JET Audio Encoder Final Report

Authors: Taylor Autrey
Eric Addison
Josh Musick

Introduction

In a world of analog to digital conversion, there are many questions that can be asked regarding how to do it effectively and efficiently. This need for high quality audio compression took major root as a result of two things: first, an internet connection became common in many households, and second, the desire to transfer audio files through the internet grew rapidly. Initially, the raw audio signal that was captured in cd quality files was far too large to transmit in a reasonable amount of time. Therefore, the need to compress the audio files into more manageable file sizes became necessary.

When compressing data, there are two types of compression that can be performed, lossless and lossy. The names give obvious clues about them, but in general a lossless compression scheme allows the original file to be completely reconstructed from the compressed file, while lossy schemes allow for "good" reconstruction of the original file. The measure of how good a compression algorithm can be determined by a few factors, the compressed file size, the time it takes to make that compressed file, and the quality of the resulting file. In this project we embark to identify what specific parts of a compression scheme give those three qualities.

Compression of normal data files dictates that any compression scheme should be lossless, but audio files do not have that stringent of a requirement. Therefore, some liberty can be taken in how that process happens. Because audio signals for common audio files are actually the overlapping of many frequency bands at a given time, there became some obvious gains in simplifying the audio signal that is eventually saved. The first gain is the removal of frequencies that are actually overpowered by stronger frequencies at a given point in time. There are also some frequency bands which are outside the spectrum of human hearing. By exploiting these two areas, many possible techniques were developed. The original intent of this project was to analyze multiple compression schemes, but due to very specific encoding standards, the specific algorithms involved in the compression of MP3s are what we focused on. There turned out to be many different algorithms involved with interesting properties surfacing with specific input types.

In this project, we have implemented a pseudo-mp3 encoding program. Instead of focusing on a single algorithm, this project allows us to explore the implementation of a system of algorithmic components. The only part our program does not do is the formatting of an actual MP3 file, due to the highly detailed format of an MP3 file, which is outside the scope of this project. We took the major pieces of the encoding pipeline and implemented them with some variable controls to analyze the performance of the algorithms with a variety of inputs. The major pieces of the pipeline include: File I/O, a filterbank, byte bufferizers, Huffman encoding, and serializing out the compressed file. The program is able to take a wav file, compress it, and then decompress the compressed file. This allows us to evaluate the "quality" of the compression, or lack thereof.

Filter Bank

The first stage in the compression pipeline is a poly-phase filter bank, which is typical of modern audio compression schemes. The main objective of the filter bank is to enable multi-rate processing; that is, to process separate frequency bands of a signal differently.

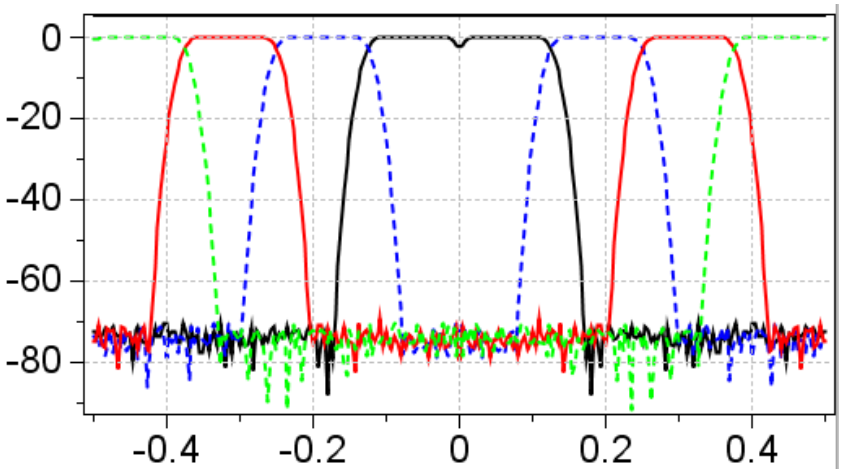

Figure 1: The frequency response of a 4-band filter bank, which will split an incoming signal into four separate subbands.

The filter bank splits the incoming signal into a specified number of frequency bands by applying filters of constant bandwidth and increasing central frequency. These filters are all derived from a single prototype filter by modulating (i.e. shifting in frequency) them to the appropriate central frequency. Time domain filtering is performed in order to avoid the subtleties involved with frequency domain filtering: time-domain signal wrap-around for example.

After the filtering operation has been performed, the resulting subband signals can be modulated down to zero Hertz, then decimated by a factor of N_{bands}. This results in no additional samples being created by the filter bank. Ideally, the decimation will result in no loss of signal fidelity since the filtering has left each subband with a limited bandwidth. In practice, however, this depends on the quality of the prototype filter. Use of the Noble Identities allows the order of decimation and filtering to be reversed, resulting in significant computational savings: since time domain filtering is an $O(N^2)$ operation, it is more efficient to filter a set of decimated signals than it is to filter one long signal.

In the reverse direction, essentially the same thing happens. Each filtered, modulated, and decimated subband signal is first remodulated to its original central frequency, then filtered to prevent aliasing, and finally upsampled to restore the original sample rate. The restored subband signals are summed to produce a reconstructed signal. The reverse operation can be very nearly lossless for a special class of filters known as Pseudo Quadrature Mirror Filters (PQMF). Construction of these filters rely on optimization techniques, however, and are beyond the scope of this project. For our rudimentary compression scheme, we accept some amount of signal distortion due to a low quality prototype filter.

Multiple parameters have associated tradeoffs in the filter bank. The length of the incoming signal windows affects the run time, as longer windows increase the processing time over multiple short windows. However, longer window times allow for effective use of longer filters, which improve signal fidelity.

The number of subbands also represents a tradeoff: more subbands results in finer frequency division and allows more sophisticated multi-rate processing to be applied, however the effects of narrow-band filtering (e.g. aliasing and imperfect reconstruction) become more evident with a larger number of subbands.

Design of the prototype filter presents a tradeoff as well. Specialized filters can be designed that contain desirable properties for audio processing, but this requires careful design choices, possibly optimization techniques, audio domain knowledge, and experience. On the other hand, a generic lowpass filter can be generated quite quickly, which is sufficient for our needs and timeline. The run time of filter application does not depend on the complexity of the filter design, only on the final length of the filter, so the tradeoff is really between design effort and filter quality.

The complexity of filtering in the time domain is $O(N^2)$. In our application, for each incoming signal window of length N_w, we split the window into N_{bands} subband signals of length N_w/N_{bands} each. The number of operations for an incoming signal window is then

$$N_{bands} O\left(\left(\frac{N_w}{N_{bands}}\right)^2\right) = O\left(\frac{N_w^2}{N_{bands}}\right).$$

This shows that increasing the number of subbands should lower the run-time of the filtering step. We can go one step further to examine the effect of the signal window size by considering the total number of samples in the audio file. Let N_s be the total number of samples, N_w be the number of samples in a window, and n_w be the number of windows such that $N_s = n_w N_w$. Then the computational cost of the filterbank for the entire audio file is

$$n_w O\left(\frac{N_w^2}{N_{bands}}\right) = n_w O\left(\frac{N_s^2}{n_w^2 N_{bands}}\right) = O\left(\frac{N_s^2}{n_w N_{bands}}\right)$$

This illustrates a previous comment quantitatively: it is more efficient to filter a larger number of smaller windows than vice versa.

Modified Discrete Cosine Transform

The Modified Discrete Cosine Transform (MDCT) is a Fourier related transform that takes the subbands produced by the filter bank and splits them into even finer frequency divisions. The complexity of a naive MDCT implementation is $O(N^2)$, though some pre- and post-processing of an FFT can be applied to achieve $O(NlgN)$ performance. The purpose of the MDCT is to apply sophisticated psycho-acoustic modelling, for example attenuating energy in neighboring frequencies that are too close for human ears to identify.

Our very simplistic psycho-acoustic model does not incorporate such advanced techniques. As such, we have included a naive implementation of the MDCT in this project for completeness, but do not expect any increase in performance, quality, or compression. We do not include the MDCT as part of our final processing pipeline.

Byte Bufferizers

You can coarsely categorize the stages of our pipeline as either signal processing steps (filtering, functional transforms) or byte related steps (Huffman encoding, serialization). In order to move from one domain to the other, explicit steps were created to take the float based audio data and convert it to a collection of bytes. We refer to these steps as ByteBufferizers.

The ByteBufferizer steps represent the transition from meaningful audio signals into generic bytes. Accordingly, this step must carefully pack all of the information required for audio reconstruction, to be recovered during the reverse processing step. This step essentially takes the audio data stream, along with a small collection of metadata (sample rate, number of channels, window size, etc), and stuffs it all into a byte buffer. The resulting byte buffer can then be passed along to the byte related processing steps, which need know nothing about the audio/signal side of the pipeline.

The audio samples we work with are constrained to lie within the range from [-1.0, 1.0]. Our initial approach to byte-ifying our samples was to transform each floating point value into an n_{byte} integer as follows:

$$v_{int} = \left\lfloor M\left(\frac{v_{float} + 1}{2}\right)\right\rfloor$$

Where v_{float} is the floating point sample value, v_{int} is the resulting integer value, and M is the maximum value that can be represented by an n_{byte} integer.

We can gain additional accuracy in our byte-ification if, instead of keeping fixed minimum and maximum values, we scan each window of data, identify the local min/max, store the new local min/max values in the byte buffer, and use them to byte-ify each sample within the window. In this case, float values are converted to integer values by:

$$v_{int} = \left\lfloor M\left(\frac{v_{float} - v_{min}}{v_{max}}\right)\right\rfloor$$

Where v_{min} and v_{max} are the local min and max values within a window. Note that for the case of fixed $v_{min} = -1$ and $v_{max} = 1$, this reduces to the above equation. For sections of the audio stream that do not make use of the full dynamic range, this can result in increased accuracy. The tradeoff is that we must store two additional values per window for use during the reverse processing step.

In either case, the byte bufferizer needs to act on each sample a fixed number of times, giving this step a complexity of $O(N)$.

Psycho-Acoustic Model

The Psycho-Acoustic Model is where most modern audio compressors (notably MP3) realize the biggest compression gains. Advanced models exploit various aspects of human physiology and aural perception to determine how much of a signal can be safely removed or stored with a reduced number of bits without adversely affecting the reconstructed audio. Some of these perception aspects include frequency masking, where a strong signal at one frequency can render a weaker signal at a nearby frequency inaudible, or similarly, temporal masking, where a loud sound at one instant in time can mask over a subsequent quiet sound.

Our simple model performs one task: given lower and upper frequencies to define an effective "passband", reduce the sample precision (byte depth) of the samples from the subbands with frequencies that lie outside of passband range.

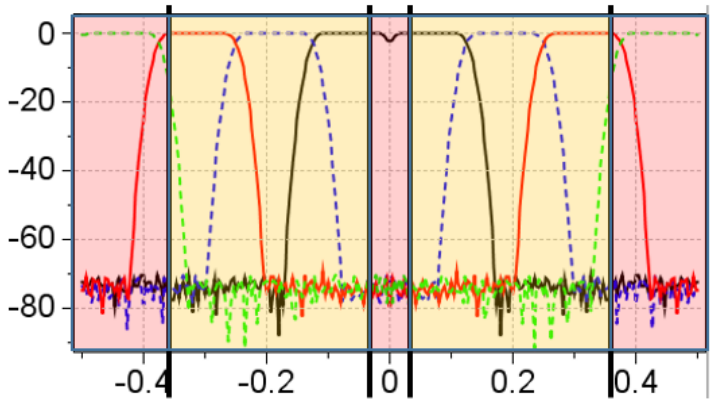

Figure 2: Illustration of the simple psycho-acoustic model. Subbands that lie within the passband defined by an upper and lower frequency (yellow region) are encoded with full byte-depth, while subbands outside of the passband (red region) are encoded at half of the full byte-depth.

For example, if we define a passband from 200Hz-10000Hz, then our psycho-acoustic model will instruct the appropriate byte bufferizer to store samples from the subbands below 200Hz and above 10000Hz with reduced precision. The reduced-precision samples can still represent the full dynamic range of the input, but in a less granular fashion. The motivation behind this simple model is that human hearing is most sensitive to frequencies in a certain range, so for signal from frequencies outside of that sensitive range, our perception of the reconstructed sound may not suffer noticeably from the lack of precision.

Huffman Encoder

The final stage of actual work in the pipeline is the Huffman Encoder. This stage actually performs lossless compression. The general strategy for Huffman encoding is to build a frequency table of characters in a file and then use a variable-length code to represent the characters, with the more frequent characters having shorter codes. In this implementation, the characters are bytes, represented by the integer values 0 to 255.

Building the frequency table is the first step. This requires an initial pass through the entire input which keeps track of the number of occurrences of each byte in an integer array. The complexity of this pass is $O(N)$. Next, the code tree and a canonical Huffman code are built from the frequency table. In this step, the algorithm is making a fixed number of passes over fixed length arrays and queues which will be constant time, so the complexity of this part is $O(1)$. Finally, when writing the output, the code table must first be written, this again is constant time. Then, a final pass through the input to do the actual encoding, which is complexity $O(N)$. Thus the overall complexity of the Huffman encoding implementation is $O(N)$.

One way in which the algorithm can be changed is by eliminating the initial pass through the input to build the frequency table. This also eliminates the need to build a canonical code tree that is included with the output. The strategy for this implementation is to start with a blank frequency table

and increment the counts for each byte as the input is being encoded. At first we have no data on frequency so the encoding is suboptimal. But as we progress through the input and update the frequency table, we periodically rebuild the code tree and use the updated Huffman codes to encode the next series of bytes. While this means that a particular byte could have a different code at different points in the output, the rebuilding of the code tree is done at the same points in both the encoding and decoding processes so that the output of the encoding is decoded correctly. The complexity of this version of the encoding algorithm is also $O(N)$, although in practice, since it's only making a single pass through the input it will take slightly more than half the time of the original implementation.

Serialization

As mentioned before, the writing of the compressed file was not actually done to the MP3 standard, but was instead a simplified writing of bytes to a raw data format (.jet). This step bypassed the complexity of encoding to a specific standard, yet enabled the program to write the data to a binary file for size comparison. This stage is making a single pass through the data stream as it writes them to file. The complexity of this stage of the pipeline is therefore $O(N)$.

Data Analysis

To explore the various performance attributes of our compressor, we designed a test bench to iterate though many parameter combinations and collect statistics. The adjustable parameters in our pipeline included:

- Signal window length
- Number of subbands in the filterbank
- Filterbank filter length
- Adaptive or standard Huffman encoding
- Adaptive or standard byte bufferizer

The metrics we collected included:

- Compression ratio
- Absolute RMS error
- Total compression time
- Total decompression time

The results provided us mainly with verifications of our expectations. For example, a longer filter length resulted in decreased RMS error (Figure 3) and a small upper frequency for the psycho-acoustic model resulted in an improved compression ratio (Figure 4).

There were some surprises, however. For instance, neither the adaptive byte bufferizer nor the adaptive Huffman encoding improved the measured RMS error, on average. In fact, using the adaptive byte bufferizer caused the average compression ratio to rise significantly (see Figure 5 and Figure 6)!

Improvements

We were able to achieve fast compression time and a reasonable compression ratio. What suffered most was the audio quality of the decompressed wav files. We could improve the performance of our compressor in several ways. Using a carefully designed, purpose built prototype filter would be a good starting place. Using overlapping signal windows could help as well, as this could potentially reduce some of the audible noise that may be due to the hard-cut window boundaries. Compression could be further improved with a more sophisticated psycho-acoustic model, which makes use of the MDCT frequency divisions.

Conclusion

The process of compressing audio signals is challenging to do well. From a purely performance or data compression perspective, there are many ways to optimize, but those routes typically cost the end product in audio quality. Therefore, finding a good mixure of the three takes very sophisticated models, and careful adjustments regarding where to accept data loss and where not to. Additionally, a wide variety of inputs can all be affected very differently by specific acoustic models. This can be an advantage, if the ideal audio input is given, or it can be a major problem if the wrong model is applied to a given input. This problem is common however, to many computer algorithms. All possible inputs must be considered, to produce a well-rounded solution. Many of the tunable parameters in this project highlighted this very fact. Many of the adjustable parameters, which were expected to improve performance in some way, actually hurt the performance, such as the adaptive byte bufferizer. Yet some parameters, behaved as we expected, based on simple analysis of the algorithms. This was the case with the improved performance when using more filter-bank sub-bands.

In the end, this provided a valuable experience using multiple algorithms with adjustable parameters, to try accomplishing a task which has multiple competing outcomes. An issue that is common in most all industry applications. If the problem were to simply make the file smaller, it would be easy, but to try making it smaller, quickly, yet retain good sound quality, requires thought and careful application of theory.

To access the source code and run the compression pipeline yourself, you can clone the project from GitHub at: https://github.com/ericaddison/Algs_Term_Project.git

Appendix A: Result Figures

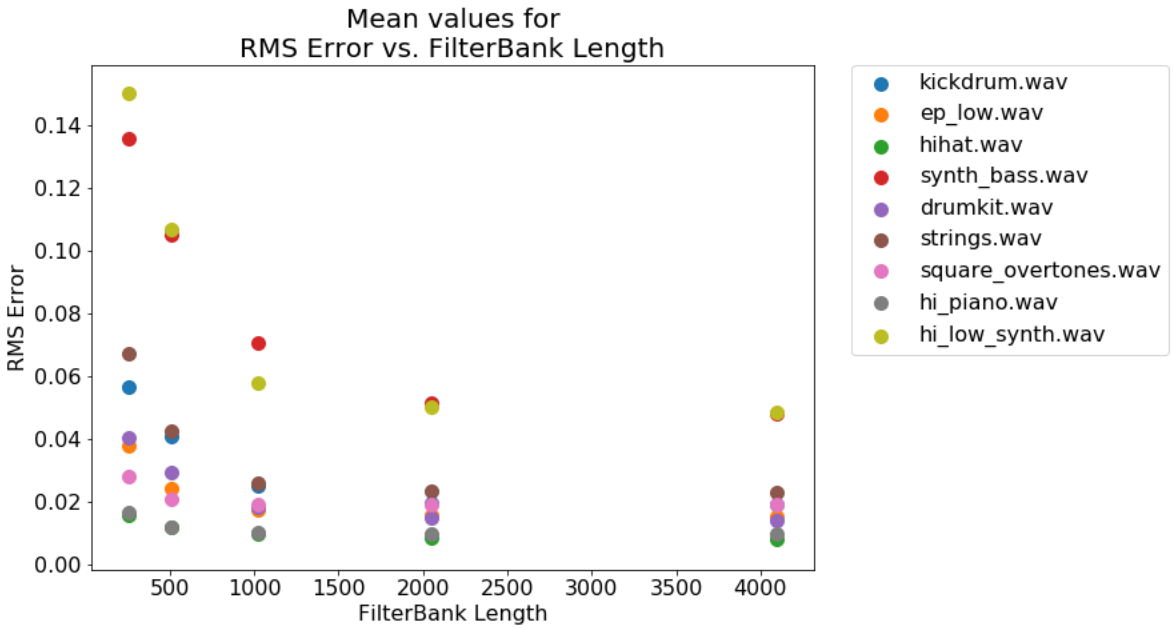

Figure 3: Mean value of RMS error vs. filter length for all input files.

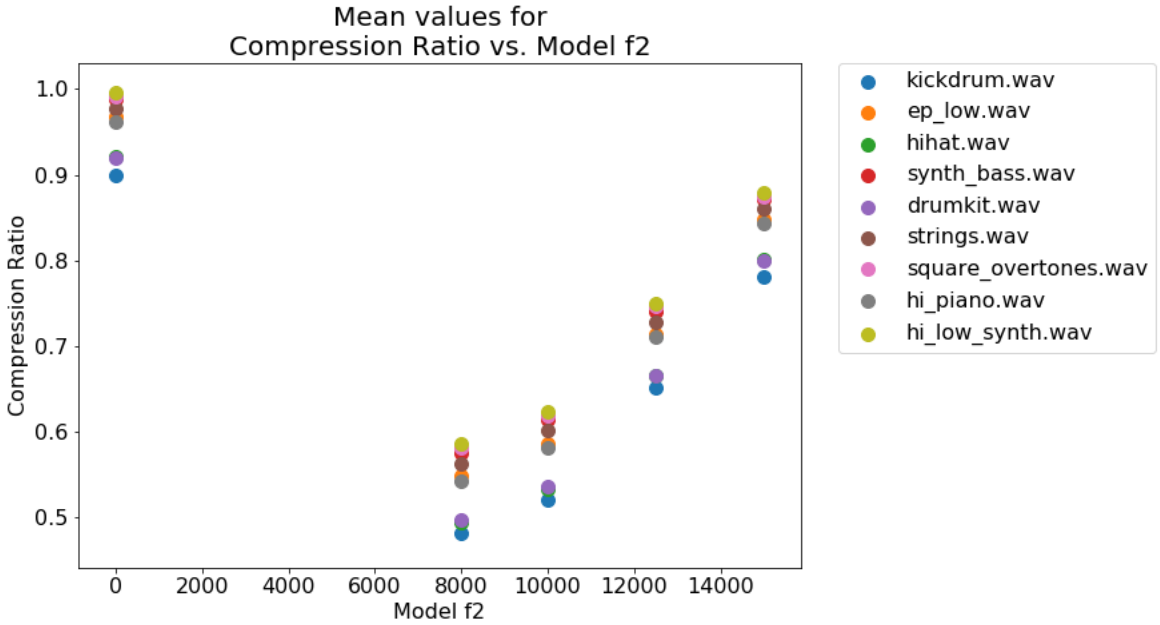

Figure 4: Mean value of compression ratio vs. model upper frequency (0 => no model) for all input files.

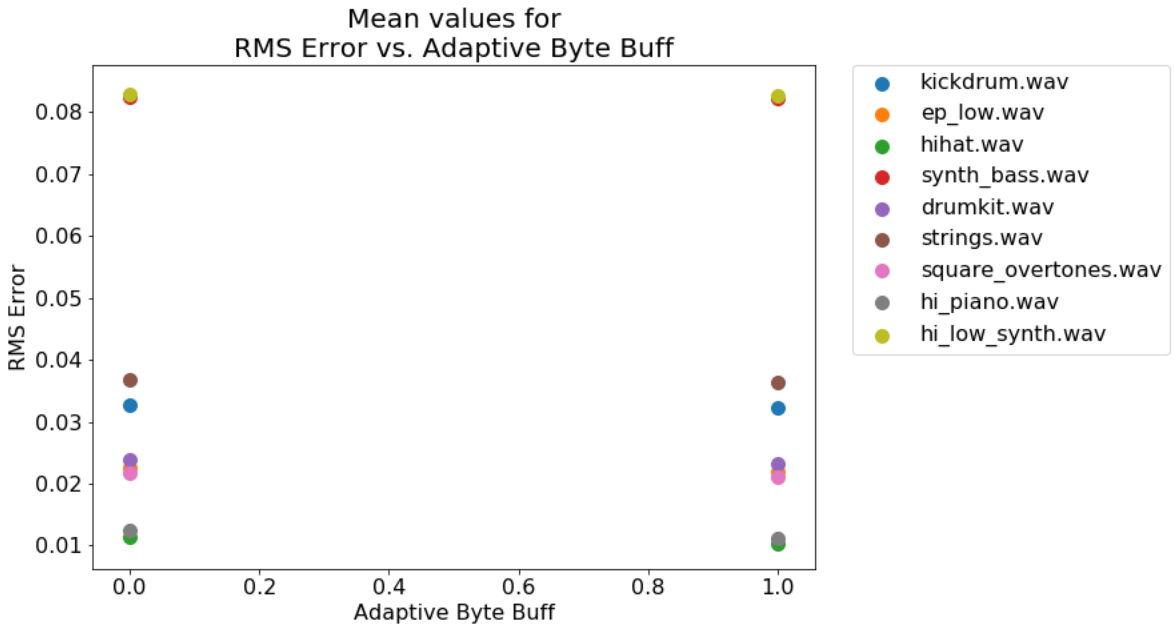

Figure 5: Mean value of RMS error vs. Adaptive Byte Buffer (0=>not enabled, 1=>enabled) for all input files.

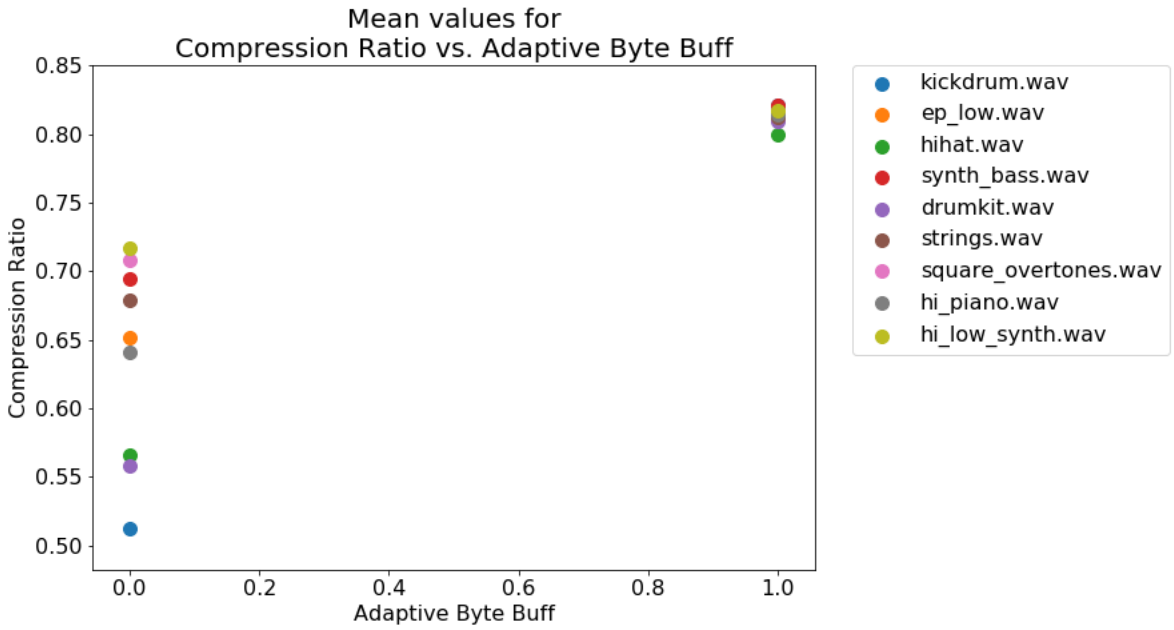

Figure 6: Mean value of compression ratio vs. Adaptive Byte Buffer (0=>not enabled, 1=>enabled) for all input files.