Dispatching

Audio request scheduling. All the three R1, R2, and R3 requirements are not yet supported on the current Android platform and its variation with real-time support, RTDroid [10]. The R1 requirement indicates that many audio requests are very time-sensitive. They can require a high precision of hundreds of microseconds in accessing an audio device. In order to meet such a requirement, it needs to take timing constraints such as deadlines explicitly into account when making scheduling decisions. However, Android and RTDroid employ LIFO and fixed-priority scheduling, respectively, without considering them. In addition, the R2 and R3 requirements suggest that an audio device should be used in a non-preemptive manner for some audio requests and that some audio requests can share the audio device depending on their frequency ranges. However, Android and RTDroid commonly have preemptive scheduling over an audio device and do not care about the frequency range of audio requests. Thus, it entails to design a non-preemptive, deadline- and frequency-aware scheduling scheme.

Audio request dispatching. According to our preliminary experiments on Android, the audio request dispatching steps S2-S4 incur unpredictable, non-negligible audio playback latency, ranging from 8ms to 150ms. We seek to deal with such dispatching latency for the following reasons. First, the long dispatching latency compromises the QoS of applications significantly. For example, the latency longer than 100ms will decrease users satisfaction seriously. It will also lead to unacceptable accuracy for acoustic sensing with low sensing rates. Second, the dispatching latency is unpredictable because

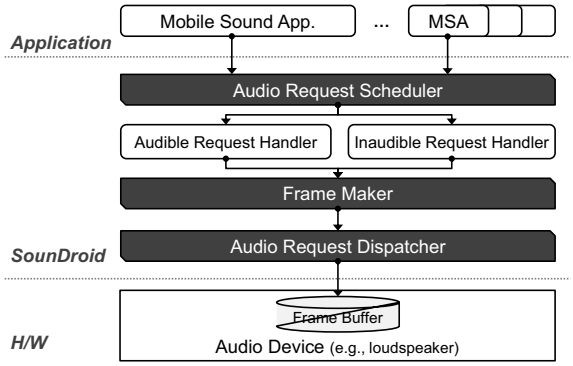


Fig. 3: Overall architecture of the SounDroid framework.

it heavily depends on the status of the audio device and/or the frame buffer. This makes it complicated to schedule audio requests in a way that they emit audio signals promptly at their designated time.

Unsurprisingly, Android also makes an effort to address the dispatching latency issue. It proposes a solution that pre-processes audio streams to reduce the stream processing latency in Step S2. However, it has rarely considered the latency occurring in the steps S3 and S4. Therefore, we need to develop a systematic solution for the minimization and regularization of the dispatching latency in SounDroid.

C. SounDroid Architecture

Figure 3 illustrates the overall architecture of SounDroid including three core components for addressing the challenges on existing platforms. Each mobile sound application first requests an audio playback with its own frequency and timing requirements. *Audio Request Scheduler* then schedules the audio requests in accordance with their constraints, specifically a deadline and the latest possible playback start time. The scheduled request is handled in its corresponding thread, and delivered to *Frame Maker* in a unit of frame[1]. Note that Audio Request Scheduler can schedule multiple requests at the same time through the frequency-based multi-resource scheduling. Once Frame Maker receives frames, it merges them into a single frame if it is necessary, i.e., there are multiple scheduled requests. Finally, the merged frame is inserted to the frame buffer through *Audio Request Dispatcher*. At this time, to mitigate the dispatching latency, Audio Request Dispatcher applies several optimization techniques such as the early audio device activation and latency stabilization.

In later two sections, we investigate more details of each technique for enabling a high degree of real-time capability on SounDroid. We first observe how both Audio Request Scheduler and Frame Maker support diverse real-time characteristics of mobile sound applications (Section IV). Then, we explain our optimization schemes for the audio request dispatching latency, applied to Audio Request Dispatcher (Section V).

IV. Real-Time Audio Scheduling

As discussed in Section III-B, there are several challenges when designing scheduling strategies for audio requests. First, it needs to support the various types of tight timing constraints. Second, an audio device should be utilized in a non-preemptive manner. Third, it needs to recognize the frequency range that each request mostly use to efficiently schedule multiple requests. In this section, to address such challenges, we first model the audio requests of mobile sound applications reflecting their timing and frequency requirements. Based on the request model, we introduce EDF with Virtual-scheduling (EDF-V) mechanism for satisfying the timing requirements of audio requests, and Acoustic Frequency Division Scheduling (AFDS) for frequency-based efficient scheduling.

A. Audio Request Modeling

Mobile sound applications utilize an audio device with multiple audio requests, and each audio request A_i can be modeled as follows:

$$A_i = (F_i, S_i, C_i, D_i, T_i).$$

Each A_i requests the exclusive and non-preemptive use of frequency F_i, where F_i is either `audible` or `inaudible`. The request can be regularly repeated with a minimum separation delay T_i, and each request has the earliest possible playback start time S_i, a constrained relative deadline $D_i \leq T_i$ (i.e., an absolute deadline $d_i = S_i + D_i$), and a playback duration C_i. A request A_i is *periodic* if $0 < T_i < \infty$, or *one-time* otherwise. It is worthy to note that S_i can be used to specify an on-time playback request. For example, when a sound application, at time t_0, wants to emit an acoustic signal punctually at a targeted time instant t_1 with a marginal error ϵ, it generates a request A_i with $S_i = t_1$ and $D_i = C_i + \epsilon$. Let R_i denote the time when a request A_i is made, where $R_i \leq S_i$. Then, A_i is said to be *prearranged* if $R_i < S_i$ or *unplanned* otherwise. In general, on-time playback requests are typically arranged some time before its targeted playback time, and the scheduling information of each instance of periodic requests can be also given in advance. On the contrary, audible feedback to user inputs are specified as unplanned requests.

SounDroid introduces new playback APIs in order to manage a life cycle of audio requests. First, mobile sound applications request an audio playback by using our proposed API, called *SounDroidPlay()*, with their time and frequency requirements. Audio Request Scheduler then stores the requests on *Audio Request Queue*, and schedules them based on their requirements. When a request completes its playback, it is removed from the queue, except for periodic requests. We denote the j-th instance of each periodic request A_i as A_i^j, where $j \geq 0$. If A_i^j's playback is finished at time t, Audio Request Scheduler just updates S_i such that $S_i^{j+1} = \max(S_i^j + T_i, t)$. SounDroid also supports other types of playback operations including stop, pause, and resume. For example, if A_i is stopped, Audio Request Scheduler just removes A_i from the request queue. Paused requests are stored in *Pending Queue* having their remaining playback duration, and returned back to Audio Request Queue when the resume API is called.

B. EDF-V Scheduling Algorithm

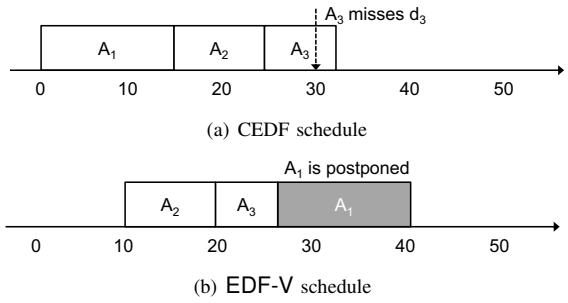
In order to satisfy the timing requirements of individual audio requests, we design a new scheduling algorithm, named EDF-V (Earliest Deadline First with Virtual-scheduling). Our design is based on two observations. One is that audio requests basically make atomic use of an audio device, and the other is that they often impose strict timing requirements in

[1]We assume that all the requests are pre-processed into the raw format of samples

Algorithm 1 Postpone condition of EDF-V

Input: Audio request set $\mathcal{A}$, the current scheduling point t
Output: A postpone decision

```
1: P ← all the playable audio requests at time t in A
2: A_D ← the earliest deadline request in P
3: if isPostponedOnCEDF(A_D, A) then
4:    return Postpone
5: end if
   /* Start the virtual estimation of given audio request sets */
6: while A is not empty do
7:    P ← all the playable audio requests at time t in A
8:    A_D ← the earliest deadline request in P
9:    if S_D > t then
10:      return Do not postpone
11:   else
12:      if isPostponedOnCEDF(A_D, A) then
13:         /* Virtually delay start time of A_D */
14:         t ← max(t, the earliest S_i in A)
15:      else
16:         if t + C_D > D_D then
17:            return Postpone
18:         else
19:            A ← A \ A_D
20:            t ← t + C_D
21:         end if
22:      end if
23:   end if
24: end while
25: return Do not postpone
```



(a) CEDF schedule

(b) EDF-V schedule

Fig. 4: Scheduling example with three audio requests: each request is specified in terms of $A_i(S_i, C_i, d_i)$; $A_1(0, 15, 100)$, $A_2(10, 10, 30)$, and $A_3(20, 7, 30)$.

terms of tight deadlines. Thereby, we consider non-preemptive deadline-based scheduling.

Traditional non-preemptive EDF-based scheduling. The non-preemptive EDF (NP-EDF) algorithm is known to be an optimal work-conserving non-preemptive scheduler [11]. However, the scheduling performance of NP-EDF is limited due to its work-conserving nature, because adding idle time can enhance schedulability under non-preemptive scheduling. An extension of NP-EDF, called Clairvoyant EDF (CEDF) [12], was introduced that adds idle time based on future scheduling information. The only difference between NP-EDF and CEDF is explained as follows. At time t, NP-EDF always schedules the request A_i with the earliest deadline as long as $S_i \leq t$. However, scheduling A_i at t may force another request to miss its deadline, and CEDF aims to avoid such a situation. Let S_i^{max} denote the latest possible start time of A_i such that $S_i^{max} = d_i - C_i$. Then, at time t, CEDF delays scheduling A_i if the following condition holds:

$$t + C_i > \min_{A_j \in \mathcal{A} \setminus A_i} S_j^{max}, \tag{1}$$

where $\mathcal{A}$ is a set of unscheduled requests.

With such a simple strategy of delaying A_i, CEDF is able to dominate NP-EDF in schedulability. However, CEDF has a room for further improvement. While CEDF considers just the direct influence of scheduling A_i on other requests, it may offer the best decision for the next request to be scheduled, but not for other remaining requests. Figure 4(a) shows an example for illustration. In Figure 4(a), CEDF schedules A_1 at time 0 since it does not violate the above condition Eq. (1). However, it leads to the deadline miss of A_3 as shown in the figure. This is because CEDF does not consider the cascade effect of scheduling A_1 on other remaining requests.

Under real-time audio request scheduling, such a cascade effect should be considered for improved schedulability due to the following reasons. First, many audio requests often come with extremely tight deadlines. For example, on-time playback requests for high-quality sound reproduction may impose a relative deadline of $(C_i + 1)$ ms. Second, acoustic sensing applications typically demand higher sensing rates and smaller delay jitters, and such demands will be specified as shorter periods and smaller relative deadlines. In this case, the cascade effect of scheduling one request on others would be more critical to meet tight timing requirements.

EDF-V scheduling. So as to overcome the limitation of CEDF, we develop a new algorithm EDF-V that generalizes CEDF in the sense that EDF-V makes decision of postponing A_i by checking not only its direct influence but also its cascade effect on other requests. Such checking can be viewed as virtual scheduling of not only a single request but also a chain of other requests under CEDF.

Algorithm 1 shows the elaborated postpone condition of EDF-V. At every scheduling point t, if A_D does not meet the postpone condition of CEDF, EDF-V checks whether a deadline miss will occur or not by the cascade effect of scheduling A_D through iterative virtual scheduling (lines 6-24). In other words, EDF-V virtually schedules the given request set $\mathcal{A}$ when A_D is determined to be scheduled right away without being postponed. In the virtual scheduling, the postpone condition of Eq. (1) is used to determine to virtually postpone a request or not (lines 12-14). When a deadline miss occurs during the virtual scheduling, it decides to postpone the request A_D (lines 16-17). For example, in the previous example, at time 0, CEDF schedules A_1 for playback without postponing (see Figure 4(a)). EDF-V, instead, virtually schedules remaining requests before dispatching A_1. It then postpones the playback of A_1 since the deadline miss of A_3 is detected, and finally meets timing constraints of all the requests (see Figure 4(b)). Note that, during this pre-evaluation, each periodic request is transformed into multiple one-time requests.

Since EDF-V follows the NP-EDF mechanism and uses the postpone condition of CEDF during the virtual scheduling, it dominates NP-EDF and CEDF in terms of schedulability[2]. Yet, its dominance comes from iterative virtual scheduling. Thus, for each scheduling decision, EDF-V has the running

[2]EDF-V is not guaranteed to dominate NP-EDF and CEDF in terms of a deadline miss rate because the current postpone decision may cause more deadline misses in the future.

time complexity of $O(n \log n)$, where n is the number of one-time requests including the transformed instances of periodic requests. This is because it virtually schedules all the remaining requests based on CEDF which is known to have the complexity of $O(\log n)$ at each scheduling point. To minimize such overhead, EDF-V first terminates the iteration when it encounters an idle time since it means that a set of virtually scheduled requests and a set of remaining requests are mutually exclusive and do not affect each other (lines 9-10 in Algorithm 1). EDF-V also bounds a number of one-time requests generated from a single periodic request, denoted N_P. It may decrease the scheduling performance of EDF-V due to the limited future scheduling information, but provide better support for our on-line scheduling with the reasonable scheduling complexity.

C. Acoustic Frequency Division Scheduling (AFDS)

As mentioned in Section II-A, mobile sound applications utilize a certain acoustic frequency range depending on its audio source, enabling us to adopt the binary frequency requirement, `audible` and `inaudible`, to our model. Leveraging this constraint, we introduce Acoustic Frequency Division Scheduling (AFDS) to improve scheduling rate by allowing multiplexing of audible and inaudible audio requests in a frequency domain. To enable AFDS, SounDroid maintains two scheduling queues, one for audible requests and the other for inaudible ones. Audio Request Scheduler assigns each audio request to one of two queues and disjointly schedules audio requests based on their frequency use.

FrameMaker then combines two audio frames scheduled from each request queue into a single one so that two audio requests can access the audio device i) concurrently in a time domain and ii) separately in a frequency domain. Note that although our model defines an audio request as either audible or inaudible, some audible source such as a music played with various musical instruments can also use an inaudible frequency range, which can cause interference between two frames in a frequency domain. In order to guarantee that two audio frames use the exactly isolated range of frequencies, Frame Maker first filters two frames based on F^A, an upper threshold on a human hearing range[3], with the complexity of $O(k \log k)$, where k is a size of the frame. More specifically, it applies a low-pass filter for audible frames and a high-pass filter for inaudible frames with F^A as a cut-off frequency. After combining two filtered frames without overlapped frequency ranges, Frame Maker passes it to Audio Request Dispatcher, and then two audio requests can be played on the audio device at the same time.

V. Audio Request Dispatching Optimization

In addition to the audio request scheduling, another important challenge for the QoS satisfaction of mobile sound applications is to minimize and stabilize latency during the audio request dispatching. In this section, we first explore the major reasons for such latency, and describe our optimization techniques to address them.

A. Latency from Dispatching Audio Requests

As discussed in Section III-B, Android still incurs a huge amount of playback latency, especially during the audio request dispatching which mainly consists of audio device activation and buffered playback.

- **Audio device activation latency.** Traditionally, mobile platforms manage its audio device efficiently in terms of energy, but not in terms of latency. For this reason, they initially let the audio device stay in the inactive state. When an audio request is newly dispatched, the audio device first needs to be awake from the inactive state, and then starts to play the requested audio sound. Once all the requested playbacks are completed, it goes into the silence state. If the silence lasts for a while (e.g., 3 seconds), mobile platforms expect that there will be no additional requests and inactivate an audio device for reducing unnecessary energy consumption. However, with this management strategy, since the audio device is activated in an on-demand manner, the requested audio playback is delayed until all the activation procedures finish, experiencing a large and unpredictable amount of latency (up to 100-150 milliseconds on the Nexus 4 device).
- **Buffering latency.** Even after the audio device is activated, there is another latency that comes from the buffered playback. Mobile platforms use a buffer to deliver frames of audio requests to the audio device without audio glitches. However, this buffering mechanism produces an inevitable latency. Since there exist multiple frames in the frame buffer, the playback of newly requested frames can be started only after all previously inserted frames and the currently played frame are emitted. Hence, this buffering mechanism introduces latency on playbacks up to `frame length` $\times$ (`buffer size` + 1). In addition, latency occurs even when the buffer is empty due to the state transition of the audio device from silence to playing state.

B. Optimizing Audio Request Dispatching Latency

Above latency is not negligible for real-time mobile sound applications that have critical timing constraints. To address above problems, SounDroid introduces the simple, yet effective optimization techniques to alleviate the impact of audio request dispatching latency.

First, SounDroid hides the activation overhead by preparing an audio device in advance. Toward this, SounDroid offers a special permission, called `LOW_PLAYBACK_LATENCY`, and activates an audio device if there is a running application with the permission. Thus, SounDroid supports zero activation latency to mobile sound applications which declare the permission in their configuration files (e.g., the Manifest file in Android applications). This pre-activation scheme may increase energy consumption compared to the on-demand one. However, since most mobile sound applications continuously use the audio device during its execution, we expect that the additional energy consumption would not be too large.

SounDroid also compensates the buffering latency through the adaptive audio request scheduling with the buffering latency stabilization. Since the buffering latency unexpectedly changes according to the number of buffered frames, it is difficult to determine the exact amount of delay. Thus, SounDroid makes the latency stable by writing zeroed data to the buffer. It enforces loudspeakers to emit a mute sound as well as keeps the buffer full. In other words, it always incurs the same amount of latency, i.e., the worst-case buffering latency, denoted as L_B^{WC} (in our implementation, $L_B^{WC} = 20ms$). Audio Request Scheduler then compensates the stabilized latency

[3]In this paper, we configure F^A as 18kHz, the maximum audible frequency of common adults.

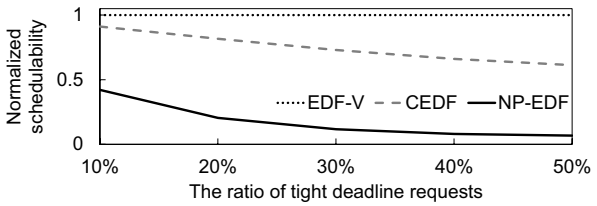


Fig. 5: Scheduling performance under different tight deadline request ratio (normalized to EDF-V).

by re-configuring scheduling parameters of each audio request A_i. It adjusts the earliest possible start time S_i and the latest start time S_i^{max} to $\max(R_i, S_i - L_B^{WC})$ and $d_i - C_i - L_B^{WC}$, respectively. From the application's view point, SounDroid enables an audio request to emerge from a loudspeaker without the buffering latency by scheduling it at most L_B^{WC} earlier than the originally requested start time.

The benefits of latency compensation can differ depending on the type of audio requests. It can be much beneficial to prearranged audio requests which their scheduling information is already known, i.e., $R_i < S_i$. For example, with this compensation mechanism, we can guarantee that SounDroid always satisfies the tight deadline of prearranged requests for high-quality sound reproduction (e.g., $D_i = C_i + 1ms$) by avoiding the unexpected buffering latency. Unplanned audio requests, on the other hand, cannot get any advantage from the compensation mechanism since these requests are scheduled as soon as they are released, i.e., $R_i = S_i$. In addition, their response time may even get worse due to the latency stabilization which always leads the worst-case latency L_B^{WC}. However, in Section II-A, we study that such kinds of audio requests require loose timing constraints (e.g., a latency of up to 100 milliseconds for acoustic feedback). Thus, SounDroid improves the system-level performance through the adaptive audio request scheduling with the latency stabilization for prearranged audio requests, while offering acceptable time delays to unplanned audio requests.

VI. Evaluation

In this section, we evaluate the capability of SounDroid to support the QoS requirements of mobile sound applications with following metrics using simulation and our prototype implementation:

- *Scheduling performance:* How well does SounDroid support various requirements of audio requests with proposed EDF-V and AFDS?
- *Request Dispatching latency:* How does SounDroid optimize the audio request dispatching latency?
- *QoS satisfaction:* Do mobile sound applications running on SounDroid work in practice?

A. Simulation

We use simulation to evaluate the scheduling performance of proposed EDF-V algorithm with sets of various audio requests in comparison to CEDF and NP-EDF. We measured how scheduling performance of three algorithms changes as the possibility of the cascade effect increases.

Simulation methodology. Our simulation was conducted with various sets of randomly generated audio requests which have realistic acoustic requirements. We assumed that all requests use inaudible frequency range, and they are one-time requests. It is worth to note that a periodic request can be converted to a number of one-time requests. Each request A_i gets randomly generated start time, playback duration, and deadline such that $S_i \in$ [0, 3000], $C_i \in$ [10, 40], and $D_i \in$ [C_i+100, C_i+1000], and each parameter is uniformly chosen within given intervals. Some requests have tight deadline to model on-time audio requests and high rate sensing requests. Tight deadline parameter is randomly decided in [C_i+1, C_i+30]. We generated 100,000 sets of audio requests with the ratio of tight deadline requests ranging from 10% to 50%, and each set has 50 requests.

Simulation results. Figure 5 plots the scheduling performance of three algorithms, EDF-V, CEDF, and NP-EDF. The x-axis represents the ratio of tight deadline requests on the request sets, and the y-axis represents the number of schedulable request sets relative to EDF-V. The figure shows a widening performance gap between EDF-V and others as the ratio of tight deadline requests increases. CEDF can schedule 90% of request sets those EDF-V can schedule when there are 10% of tight deadline requests. However, the performance of CEDF is only about 60% of that of EDF-V when the portion of tight deadline requests is 50%. In other words, EDF-V can have a lot of benefits on scheduling performance when there are many high rate sensing requests and on-time requests which can frequently cause the cascade effect. The performance of NP-EDF is the worst among the three scheduling algorithms with loss of up to 93% compared to EDF-V.

During the simulation, we also measured the number of iterations on virtual scheduling of EDF-V. On average, the number of iterations is about 6 (28 in the worst-case), which translates to 6 times longer running time than CEDF. However, in practice, the additional computation overhead of EDF-V can be ignored. In our environment (with a 1.5GHz of processor speed), it shows just 2.5μs of running time on average with 1.4μs of standard deviation.

B. Experiment with Android device

Experiment methodology. We have implemented the prototype of SounDroid as an extension to Android 4.4.1 (Kitkat). We conducted experiments with a Nexus 4 Android smartphone. During all the experiments, we configured two acoustic parameters, a frame length and a buffer size, as 10ms and 1, respectively, setting the worst-case buffering latency L_B^{WC} to 20ms. In addition, for the EDF-V scheduling algorithm, we set N_P, a number of one-time requests transformed from a single periodic request, as 10. For each evaluation, we used a different set of custom mobile sound applications, each of which requests an audio playback according to given requirements. Especially, for measuring the QoS satisfaction of real-world applications running on SounDroid, we made use of the audio collaboration application and custom sensing applications, and evaluated their performance in terms of the synchronization accuracy and sensing accuracy, respectively.

1) Impact of Audio Request Dispatching Optimization: We conducted an experiment for observing how well SounDroid optimizes the audio request dispatching latency. Toward this, we arbitrarily generated 50 audio request sets consist of one-time inaudible requests, and scheduled each set on both Android and SounDroid. We then measured how much dispatching latency each audio request A_i experiences in terms of the

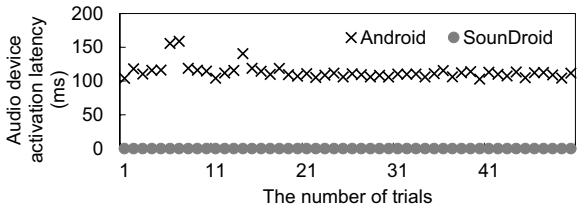


Fig. 6: Audio device activation latency on both platforms.

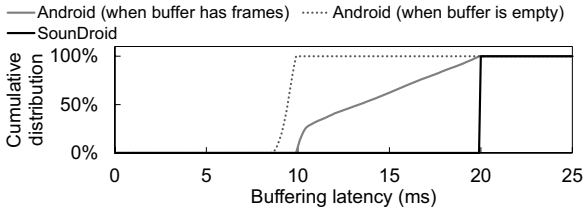


Fig. 7: Distributions of buffering latency on both platforms.

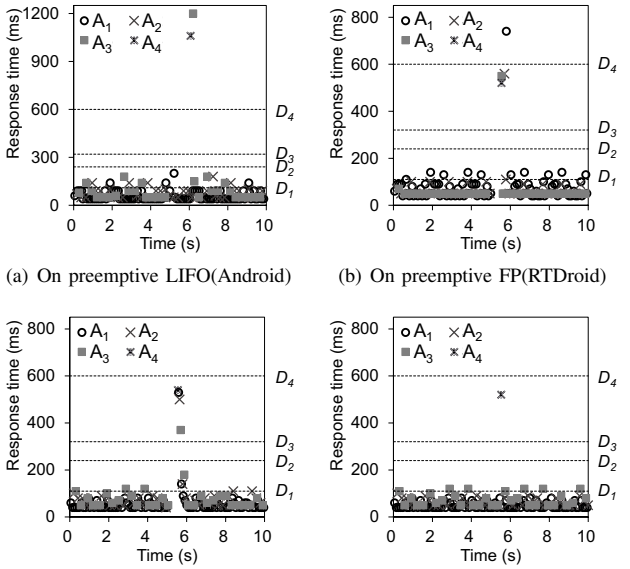
difference between an actual playback start time and the earliest possible start time S_i. It is noteworthy that time delays driven by interferences during the audio request scheduling was discarded from this result.

Audio device activation latency. Figure 6 presents the measured activation latency over 50 trials. On Android, an audio request experiences a high degree of activation delay, varying from 102.9ms to 158.9ms, which is close to, or even exceeds the common acceptable user response time of 100ms [14]. In contrast, SounDroid provides much more responsiveness with zero latency, since it hides the activation overhead through the pre-activation of an audio device.

Buffering latency. As presented in Figure 7, the buffering latency on Android is differently distributed according to the buffer status. When the buffer is empty, an audio request is delayed by from 8ms to 9.89ms due to the state transition overhead. The latency changes more unexpectedly with the existence of buffered frames, increasing up to 20ms. Such buffering latency can have a negative effect on the QoS of sensing applications. For example, its unpredictability makes the sensing jitter grow, leading a decrease of the sensing accuracy. On the other hand, SounDroid achieves to provide 20ms of much predictable buffering latency. This deterministic delay enables a prearranged request A_i such as a sensing request to be played at S_i through compensating the latency. As a trade-off for this benefit, unplanned requests always experience 20ms of time delay. However, as discussed before, it is an acceptable amount of latency for such kinds of audio requests, compared with their loose timing constraints.

2) Impact of Real-Time Audio Request Scheduling: This experiment evaluated the scheduling performance of SounDroid on mobile devices. Toward this, we first observed how our custom mobile sound applications are scheduled on various mobile platforms including Android, RTDroid, and SounDroid.

In this experiment, to solely focus on the scheduling issue, we implemented a platform to which our latency optimization techniques are applied, and scheduled audio requests on the platform with three different scheduling algorithms including preemptive LIFO scheduling for Android, preemptive Fixed-Priority (FP) scheduling for RTDroid, and EDF-V scheduling



(a) On preemptive LIFO(Android) (b) On preemptive FP(RTDroid)

(c) On EDF-V(SounDroid) w/o AFDS (d) On EDF-V(SounDroid) w/ AFDS

Fig. 8: Response time of each audio request.

TABLE I: Audio request specifications (the time unit is milliseconds.).

Index	F_i	R_i	C_i	T_i	D_i
A_1	Inaudible	0	40	110 (periodic)	110
A_2	Inaudible	100	50	240 (periodic)	240
A_3	Inaudible	0200	50	320 (periodic)	320
A_4	Audible	5,000	500	∞ (one-time)	600

for SounDroid. As seen in Table I, our custom applications generate three periodic inaudible requests (A_1, A_2, and A_3) and a single one-time audible request (A_4). They were scheduled on each scheduling mechanism for about 10s, their least common multiple period. We then measured the response time of every instance for each audio request A_i. Note that we assumed the preemptive FP scheme assigns a higher priority to an audio request which has a larger index number.

Figure 8(a) and (b) indicate that, under the scheduling mechanism of both Android and RTDroid, the response time of audio requests increases, even incurring deadline misses. This is because they assign a priority to audio requests without a deep understanding of their constraints. For example, under preemptive LIFO, A_3 is delayed by up to 1.15s due to other newly-arrived requests. In addition, the interference from higher priority requests makes A_1 experience 23 times of deadline misses on preemptive FP. The EDF-V scheduling algorithm of SounDroid also misses the deadlines of A_1, A_2 and A_3 if AFDS is not supported (see Figure 8(c)). The major reason is that A_4 which has a long playback duration competes to use an audio device with others even though it uses a different frequency range. With the support of AFDS, as shown in Figure 8(d), SounDroid satisfies all the timing constraints by scheduling A_4 independently of other requests based on their frequency requirements.

3) QoS Satisfaction for Real-World Applications: In this experiment, we evaluated the effect of SounDroid on real-world mobile sound applications. Toward this, we measured the QoS of MobileTheater [9], an audio device collaboration application. Note that the current Android cannot schedule multiple

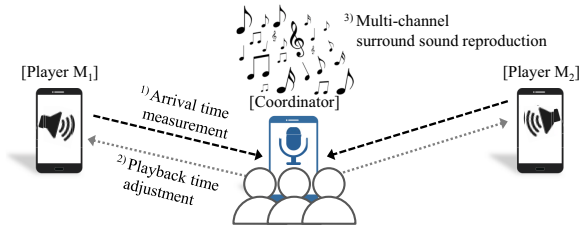


Fig. 9: Procedures of MobileTheater for immersive sound reproduction.

TABLE II: One-time audio requests of MobileTheater on each mobile device M_k (the time unit is seconds.). P_k is determined after the arrival time measurement, i.e., $P_k \geq d_{sig}$.

Index	F_i	R_i	S_i	C_i	D_i
A_{sig}	inaudible	0	5	0.011	$5 + C_{sig} + 0.001$
A_{ch}	audible	P_k	$P_k + 5$	30	$5 + C_{ch} + 0.001$

audio requests from mobile sound applications properly due to its preemptive scheduling. To carry out a fair comparison, in this experiment, we modified Android's scheduler to support the non-preemptive LIFO.

MobileTheater. In MobileTheater, each mobile device is assigned its own speaker role (e.g., left or right), and plays its own audio channel stream for multi-channel surround sound reproduction. For enriching user experience, it is required to support the following tight timing guarantee; sounds emanating from each device should arrive at a listener within 1ms of arrival time difference. For this reason, MobileTheater tries to achieve the high degree of accuracy through the sound arrival time synchronization as seen in Figure 9:

- Each mobile device M_i emits an inaudible signal at the measurement start time, and a special coordinating device measures the sound arrival time of each device.
- The coordinator then adjusts each device's playback start time P_k based on the measured arrival time.
- Each device starts to play its audio channel stream at the adjusted time.

To support the high synchronization accuracy, MobileTheater should play two different audio requests very predictably (up to 1ms). Thus, to guarantee the high degree of predictability, each playback is prearranged, e.g., 5s before its targeted playback time.

In our experiment, MobileTheater reproduced stereo surround sound effects with two mobile devices M_1 and M_2. Each mobile device played two audio requests of Mobile Theater as seen in Table II. In addition, to observe the effect of audio request scheduling, we simultaneously run three custom sensing applications on M_1 (see Table III for their constraints). We then measured the QoS of MobileTheater, i.e., the synchronization accuracy, over 50 times both on SounDroid and Android. At the same time, we also estimated the sensing rate of each sensing application.

Synchronization accuracy of MobileTheater. Figure 10 shows that Android never meets the desired accuracy of 1ms of error in all cases. In the worst-case, the error increases up to 111.2ms due to the interference of sensing applications as well as the audio request dispatching latency. Such amount of error may cause a significant degradation of user experience.

TABLE III: Periodic audio requests of sensing applications (the time unit is milliseconds.). Each request has an implicit deadline, i.e., $T_i = D_i$.

Index	F_i	R_i	S_i	C_i	T_i
A_1	inaudible	100	100	39	300
A_2	inaudible	55	55	27	100
A_3	inaudible	712	712	11	30

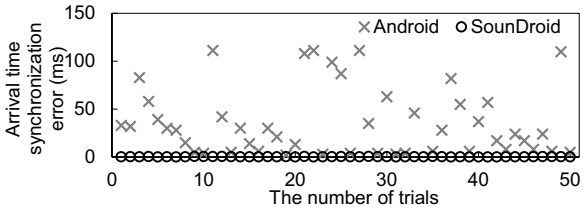


Fig. 10: Arrival time synchronization accuracy of MobileTheater.

For example, multiple correlated sounds arrive 100ms after the first-arriving sound. An auditory event is then dominantly localized by the first-arriving sound and the delayed sounds are perceived as an echo of the first sound [17]. It is worth noting that in all trials, SounDroid meets the timing requirement (i.e., 1ms) for MobileTheater with 625μs of synchronization error (at most). Thus, SounDroid enables users to be immersed in high-quality sounds of MobileTheater even in multi-application environments.

Sensing rate of sensing applications. As seen in Figure 11, both Android and SounDroid successfully support the desired QoS of A_1 and A_2, each of which has a loose timing constraint (e.g., 3.3Hz and 10Hz of sensing rate accordingly). On the other hand, A_3 which frequently requests an audio playback in every 30ms experiences 24.5% of the performance degradation, i.e., 7.6Hz of sensing rate decrease, on Android, compared to SounDroid. This is because Android delays A_3 if there is a newly-arrived request from other applications including MobileTheater, even though a playback of A_3 is more urgent. Note that such the LIFO mechanism makes 49.1% of requests of A_3 miss its deadline (30ms). Such performance gap can make a huge amount of difference on user experience. For example, if a user plays a table tennis game, one of the mobile motion games, with 10m/s of average racket speed, Android increases the racket location error by up to 9.5cm in comparison with SounDroid. Hence, the improved real-time capability of SounDroid leads to enrich user experience more.

VII. Discussion

So far, we have introduced SounDroid, specifically focusing on its approach and effectiveness. While it is the first approach towards mobile audio device management, there are a number of issues to be discussed.

Limitations of real-time audio request scheduling. Our real-time audio request scheduling is yet restricted in handling the following situations. First, each audio request can have a different importance, e.g., a phone call request versus a music request. Second, the non-preemptive long playback of audio requests can block others for a long time. Last, sometimes, multiple audio requests, e.g., background music and sound effects in a game, can simultaneously share the same range of frequencies. We believe that our request scheduling scheme including EDF-V and AFDS could be extended to resolve such

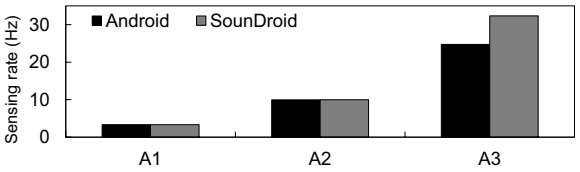


Fig. 11: Sensing rate of each sensing application.

issues. We can classify and prioritize audio requests based on their importance and allow preemption. It is also possible to use an alternative resource, e.g., vibrator, instead of an audio device for audio requests which are expected to experience long starvation time. Furthermore, through more fine-grained analysis for frequency requirements, we will re-model audio requests and extend the AFDS scheduling mechanism.

Microphone support. SounDroid focuses mainly on managing an audio output device, i.e., a loudspeaker. However, many mobile sound applications also require to use a microphone. For instance, sensing applications make use of the microphone as a receiver for sensing signals. In order to improve the QoS of mobile sound applications, it is essential to manage not only the audio output device, but also the input device. Thus, our future work includes investigation of scheduling and optimization techniques for the audio input device, and introduces a more powerful framework for mobile sound applications.

Time delays on other resources. Mobile sound applications can also suffer from latency during using other resources. As an example, the execution of core components of SounDroid can be delayed due to the CPU competition, interrupt handling, and garbage collection on Android. These traditional real-time issues have been addressed many times in existing works such as RTLinux [18] and FijiVM [19]. Therefore, SounDroid would be able to support a higher level of QoS for mobile sound applications by incorporating such techniques.

VIII. Related Works

Many operating systems such as RTLinux [18] and RTEMS [20] have supported real-time applications. Most of them try to meet timing requirements through the priority-based CPU scheduling mechanism. Some real-time works [19], [21], [22] have also handled other resources such as memory, file system, and network I/O. However, none of them have been done in focusing on real-time issues on an audio device.

RTDroid [10] explores core components of Android including *Alarm Manager*, *Message Handler*, and *Sensor Manager* for real-time support on commodity mobile devices. It then provides a novel architecture for priority-based scheduling of requests on various kinds of resources and functionalities. Especially, it further improves the real-time capability of Android by porting RTLinux and FijiVM on it. However, its resource management scheme is not suitable to satisfy various real-time characteristics of audio requests. For example, it does not take their acoustic requirements into account audio request scheduling, i.e., it just regards an audio device as a single preemptible resource. In addition, during dispatching audio requests, a playback is delayed even decreasing the QoS of mobile sound applications. Thus, for real-time supports on an audio device, we need to carefully look at audio requests and their playback procedures as this work does.

IX. Conclusion

In this paper, we describe the SounDroid framework which manages an audio device for supporting real-time characteristics of newly-emerging applications, i.e., mobile sound applications. SounDroid is basically designed to overcome several challenges of existing platforms such as Android with deeply understanding not only requirements of various audio requests and but also playback mechanisms. Capturing this, it proposes novel scheduling mechanisms, EDF-V and AFDS, which fits into the unique acoustic constraints. In addition, SounDroid supports the predictable use of an audio device by optimizing the audio request dispatching procedures. Our simulation and experimental results show that SounDroid effectively manages an audio device, while satisfying real-time characteristics of audio requests. It, especially, provides a significant enhancement in the quality-of-service of real-world mobile sound applications, which was previously difficult to achieve on existing mobile platforms. Consequently, we believe that our audio device management techniques would be synergistic to mobile platforms, enriching user experience through the improved real-time support for mobile sound applications.

X. Acknowledgments

This work was supported in part by BSRP (NRF-2010-0006650, NRF-2012R1A1A1014930), KEIT(2011-10041313), NCRC (2010-0028680), and IITP (B0101-15-0557) funded by the Korea Government (MEST/MSIP/MOTIE).

References

[1] K. Liu, X. Liu, and X. Li, "Guoguo: Enabling Fine-grained Indoor Localization via Smartphone," in *MobiSys*, 2013.
[2] S. Gupta, D. Morris, S. N. Patel, and D. Tan, "SoundWave: Using the Doppler Effect to Sense Gestures," in *SIGCHI*, 2012.
[3] M. T. I. Aumi, S. Gupta, M. Goel, E. Larson, and S. Patel, "DopLink: Using the Doppler Effect for Multi-Device Interaction," in *UbiComp*, 2013.
[4] K.-Y. Chen, D. Ashbrook, M. Goel, S.-H. Lee, and S. Patel, "AirLink: Sharing Files Between Multiple Devices Using In-Air Gestures," in *UbiComp*, 2014.
[5] Z. Sun, A. Purohit, R. Bose, and P. Zhang, "Spartacus: Spatially-Aware Interaction for Mobile Devices Through Energy-Efficient Audio Sensing," in *MobiSys*, 2013.
[6] C. Peng, G. Shen, Y. Zhang, Y. Li, and K. Tan, "BeepBeep: A High Accuracy Acoustic Ranging System using COTS Mobile Devices," in *SenSys*, 2007.
[7] Z. Zhang, D. Chu, X. Chen, and T. Moscibroda, "SwordFight: Enabling a New Class of Phone-to-Phone Action Games on Commodity Phones," in *MobiSys*, 2012.
[8] J. Qui, D. Chu, X. Meng, and T. Moscibroda, "On the Feasibility of Real-Time Phone-to-Phone 3D Localization," in *SenSys*, 2011.
[9] H. Kim, S. Lee, J.-W. Choi, H. Bae, J. Lee, J. Song, and I. Shin, "Mobile Maestro: Enabling Immersive Multi-Speaker Audio Applications on Commodity Mobile Devices," in *UbiComp*, 2014.
[10] Y. Yan, S. Cosgrove, V. Anand, A. Kulkarni, S. H. Konduri, S. Y. Ko, and L. Ziarek, "Real-Time Android with RTDroid," in *MobiSys*, 2014.
[11] K. Jeffay, D. F. Stanat, and C. U. Martel, "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks," in *RTSS*, 1991.
[12] C. Ekelin, "Clairvoyant Non-Preemptive EDF Scheduling," in *ECRTS*, 2006.
[13] Samsung. GroupPlay, http://content.samsung.com/us/contents/aboutn/groupPlay.do.
[14] S. B. Shneiderman and C. Plaisant, *"Designing the user interface"*. Pearson Addison Wesley, 2005.
[15] J. Blauert, *"Spatial Hearing: The Psychophysics of Human Sound Localization"*. The MIT Press, revised edition, 1997.
[16] Microsoft. Kinect, http://www.microsoft.com/en-us/kinectforwindows.
[17] R. Y. Litovsky, H. S. Colburn, W. A. Yost, and S. J. Guzman, "The precedence effect," *The Journal of the Acoustical Society of America*, vol. 106, no. 4, pp. 1633–1654, 1999.
[18] D. Hart, J. Stultz, and T. Ts'o, "Real-Time Linux in Real Time," *IBM Systems Journal*, vol. 2, no. 47, pp. 207–220, 2008.
[19] F. Pizlo, L. Ziarek, E. Blaton, P. Maj, and J. Vitek, "EuroSys," in *ECRTS*, 2010.
[20] RTEMS, http://www.rtems.org.
[21] H. Kim, M. Lee, W. Han, K. Lee, and I. Shin, "ACIOM: Application Characteristics-Aware Disk and Networkd I/O Manamegment on Android Platform," in *EMSOFT*, 2011.
[22] Y.-H. Wei, Q. Leng, S. Han, A. K. Mok, W. Zhang, and M. Tomizuka, "RT-WiFi: Real-Time High-Speed Communication Protocol for Wireless Cyber-Physical Control Applications," in *RTSS*, 2013.