On the performance of real-time DSP on Android devices

André Jucovsky Bianchi
Computer Science Department
University of São Paulo
ajb@ime.usp.br

Marcelo Queiroz
Computer Science Department
University of São Paulo
mqz@ime.usp.br

ABSTRACT

With the increasing availability of mobile devices comes the possibility of using (relatively) cheap, wireless hardware embedded with plenty of sensors to perform real-time Digital Signal Processing on live artistic performances. The Android Operating System represents a milestone for mobile devices due to its lightweight Java Virtual Machine and API that makes it easier to develop applications that run on any (supported) device. With an appropriate DSP model implementation, it is possible to use the values of sensors as input for algorithms that modify streams of audio to generate rich output signals. Because of memory, CPU and battery limitations, it is interesting to study the performance of each device under real-time DSP conditions, and also to provide feedback about resources consumption as basis for (user or automated) decision making related to the devices' use. This work presents an object oriented model for performing DSP on Android devices, and focuses on measuring the time taken to perform common real-time DSP tasks (such as input/output, FIR/IIR filtering and computing FFTs of varying sizes) and estimating the maximum order of FIR filters that can be run on different combinations of software and hardware. We obtain and discuss statistics of some specific combinations of device model and operating system version, and our approach can be used on any Android device to provide the user with important information that can aid aesthetic and algorithmic decisions regarding real-time DSP feasibility.

1. INTRODUCTION

Modern mobile devices provide input and output of audio and video signals, various sensors to measure acceleration, orientation, position and luminosity (among many others) and wireless communication standards, thus becoming a handy tool for artistic and scientific use. Despite that, the actual computational power of mobile devices lies often below users' expectations, and a systematic measurement of devices' performance for Digital Signal Processing (DSP) under real-time constraints brings some enlightenment on the actual capabilities of such devices.

With a market share of around 50%, Android OS – and its fully free software counterparts – allow for easy development and deploying of applications that can make use of an extensive API to access and control devices' functionalities. This article describes an implementation of an object-oriented real-time DSP model using the Android API and the use of this model to generate reports about devices' performance.

The main interest of our research is to get a feel of devices' computational intensity, i.e. the amount of computation that can be done in a certain amount of time. To determine devices' performance, we devised two strategies. In the first one we focus on time measurement for common real-time DSP tasks such as reading samples from an audio signal, writing them for playback, and running specific DSP algorithms over blocks of samples. With this, we are able to determine the amount of (concurrent) computation time the device has available to perform a DSP cycle over a block of samples, and also the percentage of the DSP cycle that specific algorithms occupy in different combinations of hardware (device) and software (Android API). In the second strategy, the device is stressed as a means to estimate the maximum amount of computation feasible in each DSP cycle. We ran stress-tests for different DSP block sizes using random FIR filters with an increasing number of coefficients, and so could determine the maximum order of filters that can be run on each device setup.

1.1 Related work

Optimizations on the audio signal routing on the media layers of the Android operating system have been proposed to diminish power use and save batteries [1]. By creating a signal route manager and using (software and hardware) compressed audio capabilities, the authors were able to reduce in 20% the power consumption for some specific audio processing tasks, when comparing to the standard Android 2.3 audio routing design. The descent onto the internals of the system is different from our approach to obtain device performance indicators on the application level. Despite that, it would be possible to use their contributions to the audio framework in our application, and with this we could find out if their model also allows for more computational intensity on the application level.

Recent efforts have been mixing the Android platform with traditional real-time DSP software, such as Csound [2] and Pure Data [3]. Both of these approaches make use of the JNI [1] to mix C/C++ code with Java code to wrap li-

[1] https://developer.android.com/sdk/ndk/overview.html

braries and allow these environments to run on the Android OS. Our approach, on the other hand, uses pure Java code to implement a minimal GUI and real-time DSP model, and the use of native code represents a further step for development and performance measurement.

The use of native code does not automatically imply better performance because it increases application complexity and also has a cost associated with calls to non-Java code. There are works that aim to find performance differences between Java and native code on different scenarios, and conclude that native code is indeed worse for some applications [4]. Nevertheless, for real-time signal processing an implementation and comparison with native code is needed and will be considered on the last section of this text as future work.

1.2 Text organization

On Section 2, we describe an implementation that permits the live use of a small DSP framework as well as device stressing and report generation. The tool allows for the variation of some DSP parameters (block size, sound source, DSP algorithm and scalar input parameters) while providing numeric and visual feedback about the device's performance (tasks mean time, CPU usage and busy fraction of DSP cycle). With this kind of feedback, it is possible for the user to take the device's performance into account during live performances, and so to derive aesthetic and algorithmic decisions based on the available computational power.

On Section 3 we describe the results obtained, and on Section 4 we make a discussion of the results and comment on some steps for future work.

2. METHODS

To get a feel of what it is like to use Android devices for real-time DSP, we have set up an environment to run arbitrary algorithms over an audio stream divided into blocks of N samples, allowing for the variation of algorithm parameters during execution. The software is an Android application [2] (using API level 7 [3]) that consists of a GUI and a DSP object model that keeps track of timing of specific tasks as they are carried out. The GUI allows for live use of the processing facilities and also for automated testing of DSP performance. Sound samples can be obtained directly from the microphone or from WAV files, and the DSP block size can be configured to be $N = 2^i$ with $0 \leq i \leq 13$ [4].

The main goal of this investigation is to measure the performance of the device for real-time DSP, which we split into (1) the amount of concurrent computation time available for signal processing, and (2) the amount of computation time required for performing common DSP tasks. In order to achieve this, the program keeps track of sample read and write times, DSP algorithm execution times and DSP callback periods. With this information, it is possible to have a picture of the device's performance by comparing the time taken to perform various tasks with the theoretical time available for DSP computation over a block of N samples ($\frac{N}{R}$ s, if R is the sample rate in Hz).

To investigate devices' performance on common DSP tasks, we devised three processing scenarios. In the first scenario, no signal modification is performed, only input/output loopback time is measured; in the second one, we compute a simple reverb effect using an all-pass filter; and in the third scenario a one way FFT of the whole sample block is computed, without modifying the input signal. In each scenario the program keeps track of the full DSP callback period (including conversion between PCM and floating point representation, timekeeping code and sample enqueueing for playback), and also the actual DSP algorithm execution time.

Following, we present the object-oriented model developed for the performance measurement of generic real-time DSP algorithms running on an Android environment. After that, we describe the parameter data flow and finally the system statistics we gathered for the analysis.

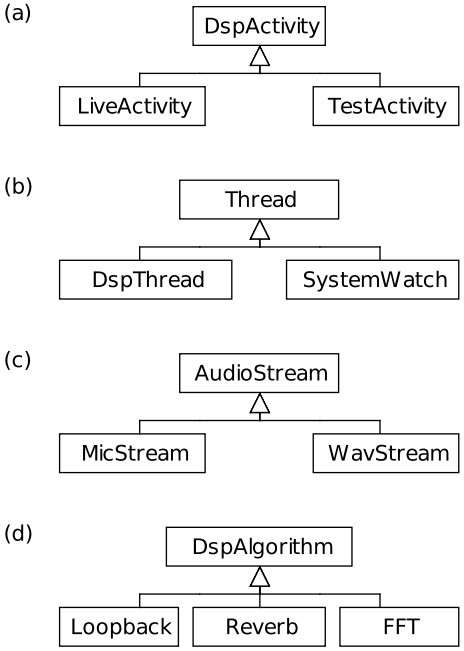

Figure 1. Class diagrams for the main parts of our object model: (a) the program has two GUIs, one for live use and the other for device testing; (b) concurrent threads take care of signal processing and system information gathering; (c) audio signals can be obtained either from the microphone or from raw WAV files; (d) audio streams are modified by DSP algorithms that extend from one common abstract class.

2.1 DSP object model

Class diagrams for the main parts of the object-oriented model we devised can be seen in Figure 1. The application is composed of two (Android) activities: one for live performance and one for automated benchmarking. The

[2] http://www.ime.usp.br/ ajb/DspBenchmarking.apk

[3] API level 7 is compatible with Android OS version 2.1 and higher.

[4] This upper limit is configurable; $N = 2^{13}$ under a 44.1 KHz sampling rate produces a latency of 186 ms, which is very noticeable.

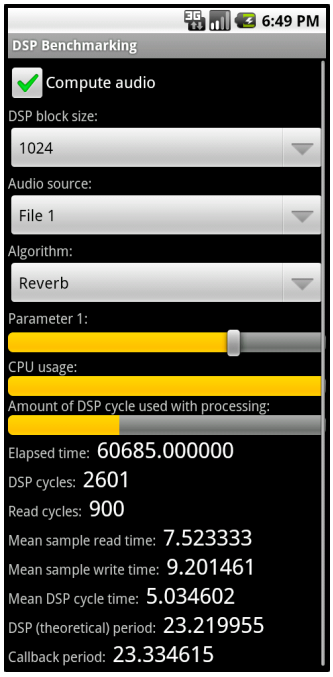

Figure 2. The GUI controlling a live DSP process. The user is able to choose the DSP block size, the audio source (microphone or predefined WAV files) and the DSP algorithm that will be run. Also, slider widgets can provide explicit parameter input, while visual and numerical statistics give feedback about the state of the system.

`LiveActivity` class allows the user to choose the audio signal source as well as alter DSP configuration such as block size, DSP algorithm and processing parameters (see Figure 2). Alternatively, the `TestActivity` class performs a set of automated tests and generates a report file with results that can be further analyzed. Both activities extend a more general class called `DspActivity`.

In our model, DSP activities are responsible for various tasks, one of them being to keep the GUI up to date with information about the device's performance. An instance of `SystemWatch` thread gathers information from the system and delivers it to the `DspActivity`, which in turn alters the GUI to reflect the system state. Also, the activity can combine these values with parameters acquired from the user through sliders, buttons and other visual widgets on the GUI, feeding them to the DSP thread, described below.

Running concurrently with the main activity and the `SystemWatch` thread, an instance of the class `DspThread` schedules a callback method to be executed at every DSP cycle (see Figure 3). The actual signal modification routine is defined by subclasses of the `DspAlgorithm` class. The callback method reads from an audio signal input buffer and writes the result of the processing back to an output buffer when it is finished. The DSP cycle period Δ_N (in seconds) is given by the relation of the sample rate R (in Hz) and the configured DSP block size N (in samples), by $\Delta_N = \frac{N}{R}$. This is also the maximum amount of time that the scheduled callback method can take to write new

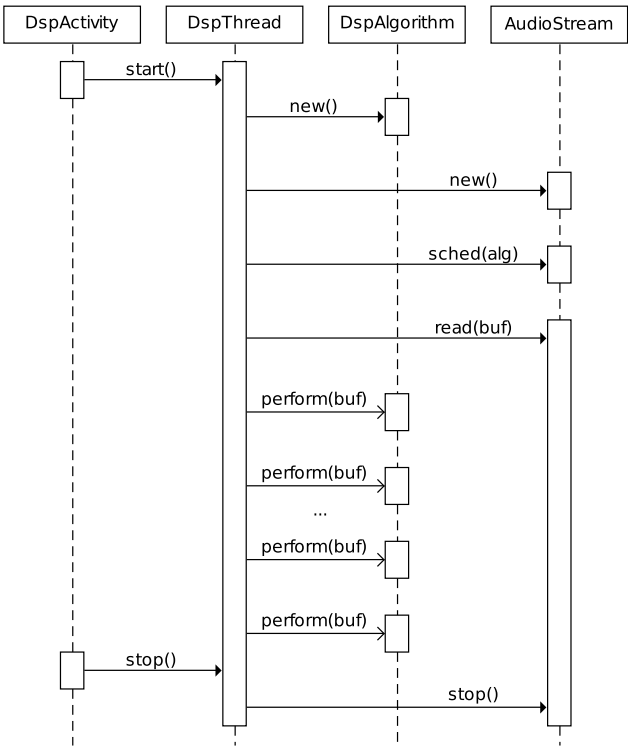

Figure 3. Sequence diagram of the main part of the DSP object model at work. The activity starts a DSP thread, which then instantiates subclasses of `DspAlgorithm` and `AudioStream`. The DSP thread then schedules a perform method that will be called at every DSP cycle, and then starts reading samples from the input buffer. Subsequent asynchronous periodic calls to the algorithm's perform method modify the input signal and write the results to the output buffer.

samples to the hardware queue under real-time constraints.

Every class that extends `DspAlgorithm` has to implement a perform-routine that acts on a buffer to generate the desired effect. The `DspThread` running instance is responsible for instantiating the user selected algorithm and schedules it in such a way that it is called at every DSP cycle to modify data lying on the buffer.

Algorithms can access and produce several parameters. Sliders on the GUI and values of sensors (camera, accelerometer, proximity, etc) can be used as input by calling specific querying methods. User-defined parameters, such as FFT coefficients derived from the analysis of audio blocks, can be periodically output as strings of values over Wi-Fi or bluetooth as well as written to a file.

Regarding input signal sources, it is fairly straightforward to use the Android API to read samples from the device's microphone by instantiating the `AudioRecord` class. On the other hand, API classes for handling media can only be used to play entire files (using Android API's `MediaPlayer` class) or adding sound effects to the overall audio output chain (using Android API's `AudioEffect` class), and cannot be used to access (read and modify) the digital signal on the application level. Because of these constraints, we implemented a WAV parser based on the

standard WAV header specification [5] and specific bit depth and sample rate configurations for our test files (PCM 16-bit samples and 44.100 Hz sample rate).

The following list summarizes the functionality of the main classes of the DSP model:

- **Activities (`LiveActivity`, `TestActivity`):** GUI and threads management.
- **DSP thread (`DspThread`):** signal input/output, callback scheduling and time accounting. Some highlights of this class are:
  - Use of `MicStream` to record from the microphone and `WavStream` to obtain samples from a WAV file (these are blocking operations).
  - Use of `AudioTrack` to write to the audio buffer after processing (this is a non-blocking operation).
  - Conversion of 16-bit short type samples with PCM audio representation to double floating point numbers and back. Some DSP algorithms can be implemented over the integers, but for many applications floating points are needed, and so we consider this as a common task that needs to be taken into account for performance measurement on arbitrary environments.
  - Scheduling of a `DspAlgorithm`'s perform method to run at every DSP cycle using `AudioStream`'s scheduling mechanism.
  - Parameter passing from the GUI to the DSP algorithm and back.
- **System watch thread (`SystemWatch`):** acquires information about the system (CPU usage and sensor values) and feeds the GUI and the DSP algorithms.
- **Algorithms (subclasses of `DspAlgorithm`):** interface for DSP algorithms. Forces a method signature for performing block computations during DSP cycles.

As all this was implemented on top of the Android software stack (i.e. as a Java Android application), usual concerns regarding memory management had to be addressed. Objects are never created unless they are really needed: it is possible, for example, to suspend the DSP processing for changing its characteristics (block size, algorithm, etc) without killing the DSP thread, and then to resume it by rescheduling the DSP callback. The garbage collector also has to be wisely used in order to guarantee that memory will not leak while not interfering with the DSP callback time.

Using the described DSP model, we devised two ways of measuring the performance of each device, which are described on the following sections.

[5] http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html

2.2 Algorithm benchmarking

At each DSP cycle, a scheduled callback is run to process the current block of audio samples. This callback includes the execution of routines for time measurement, conversion between WAV PCM `shorts` representation to `doubles` and back, the actual algorithm's `perform()` routine and sample writing for playback (i.e. writing to the hardware queue). For the purposes of this work, we call the time taken to perform this set of operations the *callback time*. The callback time is thus the time required by a minimal set of (time-measured) DSP operations to perform any desired algorithm, and thus can be compared with the DSP cycle period to determine feasibility of DSP operations.

If the DSP callback time eventually exceeds the DSP cycle period, then the system is unable to generate samples in real-time, i.e. to deliver them for playback before the DSP cycle ends. As in mobile devices there are, in general, strong constraints related to computational power, the first question that arises is if our DSP model (which is Java-based, includes timekeeping code, converts samples back and forth, relies on the API scheduling and operates in the same priority level as common applications) is indeed usable.

To answer this question, we first ran 3 simple DSP algorithms and took the mean times for a series of DSP callbacks with different block sizes. These algorithms are:

- **Loopback:** actually an empty perform method that returns immediately. This takes only the time of a method call and is used to establish the intrinsic overhead of the DSP model.
- **Reverb:** a two-coefficient IIR filter that outputs $y(n) = -gx(n) + x(n-m) + gy(n-m)$ [5].
- **One-way FFT:** A Java implementation of the FFT (which takes $O(n \log n)$ steps, where n is the block size) [6].

These three implementations can give us a feel on what it is like to do real-time DSP on different combinations of Android hardware and software, as we'll see in Section 3.

2.3 Stress tests

Supposing that the mean callback times obtained indeed leave room for more computation, then the next natural question is how much (concurrent) DSP algorithmic computation can actually be performed inside the callback while maintaining real-time generation of output samples. To answer this question we stress-tested the devices using FIR filters with increasing number of coefficients. By comparing the callback time for different sizes of filter with the DSP cycle period, we are able to estimate the maximum number of coefficients for a filter that can be applied to the input (using our DSP model) while maintaining real-time feasibility.

The search for the maximum number of coefficients happens in two phases. During the first phase, the number of coefficients is raised exponentially until the mean callback

Device model	Manufacturer	Commercial name	CPU model	CPU speed	RAM (MB)	Android API
a853	Motorola	Milestone	ARM Cortex A8	550 MHz	256	7 and 8
gti5500b	Samsung	Galaxy Europa	ARM11	600 MHz	140	7
gti9000	Samsung	Galaxy S	ARM Cortex A8	1 GHz	512	10
gti9100	Samsung	Galaxy S II	ARM Cortex A9	1.2 GHz (dual)	1000	10
gtp1000l	Samsung	Galaxy Tab	ARM Cortex A8	1 GHz	512	8
mz604	Motorola	Xoom with Wi-Fi	NVIDIA Tegra 2	1 GHz (dual)	1024	13
r800i	Sony Ericsson	Xperia PLAY	Qualcomm Snapdragon	1 GHz	512	10
tf101	Asus	Eee Pad Transformer	NVIDIA Tegra 2	1 GHz (dual)	1024	15
x10i	Sony Ericsson	Xperia X10	Qualcomm Snapdragon	1 GHz	384	10
xoom	Motorola	Xoom 4G LTE	NVIDIA Tegra 2	1 GHz (dual)	1024	15

Table 1. Table of tested devices. Note that we tested two Motorola Milestone devices with different API levels (both are represented in one line of the table). In the graphics, the Milestone device using Android API version 7 is named "*a853 1*" and the one using API version 8 is named "*a853 2*".

time exceeds the DSP cycle period. At this point we can establish two invariants: (1) that the device is capable of performing FIR filtering with m coefficients, and (2) that the device is not able to perform FIR filtering with M coefficients ($M = 2m$ on the first run). Once this is achieved, then a recursive binary search is performed over the value $k = m + (M - m)/2$, going up and down depending on the performance of the device for each k, to finally converge on a number K which is the maximum number of coefficients that the device is able to compute during one DSP cycle.

Note that the filter equation indeed gives us an upper bound on the number of arithmetic operations that can be performed on a DSP cycle: if $y(n) = \sum_{i=0}^{K-1} \alpha_i x(n-i)$, then the calculation of $y(n)$ requires at least K multiplications, K vector accesses and $K - 1$ additions. Other algorithms can be implemented as extensions of `DspAlgorithms` to obtain statistics and estimate the feasibility of arbitrary operations.

2.4 Experiment design

As discussed above, all performance measurements and stress tests are made by an application started by the user. User interactive assistance is kept at a bare minimum, by starting the experiment and pressing a button to e-mail the results to the authors. As we had only one Android device fully available for development and testing, we decided to launch a call for participation and so could gather test results from 10 devices belonging to students and teachers (making a total of 11 devices). Instructions were sent to stop all applications and turn off all communication to impose an "idle" scenario on every device.

As most of the devices available for our experiment were of strict personal use, we had to keep the experiment under a reasonable time limit. The time required to estimate the maximum size of the filter for a given block size is bounded by the logarithm of K because of the binary search. Despite that, the number of repeated measurements (for which the mean is taken) has to be carefully chosen, in order to keep a reasonable experiment length. To reduce test length, we limited the number of DSP cycles for different block sizes, as can be seen in the following table:

Block sizes	Max # of cycles	Max time (s)
$64 - 512$	1000	11.60
$1024 - 2048$	500	23.21
$4096 - 8192$	100	18.57

Besides that, every stress test with a certain filter size is interrupted after 100 DSP cycles if the device performance has already shown to be poor for this filter size. The result of imposing these constraints is an overall experiment that automatically cycles through all benchmarking algorithms and stress tests, runs for about 25 minutes, and then sends an e-mail report with its results back to the authors. The list of devices that contributed to this research can be seen in Table 1.

3. RESULTS

3.1 Algorithm benchmarking

In Figures 4, 5 and 6 we can see the results of, respectively, the loopback, reverb and FFT algorithms running on different devices with different API versions. In every figure we also plot the DSP cycle period, which corresponds to the maximum amount of time the DSP callback method can take to write new samples to the hardware queue, under real-time constraints.

Since loopback and reverb take linear time with respect to the block size, the patterns of the first two graphs are expected. Even so, these graphs are useful for ranking the devices with respect to computational power, and also for giving a precise account of the time slack available for extra computation. The FFT takes $\mathcal{O}(n \log n)$ time, which does explain the upward tilt seen in Figure 6 on some of the devices. It should be expected that for larger block sizes real-time FFTs would become infeasible on every device.

On faster devices the callback mechanism leaves plenty of room for DSP algorithms such as the reverb filter, and even real-time FFT is feasible for every device except the gti5500b. As for this latter device, the DSP model alone occupies more than half the period of the DSP cycle, even though the real-time reverb filter is feasible; on the other hand, FFT calculation is prohibitive for all block sizes.

Regarding differences between API levels, we can observe that, for the a853 model (the only model with two different API levels among the tested devices) there is a

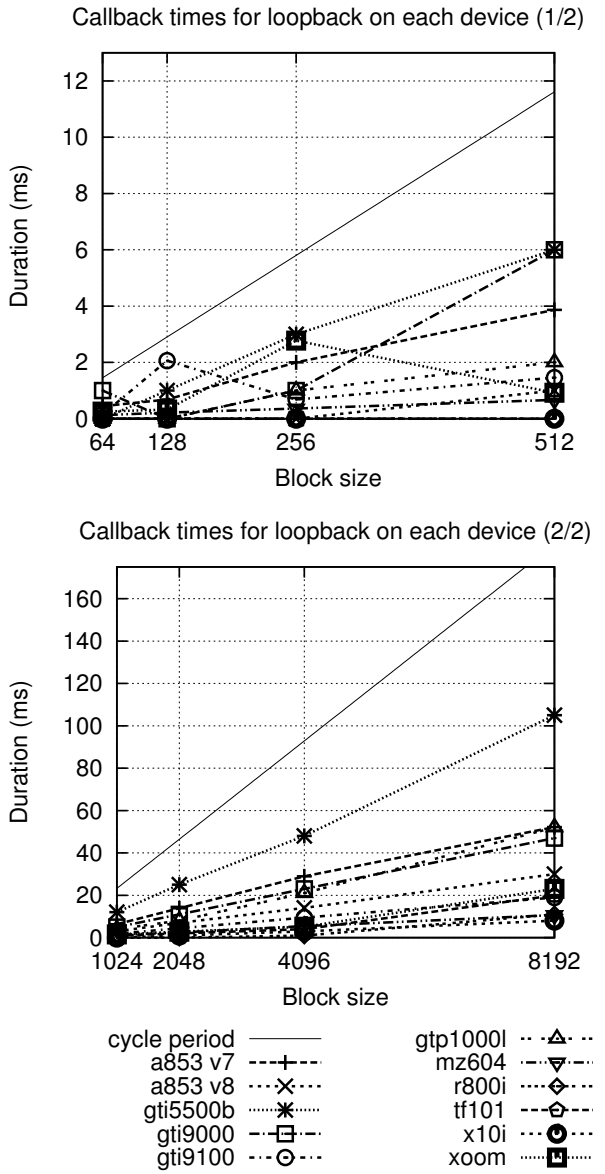

Figure 4. Times for the DSP callback with an empty loopback algorithm for each device.

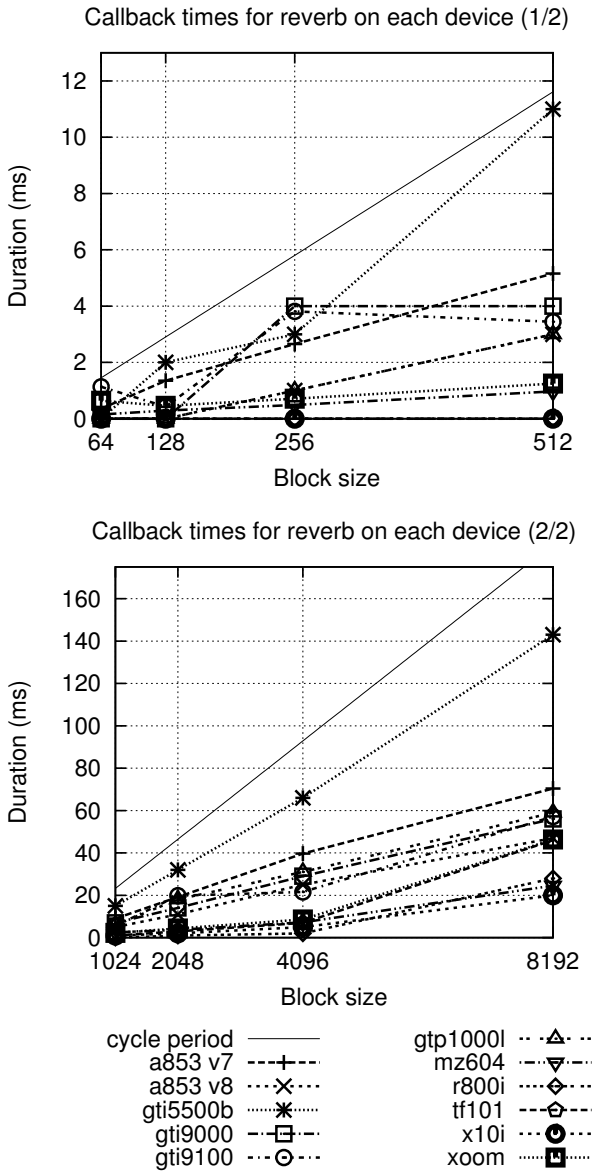

Figure 5. Times for the reverb algorithm (IIR filter with 2 coefficients) on each device.

significant difference of performance regarding the FFT algorithm. While for the device with API level 7 the FFT is prohibitive for blocks with 512 samples or more, for the device with API level 8 we were able to carry the FFT for blocks up to 8192 samples. To draw more general conclusions on API level differences we need more devices of the same model with different API levels.

3.2 Stress tests

The results of running the stress tests on different devices can be seen in Figure 7. Each line gives the maximum number of filter coefficients a device is able to process in real-time, as a function of block size. The nearly horizontal pattern is noticeable and might seem surprising at first glance, but it is actually expected, because of the nature of the stress tests. Even though large DSP blocks have more time available between callbacks, they also have more samples to be processed, so these two factors cancel each other out for any linear time DSP algorithm.

If we consider an ideal sample-wise stream processing DSP algorithm, we expect that the time available for computation between adjacent samples is simply the reciprocal of the sampling rate, and so the maximum number of operations should be $\mathcal{O}(1)$. The actual scenario is a lot more complicated due to the multitasking nature of the operating system, its priority rules for scheduling tasks, and also due to concurrent activities which are not controllable by a regular user. It is conceivable that the variations that appear on very small and very large blocks are due to priority issues: a process that uses less CPU during a single callback might get a relatively bigger share of CPU time than a process that eats up a lot of CPU time in a single callback.

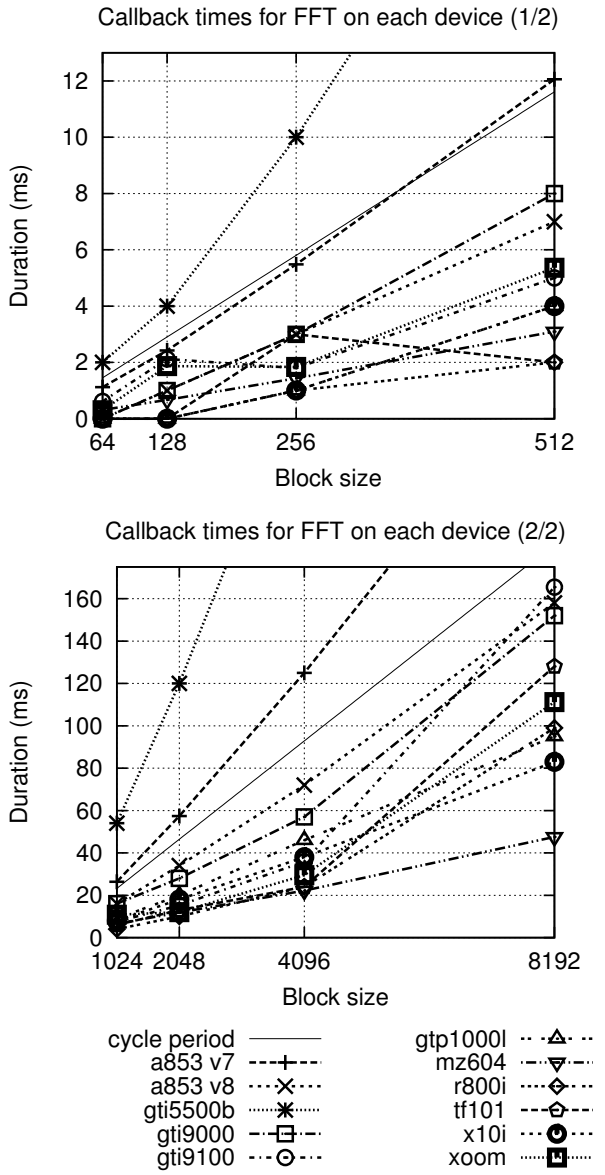

Figure 6. Times for the FFT algorithm on each device.

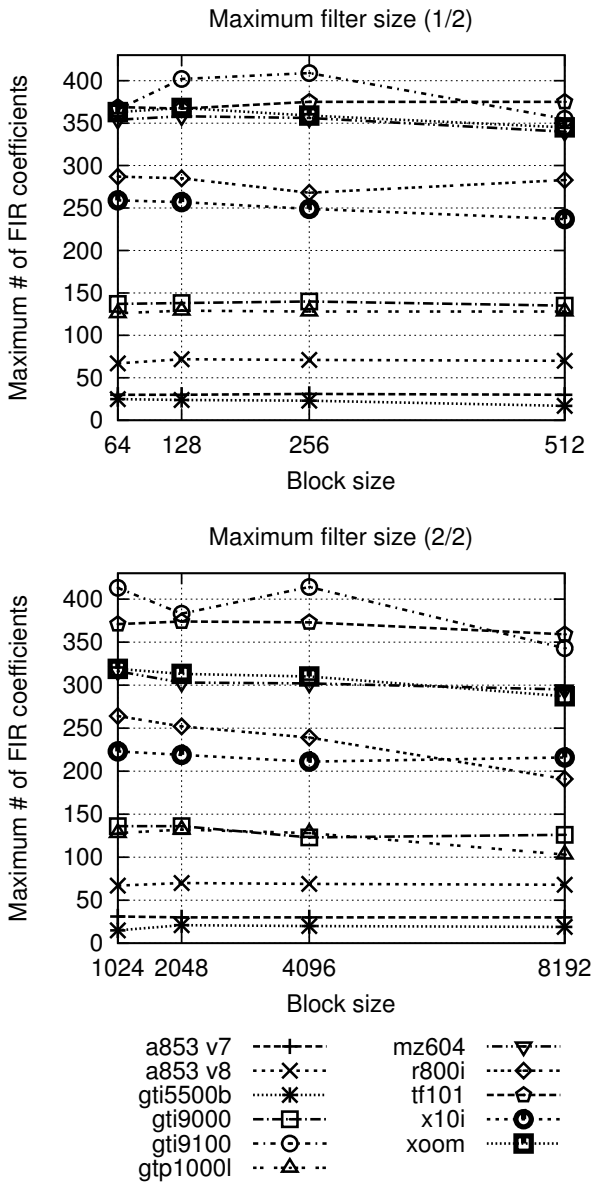

Figure 7. Result of maximum FIR filter size on each device.

If this is indeed the case, the larger K values for smaller blocks and the smaller K values for larger blocks would have been expected.

It is interesting to notice that the ranking of devices given by the stress tests is only partially preserved with respect to the ranking of callback times for the loopback algorithm (as seen in Figure 4). The exceptions are the tf101 and xoom device models, which are the only models that provide dual-core CPUs in our experiment. This might explain the different positions of these devices on the two tests. For single-core devices, the thread that runs the DSP callback has to share processor time with every other running task. For dual-core devices, the operating system scheduler can decide which task to run on each core, and the DSP thread might be allocated to the least occupied core, which increases the amount of computation allowed between callbacks.

4. DISCUSSION

We have devised a Java DSP model for Android devices and performance tests that can be carried on any device running Android API 7 or newer. With this application, we obtain time statistics about usual DSP computations and also stress devices to measure the feasibility of other DSP tasks under real-time constraints. All this provides us with upper limits for computations using our DSP model, but results could be optimized since our model runs with no special privileges on the system, no native (C/C++) code and very few code optimizations.

We could find few documentation on real-time DSP using the Android API on the application level, so we hope that this work can fill part of this gap by providing the source code (published under a free software license) and the model documentation (which is conveyed, in part, by

this text). Also, this work has been developed with the hope that it would be useful for computer music researchers and artists to obtain important information that can aid the choice of hardware to use on real-time applications and also the algorithmic decisions that have to be made during the real-time performance. By providing a systematic way to obtain performance measures related to usual DSP tasks and upper limits on FIR filters (which may be regarded as a fairly general DSP model as far as computational complexity is concerned), we hope to have achieved our goals.

4.1 Future work

Some ideas for future work are briefly discussed below:

- **libpd** [6] is a port of Pure Data's core engine which make possible to run Pd patches as C native code over Android's Dalvik Java VM. Comparisons of our Java DSP model with libpd's model can greatly improve the understanding of performance differences between Java and native C code.
- The Android API provides the `AudioEffect` class, which allows for native C/C++ code to be run over the system's streams of audio. A comparison of our model with native implementations using the Android's Native Development Kit (NDK) [7] could give us important insights about performance differences of different software stack levels running with different priorities.
- By allowing the DSP model to iterate through overlapped input signal blocks, we can estimate the maximum overlap factor for which it is feasible to process entire blocks with specific algorithms and then overlap-add them to generate the output signal [7].
- Other algorithms such as a complete FFT/IFFT procedure or a full Phase Vocoder implementation (with phase estimation and sinesum resynthesis) can be run to determine the feasibility of other kinds of basic DSP building blocks.

5. ACKNOWLEDGEMENTS

We would like to thank the members of the Computer Music Group [8] of the Computer Science Department of the Institute of Mathematics and Statistics of the University of São Paulo. This work has been supported by the funding agencies CAPES and FAPESP (grant 2008/08632-8).

6. REFERENCES

[1] K. Pathak, "Efficient audio processing in android 2.3," *Journal of Global Research in Computer Science*, vol. 2, no. 7, pp. 79–82, 2011.

[2] S. YI and V. LAZZARINI, "Csound for android," *To appear in: Linux Audio Conference 2012*.

[3] P. Brinkmann, *Making Musical Apps*. O'Reilly Media, 2012.

[4] C.-M. Lin, J.-H. Lin, C.-R. Dow, and C.-M. Wen, "Benchmark dalvik and native code for android system," in *Innovations in Bio-inspired Computing and Applications (IBICA), 2011 Second International Conference on*, dec. 2011, pp. 320 –323.

[5] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-time signal processing (2nd ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1999.

[6] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[7] T. G. Stockham, Jr., "High-speed convolution and correlation," in *Proceedings of the April 26-28, 1966, Spring joint computer conference*, ser. AFIPS '66 (Spring). New York, NY, USA: ACM, 1966, pp. 229–233. [Online]. Available: http://doi.acm.org/10.1145/1464182.1464209

[8] B. A. Shenoi, *Introduction to Digital Signal Processing and Filter Design*. John Wiley & Sons, 2006, vol. chipselect.

[6] http://puredata.info/downloads/libpd/

[7] https://developer.android.com/sdk/ndk/overview.html

[8] http://compmus.ime.usp.br/en/