On the performance of real-time DSP on Android devices

André J. Bianchi and Marcelo Queiroz

Institute of Mathematics and Statistics - Computer Science Department,
University of São Paulo

{ajb,mqz}@ime.usp.br

1. Measuring device's performance

Modern mobile phones are (time and space bounded) Universal Turing Machines. Because of their memory, CPU and battery limitations, it is interesting to study their **performance for real-time Digital Signal Processing (DSP)**. By collecting data about devices' performance, we may be able to provide feedback about resources consumption as basis for (user or automated) decision making related to the devices' use for artistic or research purposes.

Two strategies were devised to study Android-powered devices' performance:

- **Measurement of the time taken to perform common real-time DSP tasks** (such as input/output, FIR/IIR filtering and computing FFTs of varying sizes). This enables 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).
- **Stressing of the device** 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.

2. Benchmarking 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 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 $0 \leq i \leq 13$[1].

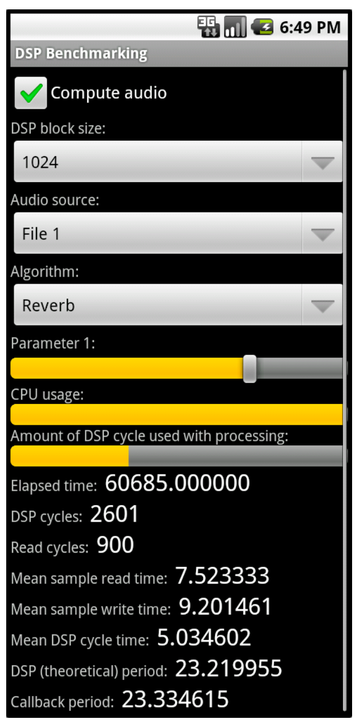


Figure: *The GUI controlling a live DSP process. The user is able to choose the DSP block size, the audio source (microphone or predefined WAV files) and the DSP algorithm that will be run. Also, slider widgets can provide explicit parameter input, while visual and numerical statistics give feedback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

With the results of sample read and write times, DSP algorithm execution times and DSP callback periods, it is possible to have a picture of each device's performance by comparing the time taken to perform these 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).

2.1 Algorithm benchmarking

One useful metric is the **DSP callback period**, which is the time required by a minimal set of DSP operations to perform any desired algorithm. That can be compared with the theoretical **DSP cycle period** to determine feasibility of that set of DSP operations.

To check if our DSP model (which is Java-based, includes timekeeping code, converts samples back and forth between PWM and floating point representation, relies on the API scheduling and operates in the same priority level as common applications) is indeed usable, we ran some simple DSP algorithms and took the **mean times** for a series of DSP callback periods for different block sizes.

The algorithms we implemented were:

- **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 $y(n) = -gx(n) + x(n-m) + gy(n-m)$ [4].
- **One-way FFT:** A Java implementation of the FFT (which takes $O(n \log n)$ steps, where n is the block size)[2].

2.2 Stress tests

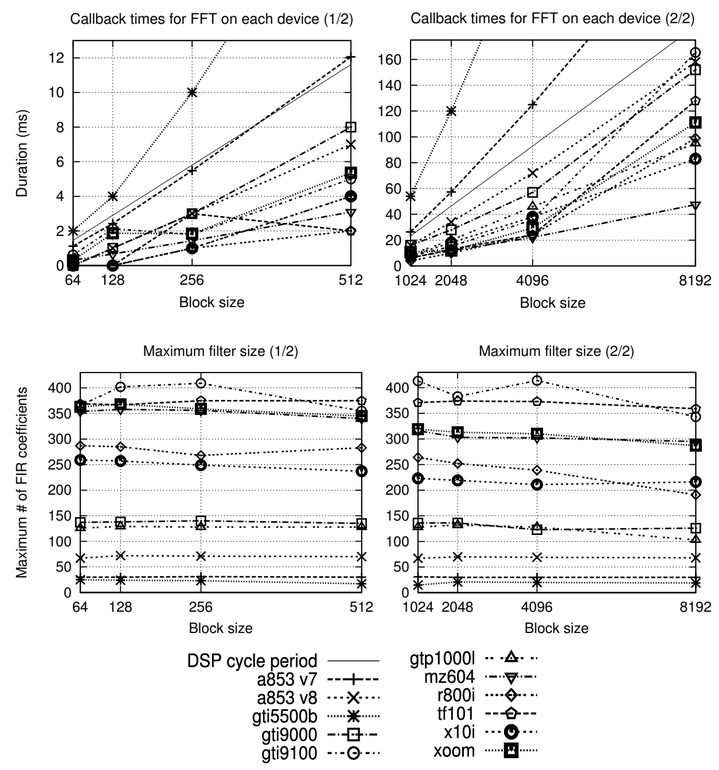
To determine how much (concurrent) DSP algorithmic computation can actually be performed inside the callback routine while maintaining real-time generation of output samples, we **stress-tested the devices using FIR filters** with increasing number of coefficients. By comparing the callback period for different sizes of filter with the theoretical DSP cycle period, we were 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.

Note that the general FIR 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.

3. What the results tell us

We ran our tests on a set of 11 different devices, and results were emailed automatically after the tests were finished.

The graphs obtained by running the algorithm tests are useful for **ranking devices with respect to computational power**, and for **giving a precise account of the time slack available for extra computation** on each algorithm. The results for the FFT algorithm can be seen below on the upper figures. As it takes $\mathcal{O}(n \log n)$ time with respect to input size, we can observe an upward tilt in on some of the devices. It should be expected that for larger block sizes real-time FFTs would become infeasible on every device.



On the lower figures above, we can see the results for the stress tests for each device. One example of an interesting result that we get by comparing the graphs for both strategies of evaluation, is that not always devices which perform filtering with more coefficients than others can also perform the FFT faster than others. Some examples of this are the models that provide dual-core CPUs (*tf101* and *xoom*).

4. Discussion

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 this work to be useful for computer music researchers and artists to obtain important information that can aid the choice of hardware and software to use on real-time applications and also the algorithmic.

References

[1] Peter Brinkmann. *Making Musical Apps*. O'Reilly Media, 2012.

[2] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[3] Cheng-Min Lin, Jyh-Horng Lin, Chyi-Ren Dow, and Chang-Ming Wen. Benchmark dalvik and native code for android system. In *Innovations in Bio-inspired Computing and Applications (IBICA), 2011 Second International Conference on*, pages 320 –323, dec. 2011.

[4] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time signal processing (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[5] Kailash Pathak. Efficient audio processing in android 2.3. *Journal of Global Research in Computer Science*, 2(7):79–82, 2011.

[6] Steven YI and Victor LAZZARINI. Csound for android. *To appear in: Linux Audio Conference 2012*.

[1] 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.