APPROACHES FOR CONSTANT AUDIO LATENCY ON ANDROID

Rudi Villing,
Department of Electronic Engineering,
Maynooth University
Ireland
rvilling@nuim.ie

Victor Lazzarini,
Sound and Digital Music Research Group,
Maynooth University
Ireland
vlazzarini@nuim.ie

Dawid Czesak, Sean O'Leary
Department of Electronic Engineering,
Maynooth University
Ireland
dczesak@nuim.ie

Joseph Timoney
Sound and Digital Music Research Group,
Maynooth University
Ireland
jtimoney@nuim.ie

ABSTRACT

This paper discusses issues related to audio latency for realtime processing Android OS applications. We first introduce the problem, determining the difference between the concepts of low latency and constant latency. It is a well-known issue that programs written for this platform cannot implement low-latency audio. However, in some cases, while low latency is desirable, it is not crucial. In some of these cases, achieving a constant delay between control events and sound output is the necessary condition. The paper briefly outlines the audio architecture in the Android platform to tease out the difficulties. Following this, we proposed some approaches to deal with two basic situations, one where the audio callback system provided by the system software is isochronous, and one where it is not.

1. INTRODUCTION

The support for realtime audio on the Android platform has been shown to suffer from significant latency issues [1]. Although support for lower latency audio paths has been provided in later versions of the operating system, this provision can vary significantly from device to device. One of the difficulties here, as identified by its developers, is that there is no uniformity in the deployment of the system, which has to be able to adapt to a variety of vendor-designed hardware set-ups. Its competitor, iOS, on the other hand, has been built to run specific devices, which have much less variance, and, for this reason, it can deliver a better support for realtime audio to developers.

While low latency is critical for certain applications, for others, it is possible to design systems around a moderate amount of audio delays. In this case, the next requirement tends to be constant latency. In other words, we might not be worried if a given sound event is to be delivered a certain number of milliseconds in the future, but we would like to be able to predict with a certain degree of precision what this delay will be. Generally speaking, constant latency is guaranteed in a working full-duplex audio configuration, with respect to the input signal, since any modulation of this delay would be associated with sample dropouts.

The situation is, however, more complex when we are trying to synchronise audio output to external control inputs, operating asynchronously to the audio stream (fig.1). This situation typically arises, for instance, when trying to synchronise touch events, or responses to sensors, with specific parameter changes (e.g. onsets, pitch changes, timescale modifications). In this case, constant latency is not a given: it depends on the audio subsystem implementation. On Android, the audio subsystem is vendor-implemented, and, as noted, can vary significantly from device to device. For certain devices, achieving constant latency with regards to an asynchronous control is not much more than a matter of using a common clock, as the audio subsystem callback mechanism is tightly chained to the sample stream output (itself synchronised to a regular clock). On other devices, the situation is more complex, as there is no guarantee that the callback functions will occur with only small amounts of jitter (with regards to a common system clock). Furthermore, in these cases, stream time is not reported with a good degree of accuracy.

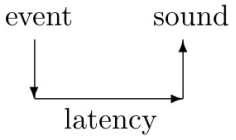

Figure 1. *Event-to-sound latency*

In this paper we will look at approaches to obtaining constant audio output latency with regards to external asynchronous control events. Constant latency is required when we need to estimate the exact time of the change of parameters in the sound stream in response to a control input. For instance, if we want to try and synchronise musical beats to a step detection algorithm, we will need to know how the time between the sending of a tempo adaptation command and the appearance of the effect at the output.

The paper is organised as follows: an overview of the Android audio architecture is provided, which will identify the difficulties with latency control, and identify the various sources of delays in the system. This will be followed by an examination of approaches to achieving constant latency in the two cases mentioned above: isochronous and anisochronous callback mecha-

nisms. The paper will demonstrate how it is possible to minimise latency jitter in these two situations, describing the algorithms employed and their results. The paper concludes with a brief discussion of the applications for the principles outlined here and some indications of future work.

2. ANDROID AUDIO SYSTEMS

The audio implementation in Android is built on a number of layers [2]. The lowermost components are not exposed to developers, and they can only be manipulated indirectly via the topmost application programming interfaces (APIs), of which there are a few. High-level development for Android uses a version of the Java language, and the simplest APIs are provided as part of the Software Developer Kit (SDK). The most flexible of these is AudioTrack, which provides a blocking-type audio writing functionality (synchronous), but it is still considered not enough low-level for any serious audio application.

For this, Android provides a C API based on the OpenSL ES 1.0.1 specification [3] in their Native Developer Kit (NDK). The NDK is a collection of C/C++ APIs that can be used to build dynamic modules for Java-based applications, whose functionality is accessed through the Java Native Interface. The NDK allows developers to port large portions of C/C++ code, and in the case of audio applications, to bypass the performance problems associated with Java garbage collection. However, it is important to note that both the Java AudioTrack and the NDK OpenSL ES APIs use the same underlying audio implementation. The documentation is clear in saying that using the latter does not guarantee better performance in terms of audio latency, although in some systems a "fast" audio mixer signal path might be available to NDK-based applications [4].

2.1. OpenSL ES

The OpenSL ES API provides the means to open audio streams for input and output, and to access these asynchronously via a callback mechanism. This method is generally the standard in audio APIs across platforms, where the developer provides the code to fill in audio buffers at the times requested by the system.

For the developer, this gets exposed via an enqueue mechanism: after the device is initialised and opened, the developer starts the process by enqueuing a single buffer, and registering a callback. This is invoked by the system, according to the specification, when a new buffer is required, and so the callback will include code to generate the audio and to enqueue it.

The specification does not say anything in relation to the timing of these callbacks. In others words, depending on the implementation, these can either be relied upon to happen at regular times (isochronous behaviour) or not (anisochronous). Depending on this, different strategies are required for the implementation of constant latency.

OpenSL also provides some means of polling the system for the current stream time. However, it is not clear from the specification how this should be implemented, and to which point in the audio chain it refers. In reality, we have observed that the information obtained is not reliable across different devices, so it is not always usable.

2.2. Lower levels

The implementation of OpenSL ES sits on top of the AudioTrack C++ library, which is also the backend of the Java AudioTrack API, as noted above. Although the two libraries share the name, they are actually distinct, the Java API occupying the same level as OpenSL on top of the C++ library. The following is a description of the lower-level implementation in Android version 5.0 (fig.2).

The actual code implementing the OpenSL exposed functionality is a thin layer. The enqueueing code simply places a pointer to the user-provided memory in a circular buffer. The data in this memory location is consumed by an AudioTrack function, which copies it into its own buffer. Once all supplied data is read, it issues the user-supplied callback, which should enqueue more data. The OpenSL specification asks for users to implement double buffering of audio data, ie. enqueueing one buffer while filling a second one. However, the implementation is clear: the enqueueing is only requested when the buffer data has been copied completely. Thus double buffering is not really a necessity.

AudioTrack shares its buffer memory with the next lower level service, AudioFlinger. This is responsible for mixing the audio streams in its own playback thread and feeding the Hardware Abstraction Layer (HAL), which is the vendor-implemented part of the code that communicates with the audio drivers running in the Linux kernel. Some devices implement a feature provided by AudioFlinger to reduce latency, a "fast mixer track", which, when available, is used if the developer employs a specific buffer size and sample rate dictated by the hardware, via OpenSL. The presence of this feature can be enquired via a Java call provided by the AudioTrack API, which also allows developers to obtain the native parameters for it. Surprisingly, it is not possible to do this via the NDK.

The consumption of audio data is ultimately linked to the HAL implementation, and so the timing behaviour of the callback mechanism is influenced by this. The HAL also is supposed to supply the information regarding stream time, which is exposed by the OpenSL ES, thus its reliability is related to how well this is implemented there.

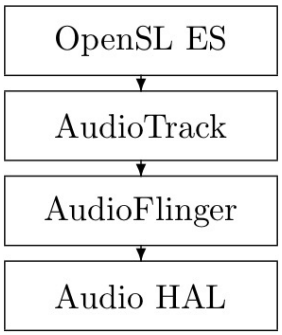

Figure 2. *The Android audio stack*

2.3. Achieving low and constant latency

It is clear from this discussion, that any approach to reducing latency is highly dependent on the hardware involved. Some recommendations have been provided, such as using the native buffer size and sample rates, but these are not guaranteed to have significant effect on all devices. Avoiding the interference of the

Java garbage collector is also important for constant latency, which in practice means migrating all relevant code from the SDK level to the NDK. In general, an analysis of a device's behaviour in relation to its callback timing appears to be needed for a full assessment of its particular latency improvement potential (both in terms of low and constant characteristics).

3. CONSTANT LATENCY APPROACHES

To achieve approximately constant latency, we first evaluate the most basic approach using OpenSL ES, namely enqueueing sound data in the OpenSL buffer queue as soon as possible after the request occurs. Thereafter, we examine approaches which instead estimate the play head position in the output stream and enqueue sound data at a fixed delay relative to that play head position.

All positions are defined relative to the start of the audio data stream. We define the enqueued position as the position within the data stream corresponding to the last sample enqueued to date. We define the true play head position as the position in the data stream corresponding to the sound that is currently being output from the device. In Android devices the delay between the true play head position and the enqueued position ranges from a few tens of milliseconds to more than 100 ms for some devices [1]. In general, if this delay is constant it should be easier to achieve constant latency between sound request and sound output.

To date, we have not encountered any method for measuring this latency in an absolute manner. To estimate the latency some authors measure the audio loopback delay, based on a round-trip path (input to output): an audio signal is fed to the input and the time it takes for it to appear at the output is measured. This can be achieved via the Larsen effect [5], or via a custom loopback audio connection. From this, it is estimated that the output latency is half of this time [6]. Nevertheless, it is not guaranteed that the audio processing path is symmetrical and therefore that the output latency is equal to the input latency.

Other approaches, which are more appropriate to the issue of constant latency, as discussed in sect. 1, include measuring the delay between a touch event on the Android device (or some external input) and the corresponding sound output (as illustrated by fig.1). However, even here there is a difficult to quantify (and usually variable) delay involved in receiving and processing the source event before a sound request is ever issued. Therefore, for this work, we developed an alternative approach based on the relative time at which sound requests were issued and the relative time at which sound outputs occurred.

3.1. Evaluation methodology, equipment and materials

Two different Android devices were used in this work: A Sony Xperia Z2 and a Motorola MotoG (first generation). Both devices were running Android 4.4.4. The most relevant audio properties of both devices are shown in Table 1. These devices were chosen as exemplars of a high end device (the Xperia Z2) and a mid-range device (the MotoG). As there are a great variety of Android devices and capabilities, this work with just two devices should be considered a preliminary examination of the problem space.

Table 1: *Key Audio Properties*

Device	Low latency	Native sample rate	Native buffer size (frames) / duration (ms)
Xperia Z2	No	48000	960 / 20
MotoG	No	44100	1920 / 43.5

A custom Android app was developed which included a C based NDK component and a java component. The NDK component implemented the sound engine as a thin layer on top of OpenSL ES and was responsible for all timing measurements (measured using clock_gettime specifying CLOCK_MONOTONIC). The Java component implemented test runs which consisted of a sequence of 500 sound requests issued at anisochronous intervals between 400–500 ms apart. (This interval was chosen so that even substantial latency jitter would not create any doubt about which sound request and which sound output correspond to one another.) The sound engine logged the precise time (in microseconds) at which each request was made and stored this in a log file along with other data required for subsequent processing.

Each sound request resulted in production of a short, 10 ms, 1000 Hz tone pip (with instant attack and release, limited only by the analogue hardware response). The audio output of the Android device was attached to the line input of a USB sound device (Griffin iMic) and recorded as a RIFF-Wave file (mono PCM, 16 bits, 44100 Hz) on a PC. This soundfile was then post-processed with threshold based onset detection to determine the times (within the audio file) at which tone pips occurred.

It is not possible to directly synchronize the clock used to measure sound request times with that used to measure sound onset times using standard Android devices. Therefore it was necessary to convert from raw times using these unsynchronized clocks to times that can be compared. To achieve this, all raw request times from the log file were made relative to the time of the first request (by subtracting the first request time from each of them). Similarly all raw sound onset times from the audio file were made relative to the time of the first sound onset (by subtracting the first sound onset time from each of them). After this calculation the first request and sound onset time are both zero and this allows subsequent request and onset times to be compared. The initial real world latency between the first request and the first sound onset cannot be measured (because the clocks cannot be synchronised) and appears as zero. If the latency is constant, all subsequent sound onsets occur at precisely the same time offsets as the corresponding requests and the time difference between them should be zero. In contrast, if there is latency jitter, the sound onset times will differ from the request times by an amount equal to the jitter.

In the subsequent evaluations it was noticed that there was a slight mismatch between the clock used to time requests on the Xperia Z2 and the sample clock in the USB microphone audio input to the PC. This mismatch caused the two clocks to drift apart by 1-2 ms over the duration of a test run. Therefore Xperia Z2 plots shown in the following sections have been de-trended to compensate for the drift which is unrelated to the latency algorithms being evaluated.

3.2. Latency using the Next Buffer approach

The most basic approach to approximating constant latency with OpenSL ES is to enqueue audio data as soon as possible after it is requested. In other words:

$$insertPos = enqueuedPos \quad (1)$$

where *insertPos* is the insertion position of the sound in the data stream and *enqueuedPos* is the end of data stream position after the most recent callback preceding the request has completed.

In general the Next Buffer approach can be expected to yield at least one buffer duration jitter as the request may occur just before a callback is due, resulting in a very short request to enqueue delay, or just after a callback has completed, resulting in a longer request to enqueue delay. For small buffer sizes (for example 20 ms) this jitter may be acceptable, but for larger buffer sizes (for example the 43 ms of the MotoG) the jitter can become audible.

Furthermore, if OpenSL callbacks occur at isochronous intervals, then two requests occurring within one buffer duration of each other will never be enqueued more than one buffer apart (even if they occur just before and just after a callback). If, however, callbacks do not occur at isochronous intervals (that is, some inter-callback intervals are shorter than others) then it is possible that two requests occurring within one buffer duration of each other in real time may be enqueued more than one buffer apart resulting in jitter that is even greater than one buffer duration in this case.

The results for the Xperia Z2 are shown in Figure 3.

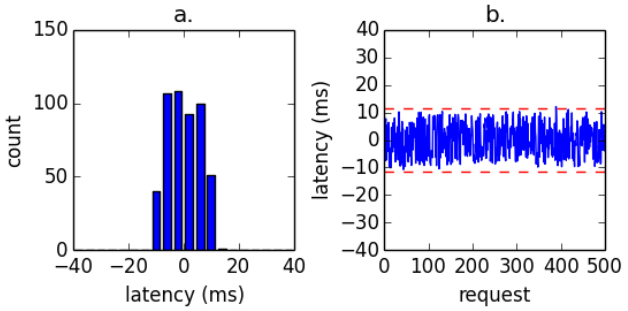

Figure 3 Relative latencies using Next Buffer on the Xperia Z2: (a) histogram, (b) values. Dashed lines show two standard deviation boundaries.

In this case the latency jitter is reasonable and limited to a range that is just larger than one buffer duration (-10.7 to 11.4 ms). Closer investigation showed that OpenSL callback times were nearly isochronous for this device.

Figure 4 shows the MotoG results and in this case the latency jitter is much larger than the Z2.

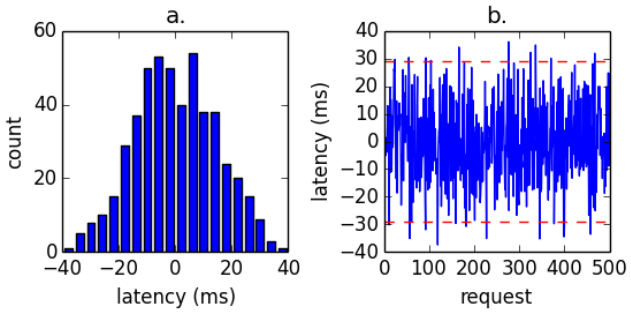

Figure 4 Relative latencies using Next Buffer on the MotoG: (a) histogram and (b) values. Dashed lines show two standard deviation boundaries.

In fact the latency jitter for the MotoG is even larger than one buffer duration (95% of the values were within a 58 ms range). This range is really too large and can result in audible jitter. The main cause for the jitter appears to be that the callback is not invoked at regular intervals on the MotoG. Instead a typical sequence of callback intervals measured in milliseconds might be: 40, 60, 20, 60, 40, 40, 80, 30, and so on.

Therefore it appears that the Next Buffer approach is only suitable if two conditions are satisfied by the device: (1) the native buffer size is relatively small, and (2) callbacks are invoked at isochronous intervals. For devices which do not satisfy these requirements, such as the MotoG, another approach is required.

3.3. Latency using the OpenSL position

Rather than attempting to enqueue a sound as soon as possible after the request and suffering the jitter that results it would be better if it was possible to enqueue the sound such that it was output a constant latency after the request. If the current play head position is known, then constant latency can be achieved by adding the sound to the data stream at some fixed delay relative to the play head position. This fixed delay must be large enough that the desired insert position of the sound is never earlier than the already enqueued position despite any callback jitter.

As discussed previously, the OpenSL ES API defines a function to read the current play head position and this is direct estimate of the play head position which may be used to calculate an appropriate insert position for the requested sound data as follows:

$$insertPos = requestSLPos + fixedDelay \quad (2)$$

where *requestSLPos* is the OpenSL position at the time of the request and the fixed delay (which was determined empirically for each device) guarantees that the insertion position is never earlier than the end of data already enqueued. This fixed delay does not appear in subsequent plots due to the relative nature of the times used.

Although the relative latencies for the Xperia Z2 using the Next Buffer approach were already quite good, we decided to evaluate the achievable latency jitter using the OpenSL position also. The results are shown in Figure 5.

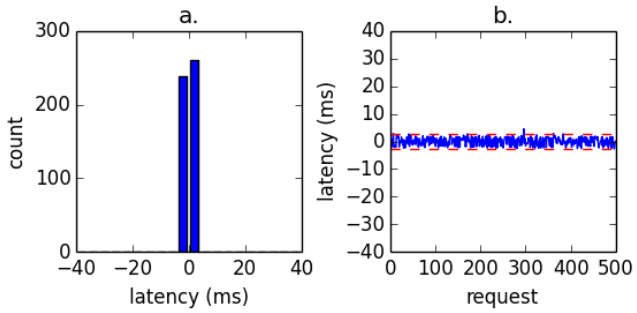

Figure 5 Relative latencies using OpenSL position on the Xperia Z2: (a) histogram, (b) values. Dashed lines show two standard deviation boundaries.

It is clear that inserting sounds a fixed delay after the OpenSL position (measured at the time of request) has substantially reduced the latency jitter (95% of the values are in 5.6 ms range) relative to the Next Buffer approach.

When this latency approach was applied to the MotoG the results improved somewhat relative to the NextBuffer approach but were still much worse than the Xperia Z2 as can be seen in Figure 6.

In this case, 95% of the latencies are contained within an 36.8 ms range (compared to a 58 ms range previously). This is still a relatively large range and, unlike the Xperia Z2, there are a number of outlier values which substantially depart from the typical range.

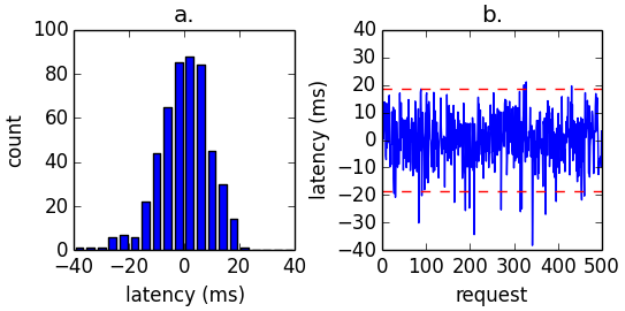

Figure 6 Relative latencies using OpenSL position on the MotoG: (a) histogram, (b) values. Dashed lines show two standard deviation boundaries.

Closer examination of OpenSL positions on the MotoG indicated that the intervals between OpenSL positions on consecutive requests did not match the intervals between the real time of those requests very well. Further investigation indicated that the API call to read the OpenSL position appears to read a cached position value which is not always up to date rather than reading the real position from the low level hardware or driver.

In some cases reading the OpenSL position twice 3 ms apart suggested that the position had not advanced at all whereas at other times a similar pair of requests separated by just 3 ms of real time indicated that OpenSL position had changed by as much as 30 ms. Therefore a better approach is still required for the MotoG and similar devices. As noted in section 2, the implementation of the position polling in Android devices cannot be relied on, and thus this approach is not general enough for all devices.

3.4. Latency using the Filtered Callback Time

Results to date indicated that the interval between callbacks on the MotoG was not even approximately constant. Nevertheless the mean callback interval precisely matches the buffer duration. This is to be expected: if the mean callback interval were any larger it would mean that the buffer level (the difference between the true play head position and the enqueued position) would gradually decrease and underflow as new buffers would be added more slowly than they were consumed. Similarly if the callback interval were any smaller than the buffer duration, the buffer level would increase until overflow occurred. Therefore, the fact that the callback intervals are distributed around a predictable mean value suggests a possible approach to achieving constant latency.

The callback intervals recorded for the MotoG are suggestive of a low level task which polls the state of the low level audio buffers approximately once every 20 ms and initiates a callback if the buffer level is less than some threshold. We assume that, for various reasons (including the mismatch between the polling rate and the buffer duration), the buffer level is different for consecutive callbacks and therefore there is variable latency between the true play head position and the next data enqueued by the callback. Our approach is to estimate the times at which callbacks would have been invoked if they were invoked at the same buffer level each time and therefore invoked at isochronous intervals.

The basic approach is to record the times at which callbacks occur and filter these to estimate the time at which a constant latency callback would have occurred.

The technique used to estimate the filtered callback time was double exponential smoothing [7] as defined by the following standard equations:

$$s(n) = \alpha x(n) + (1-\alpha)\left[s(n-1) + b(n-1)\right] \quad (3)$$

$$b(n) = \beta\left[s(n) - s(n-1)\right] + (1-\beta)b(n-1) \quad (4)$$

where $s(n)$ defines the smoothed output (the filtered callback time) at time n, $x(n)$ defines the input (the callback time), and b(n) defines the trend in the data (corresponding to the smoothed interval between callbacks in this case). The parameters α and β determine the smoothing factors and rate of convergence of the filter.

We initialize $s(0)$ to be $x(0)$ and $b(0)$ to be the native buffer duration. On each callback, (3) provides an estimate of the filtered callback time based on the linear trend up to that point. Thereafter (4) updates the value of the trend. The trend should not change much over time but using these equations allows the algorithm to adapt to slight mismatches between the system clock used by Android and the sample clock used by the audio hardware. (We previously used simple exponential smoothing [7] with a fixed estimated interval between callbacks but preliminary results showed that the two clocks soon drifted audibly apart using this approach.)

The filtered callback time is calculated at the start of each callback before any new data has been enqueued. Therefore the play head position at the start of the callback may be estimated (save for a constant offset) as

$$cbPlayPos = enqueuedPos + \left(cbTime - filteredCBTime\right) \quad (5)$$

where *enqueuedPos* is the end of data stream position after the preceding callback completed, *cbTime* is the time at which the current callback was invoked, and *filteredCBTime* is the time at which the current callback should have been invoked if it were invoked exactly when the buffer level drained to its mean level. This estimate of the play head position incorporates a constant offset (the mean buffer level) which should be subtracted to get a more accurate estimate of the true play head position. In our approach, this offset is incorporated as part of the fixed delay added later in the algorithm.

In the sound request a new estimate of the play position estimate is made as follows:

$$playPos = cbPlayPos + \left(reqTime - cbTime\right) \quad (6)$$

where *reqTime* is the time at which the request was made and other values are as already defined based on the most recent callback preceding the request. Therefore this step accounts for the time that has elapsed since the play position was last estimated. Thereafter, the insertion point may finally be determined as

$$insertPos = playPos + fixedDelay \quad (7)$$

Using the filtered callback time technique to achieve low latency on the Xperia Z2 yielded results that were essentially identical to those obtained using the OpenSL position. For that reason they are not shown here. The MotoG results, however, did differ from those obtained using the OpenSL position as shown in Figure 7.

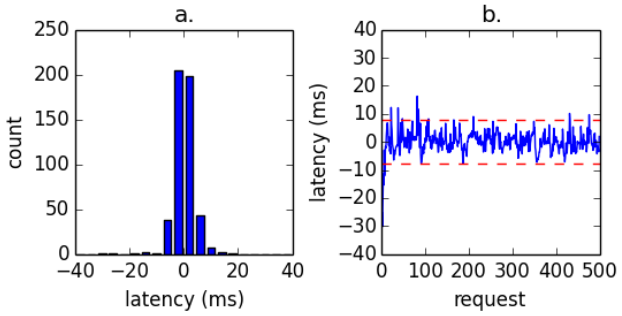

Figure 7 Relative latencies using Filtered Callback Time on the MotoG: (a) histogram, (b) values. Dashed lines show two standard deviation boundaries.

These results show a clear improvement over those obtained using the Next Buffer and OpenSL position techniques. In this case 95% of the relative latencies are within a 16 ms range resulting in jitter that should be inaudible to most, if not all, listeners. The results also indicate a number of outlier values, but these occurred at the start of the test run during the convergence period of the filter.

Despite the reduction in relative latency jitter that has been achieved there is still some residual jitter remaining. The detailed sources of this residual jitter are currently unknown but it is possible that at least some of it is due to scheduling jitter within the operating system since there are several interacting threads involved in the process of audio output.

4. CONCLUSIONS AND FUTURE WORK

The work described in this paper described the motivation for and difficulty in achieving constant latency on Android devices as many devices, even recent devices, do not implement the low latency feature now supported by the Android operating system.

We made two main contributions: we measured the relative latency jitter achieved by two Android devices using OpenSL ES and we proposed two schemes which specifically aimed to reduce this latency jitter and approach constant latency output of audio events.

The commonly used Next Buffer scheme was investigated and found to be sub-optimal in achieving constant latency for all devices (although for low latency devices it may be sufficient). Using the Open SL position to achieve constant latency worked well for one phone (the Xperia Z2) but badly for another (the MotoG) and we conclude that the success of this scheme is highly dependent on the quality of the OpenSL ES implementation on the device. Finally we proposed a novel scheme based on filtering the callback times to estimate the play head position and showed that this scheme worked quite well even when the underlying OpenSL ES implementation appeared to have several shortcomings (as was the case for the MotoG).

Notwithstanding the successes reported in this paper, preliminary investigations with additional Android devices have indicated that there seem to be quite a variety of different OpenSL ES implementations and buffer strategies used by different device manufacturers. Consequently, it appears that even the techniques described above must be supplemented by additional techniques if constant latency is to be achieved on all devices. In the future, as part of our ongoing work, we plan to investigate additional techniques and a method of selecting the best technique for a particular device (if no single technique works for all devices).

5. ACKNOWLEDGMENTS

This research was supported by BEAT-HEALTH, a collaborative project (FP7-ICT) funded by the European Union. For more information, please see http://www.euromov.eu/beathealth/.

6. REFERENCES

[1] G, Szanto and P. Vlaskovits, “Android’s 10 Millisecond Problem: The Android Audio Path Latency Explainer” , Available at http:// superpowered.com/androidaudiopathlatency/ #axzz3bzkezdMg" Accessed June 3, 2015

[2] Android Open Source Project. Android Source Code. Available at https://source.android.com. Accessed April 29, 2015.

[3] Khronos Group Inc., The OpenSL ES 1.0.1 Specification. Available at https://www.khronos.org/registry/sles/specs/OpenSL_ES_Specification_1.0.1.pdf.. September 2009.

[4] "OpenSL ES for Android". Android NDK Documentation, "file://android-ndk-r10/docs/Additional%20library%20docs/opensles/index.html". Available from http://developer.android.com/ndk/downloads/index.html

[5] A. Larsen, “Ein akustischer Wechselstromerzeuger mit regulierbarer Periodenzahl für schwache Ströme”. Elektrotech. Z., ETZ 32, pp. 284–285, 1911.

[6] Android Open Source Project, "Audio Latency Measurements". Available at https://source.android.com/devices/audio/latency_measurements.html. Accessed June 11, 2015.

[7] NIST/SEMATECH e-Handbook of Statistical Methods, http://www.itl.nist.gov/div898/handbook/. Accessed June 11, 2015