Perceptual Audio Coder with Transient Detection, Block Switching, and Entropy Compression Algorithms

Pranavi Boyalakuntla, Audrey Lee, Lycia Tran
Electrical Engineering, Stanford University
Email: pranavib@stanford.edu, alee22@stanford.edu, lyciat@stanford.edu

***Abstract*—In this project, we created an audio coder to improve the audio quality for low bit rates (96 kbps and 128 kbps) by implementing transient detection and block switching. Transient detection and block switching should help us remove the presence of pre-echo in the transient blocks which was the most noticeable artifact present in our original coder. Since block switching and transient detection can be quite bit demanding, we attempted to implement different entropy coders (Huffman coding and Arithmetic coding) to reduce the number of bits needed to encode our audio files.**

I. Motivation and Background

Perceptual audio coders enable compression of signals while maintaining the important components of the sound such that the compression is minimally audible. Through the Music 422 class, we worked to implement various portions of a perceptual audio coder at a variety of data rates and coding schemes including 128 Kbps and 192 Kbps, as well as time-domain and frequency-domain compression schemes for scale and mantissa bits.

Perceptual audio coders suffer from artifacts often due to the loss of signal information. One common artifact that is found is pre-echo which, as per the name, is a faint echo of the sound directly before the sound itself [1]. This pre-echo artifact occurs because the quantization noise from the transient is ”spread out” across a longer block size. Therefore, we can mitigate this artifact by introducing shorter block sizes when a transient is detected. Once a transient is detected, we can utilize block switching to change the block size for the entire block such that the transient is only ”spread out” over a short block size, thereby reducing the effects of the pre-echo. For harmonics and steady-state signals, we can still employ longer block sizes for the improved benefit of greater frequency resolution.

Since block switching is very bit intensive and it is good practice to try and reduce the number of bits needed to encode data, we wanted to implement a method to save bits. Entropy coding would allow us to reduce the number of bits needed to encode and decode the mantissa values, which should help us improve our audio quality at lower data rates.

Entropy coding is a type of lossless data compression method that must have an expected code length greater than or equal to the entropy of the source. The source coding theorem states that for any source distribution, the expected code length satisfies

$$E_{x\sim p}[k(c(x))] \geq E_{x\sim p}[-\log_b(P(x))] \qquad (1)$$

where k is the number of symbols in a code word, c is the coding function of the lossless encoder of choice, b is the number of symbols used to make the output codes based on the lossless encoder, and P is the probability of the source symbol. With entropy coding, we expect that the average number of bits needed to encode the data sample should not be greater than the calculated entropy given the probabilities. [2]

II. Project Overview

In this project, we aimed to improve the perceptual audio coder implemented earlier in the Music 422 class at lower data rates such as 96 Kbps and 128 Kbps. We also wanted to address the pre-echo

artifact and aimed to utilize compression algorithms to reduce the storage size of the PAC files for encoding and decoding.

III. Baseline Codec

Our original baseline coder is an MDCT based coder that windows the original input data using a Sine window. It also uses the FFT to determine the peaks of the masking curve and calculates the Signal to Mask Ratio (SMR) to determine how many blocks should be allocated to each MDCT line and band [3]. These original baseline coder features can be seen in blue in Figure 1. While our baseline coder does perform decently well, we aim to improve its performance by implementing transient detection and block switching. The most noticeable artifacts we had using our baseline coder was pre-echo. This is caused because our block lengths are too long, so we hear an echo of a transient before it officially starts. Transient detection and block switching would allow us to detect when a transient is about to occur and switch to a shorter block length to reduce the pre-echo artifacts [3]. Since block switching is often a very bit demanding process, we chose to implement Huffman coding (and attempted to implement Arithmetic coding as well). Huffman coding is an entropy coder that allows us to encode our data with fewer bits by encoding more commonly occurring symbols as shorter bit strings [4]. We also attempted to implement Arithmetic coding which is another entropy coder that encodes an entire block of data into a single bit string [5]. These additional features can be seen in the gray blocks in Figure 1.

IV. Block Switching and Transient Detection

Block switching helps remove the pre-echo artifact due to longer block sizes. This pre-echo is more audible in transients; meaning that if we can detect which blocks contain transients, we can then use a shorter block length. The transient detection and block switching algorithm is listed below:

A. Transient Detection Algorithm

To accurately detect transients, we used two separate methods.

1) High Frequency Energy

2) Time Frequency Spectral Flatness Measure

High Frequency Energy

High Frequency Energy (HFE) can be described in Equation 2 where N is the block size and K is the high frequency cutoff index. The units of HFE are the high frequency Sound Pressure Level (SPL). This implementation uses the Kaiser-Bessel Derived (KBD) window, where $\frac{8}{N^2}$ is the associated normalization factor [6].

$$\begin{aligned} \text{HFE} &= \text{SPL}\Big(\sum_{K}^{N/2-1} \frac{8}{N^2}|block_{FFT}|^2\Big) \\ K &= N \times \frac{\text{cutoff frequency}}{f_s} \\ \text{block}_{\text{FFT}} &= \text{FFT}(\text{KBD}(\text{block})) \end{aligned} \tag{2}$$

Rapid changes in the HFE metric across two blocks (the current block and the previous block) can be used to identify whether or not the current block contains a transient.

For the HFE metric, we can use a calibrated threshold to determine whether or not a block contains a transient.

Time Frequency Spectral Flatness Measure (TFSFM)

The Spectral Flatness Measure (SFM) measures the whiteness of a signal and is defined as the ratio between the geometric mean and mean of the same signal in the frequency domain [7]. Equation 3 defines SFM where $K = N/2$ [6]. This means that as the signal becomes more flat, the SFM metric approaches 1. As proposed by Fan et al., the SFM for random signal is similar to a signal with a transient. Therefore, the SFM needs to be compared to the Temporal Flatness Measure (TFM) [6].

$$\text{SFM} = \frac{(\Pi_{k=0}^{K-1}|X(k)|^2)^{1/K}}{\frac{1}{K}\sum_{k=0}^{K-1}|X(k)|^2} \tag{3}$$

The TFM describes the fluctuation of a signal in the time domain while the SFM describes a signal's fluctuation in the frequency domain. Similarly, the TFM is defined as the ratio between the geometric mean and mean of the same signal in the time domain. Equation 4 describes how the TFM can be calculated [6].

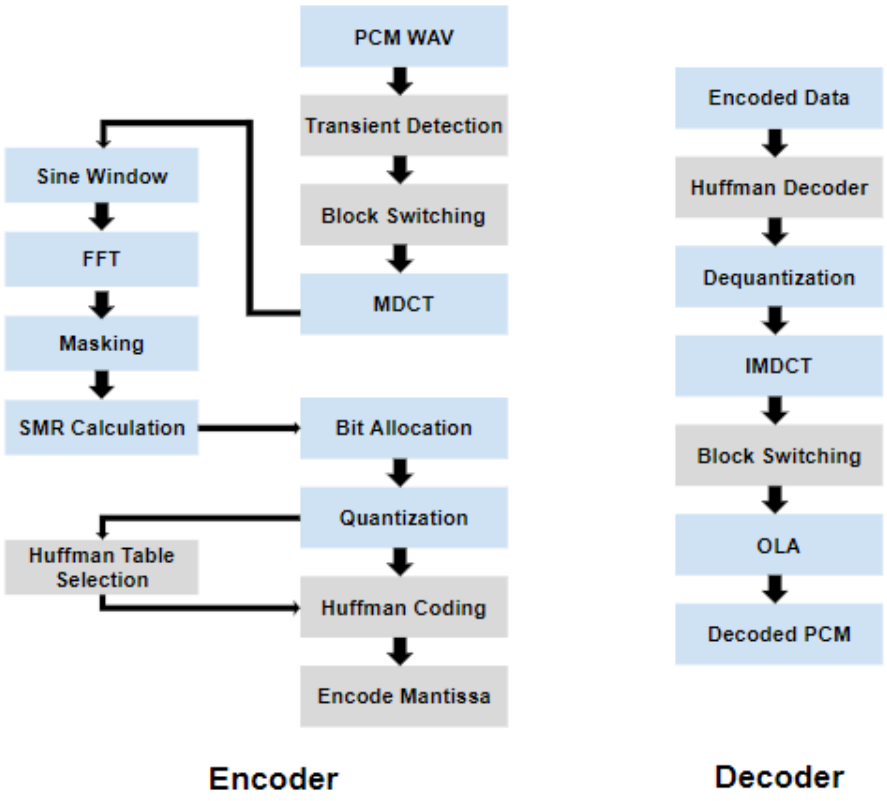

Fig. 1. [left] Block Diagram of our improved audio encoder, [right] Block diagram of our improved audio decoder

$$\text{TFM} = \frac{(\Pi_{n=0}^{N-1}|x(n)|^2)^{1/N}}{\frac{1}{K}\sum_{n=0}^{N-1}|x(n)|^2} \tag{4}$$

Since transients would have a higher SFM and a lower TFM, the ratio of the SFM and TFM can be calculated to generate the Time Frequency Spectral Flatness Measure, or TFSFM as shown in Equation 5.

$$\text{TFSFM} = \frac{\text{SFM}}{\text{TFM}} \tag{5}$$

For the TFSFM metric, we can use a calibrated threshold to determine whether or not a block contains a transient.

B. Block Switching

Block switching enables perceptual audio coders to have adaptive filterbank resolution to handle encoding transients with less artifacts such as pre-echo where the quantization noise gets spread. One of the first transition block solutions was proposed by Edler where the long block was turned into a transition window (long block to short block) by setting a portion of the right half of the transition window to zeros [8]. Another solution to the block switching problem was proposed by the Dolby AC-2A team [9]. In this method, the long-to-short transition window, is the left half of the long block, and the right half of the short block. The short-to-long transition window is the time reversal of the long-to-short transition window. In both Edler block switching and AC-2A block switching the data rate will increase when going from long to short blocks without a data rate increase when returning back to long blocks [3].

V. Entropy Coding

Since block switching can be a very bit intensive method, we tried implementing two different entropy coders (Arithmetic coding and Huffman coding) to reduce the number of bits needed to encode our mantissa values.

A. Huffman Coding

Huffman coding is an efficient and lossless way to compress data. Huffman's methodology uses the concept of symbol counting where unlike many other compression algorithms (such as Arithmetic coding), Huffman coders scan the file and generate a frequency table and tree before beginning the true compression process. [4] Huffman coding can be broken down into three components:

- Frequency Counting
- Tree Building
- Character Encoding

Frequency Counting

The Frequency Counting step calculates how often a certain symbol will show up in our encoded data stream. For example, if the coder is trying to encode a string that contains "Hello".

Table I shows our frequency counting table

TABLE I
FREQUENCY COUNT

Letter	Counts
h	1
e	1
l	2
o	1

After the frequency counting table has been constructed, the Huffman coder transforms these counts into probabilities of how often the symbols will show up as shown in Table II where the probabilities are in decimal format and sum to one.

TABLE II
SYMBOL PROBABILITY

Letter	Probability
h	0.2
e	0.2
l	0.4
o	0.2

From these probabilities, a tree structure can be built to encode these symbols.

Tree Building

Once the frequency table is created, the Huffman coder builds a tree. This tree follows the same structure as a normal binary tree where it contains nodes and leaves, and where each leaf contains the code corresponding to a symbol from the frequency table. Each leaf contains two values, the symbol and its corresponding probability. [4]

To build the tree, the Huffman coder will traverse the probability table, and push the symbols with the highest frequencies to the top of tree. It will continue this traversal until all values from the table are encoded on a leaf within the tree. [4]

A drawing representation of the tree for the word "Hello" can be seen in Figure 2. In this diagram, the word "Hello" can be broken down by the symbols and their probabilities. Symbols with higher probabilities are higher up in the tree whereas symbols with lower probability of showing up will be on a lower leaf. So, the symbols with higher probabilities will have a shorter bit string. This method prevents symbols from overlapping in terms of bit string representation as each symbol is encoded to be leaf of the tree and never a node.

This tree can then be used for the actual encoding and decoding of the data of interest.

Character Encoding

Character encoding is the final step. From the Huffman Tree, the Huffman coder can encode the symbols from the file of interest and write the encoded bytes to a new (compressed) file. Since each bit string within the compressed file corresponds to a known symbol, the file is able to be decompressed without losing any information by just traversing the known, pre-calculated, Huffman Tree. [4]

B. Arithmetic Coding

Encoding

Arithmetic coding works by first initializing a range $[L, H)$ (often $L = 0$ and $H = 1$) that corresponds to the entire sequence of symbols in the block. We then divide the interval based on the probability we see each symbol. So if we had symbols A, B, and C with the probabilities 0.3, 0.5, and 0.2 respectively, our starting interval would be A = $[0, 0.3)$, B = $[0.3, 0.8)$, and C = $[0.8, 1)$. From this, our next interval range will be the interval of our first symbol. This means if the first symbol in our block is B, the new interval is $[0.3, 0.8)$. Now we can resplit this new interval again with our symbol probabilities to get BA = $[0.3, 0.45)$, BB = $[0.45, 0.7)$, BC = $[0.7, 0.8)$. Based on the second symbol of our block, we would choose the corresponding interval to be our new interval. We

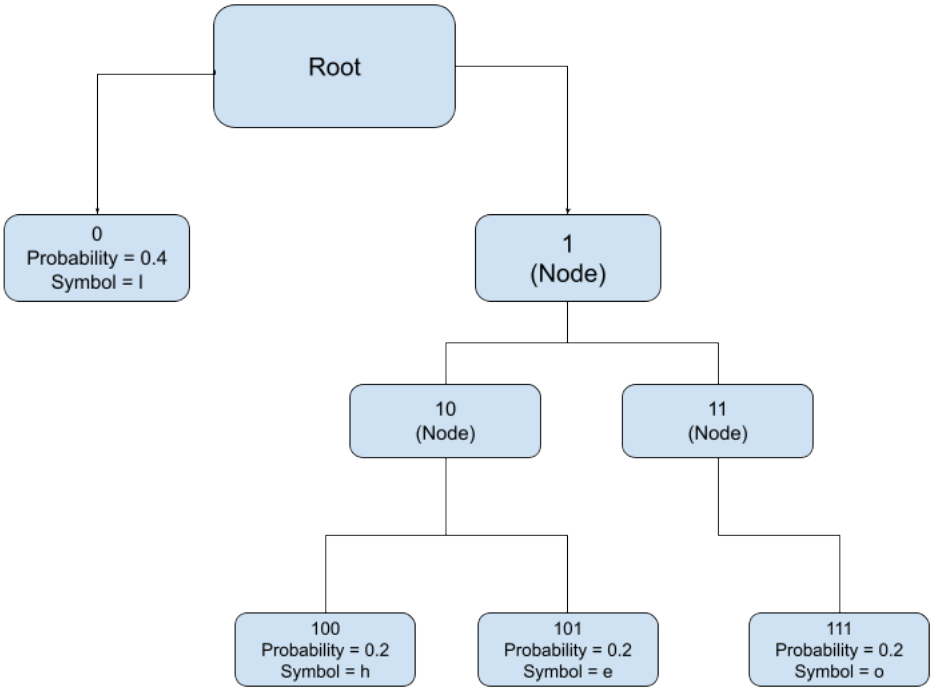

Fig. 2. A Huffman Tree Representation of "Hello"

will keep repeating these steps of redividing our interval based on symbol probabilities and choosing the corresponding interval until the entire block has been encoded. This will result in a final interval that we can recover our symbol values and their order from.

We now have to communicate this final interval through the bit stream. To do this, we first pick Z to be the midpoint of our final interval, $[L, H)$

$$Z = \frac{L + H}{2}$$

From this, we truncate Z into Z_{hat} such that Z_{hat} is k bits and Z_{hat} and any extension of Z_{hat} in binary are still in the interval $[L, H)$. We want to truncate Z_{hat} to be as few bits as possible while still meeting these two requirements. Our final encoded bit array will include our Z_{hat} bit array and n, the number of symbols encoded in our bit array [5].

Decoding

We start with our Z_{hat} value and the probabilities of our symbols as well as our initial interval. We start the same way as we do in encoding by dividing the interval based on the probability we see each symbol. From this initial interval, we find in which symbol interval does Z_{hat} lie in. This will be the first symbol we recover. From that symbol interval, we divide it up again based on the probabilities of our symbols and find which symbol interval contains Z_{hat}. We will keep repeating these steps we have decoded all of our n symbols [5].

VI. Implementation

For our implementation, we implemented all of the new features – transient detection, block switching, and entropy coding – as shown in Figure 1. As a warning for running the code, this code will take hours to run if you want to encode and decode all of the different sound files available. This is because the implementation has to run the block switching and transient detection, and then for every small and large block, we have to run Huffman to get the number of bits needed to allocate for the PAC file and run Huffman again for the actual encoded bitstream.

A. Transient Detection

In order to detect transients reliably we used a combination of the TFSFM and HFE techniques to determine transients. As specified earlier, in order to use these two metrics to detect transients, you need

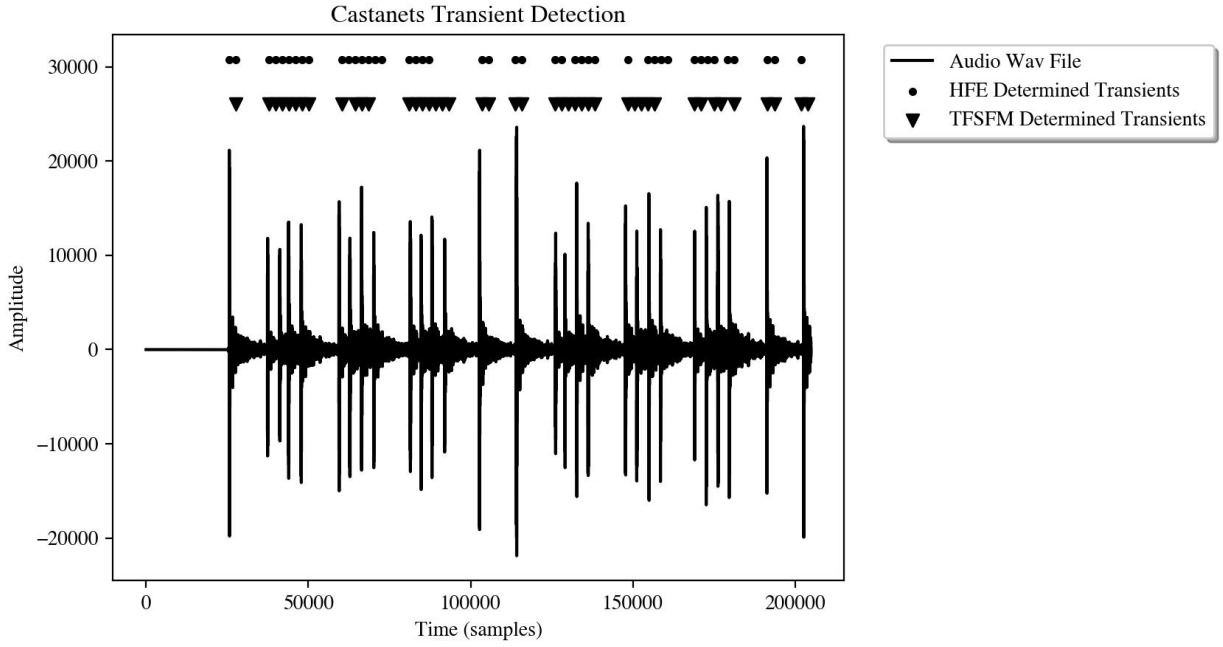

Fig. 3. Transient detection on the castanets critical item

to calculate the metrics for both the current block and the previous block and then take the difference between the two. This difference is then checked against a threshold value to determine whether or not the current block contains a transient. The threshold proposed in Fan et al. for TFSFM was 0.6 [6]. For these audios, we found that this threshold was slightly too sensitive and detected transients when there was not a transient in the block. After some tuning, we ended up using an increased threshold of 0.75. Unfortunately, when just using TFSFM alone, transients were still detected far too often, so combining TFSFM and HFE resulted in transients being more accurately identified. As proposed in Fan et al., a threshold that can be used for HFE is 10 [6].

Before calculating the TFSFM and HFE metrics, the signal is low pass filtered with a cutoff frequency of 8 kHz. We use a butterworth highpass filter to perform this operation

Figure 3 shows the transient detection methods on the castanets critical item. It can be seen that the TFSFM does detect significantly more transients than the HFE method. Figure 4 shows the transient detection methods on the glockenspiel critical items. Here, you can see that the TFSFM metric determines that a block contains a transient especially in the resonance region of this audio. However, you can see that the HFE metric accurately determines the transients for this audio. The utility of both of these metrics are are shown in Figure 5 where the transients are more randomly spread for speech.

B. Block Switching

Once a transient is detected, the block transitions to a shorter block. For this implementation, we used the AC-2A transition windows. For the longer blocks, we used the KBD windows. For the shorter blocks, we used Sine windows. The length of the longer windows is 2048, while the length of the shorter windows is 256. In Figure 6, we can see what the windows look like when switching from a long block to an AC-2A transition window to a full block of short windows and back to a long window. As shown, we use a short window length that divides evenly into the long window length allowing 8 short blocks to fit in the length of a longer block.

For this implementation, we keep track of the current and previous block as well as if the next block contains a transient or not. If a transient is detected in the current block, the previous block either transitions to a short block using the AC-2A transition windows or continues being a group of 8 short blocks. The state of the blocks are stored using the enum type according to Table III. Every time a

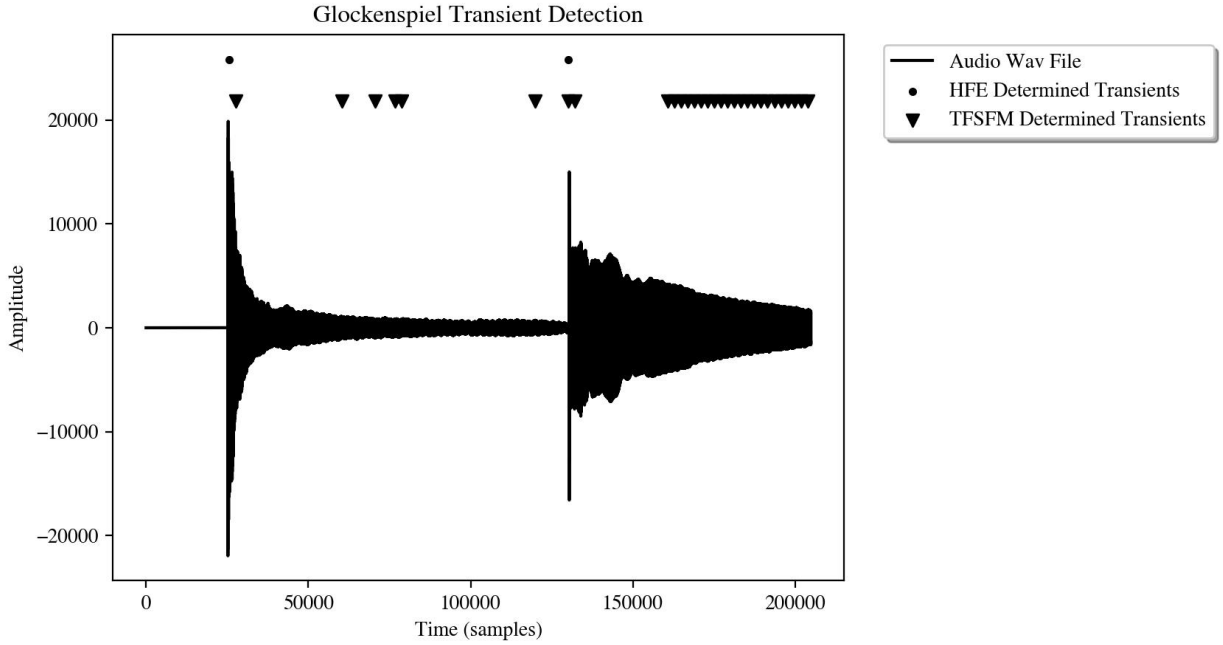

Fig. 4. Transient detection on the glockenspiel critical item

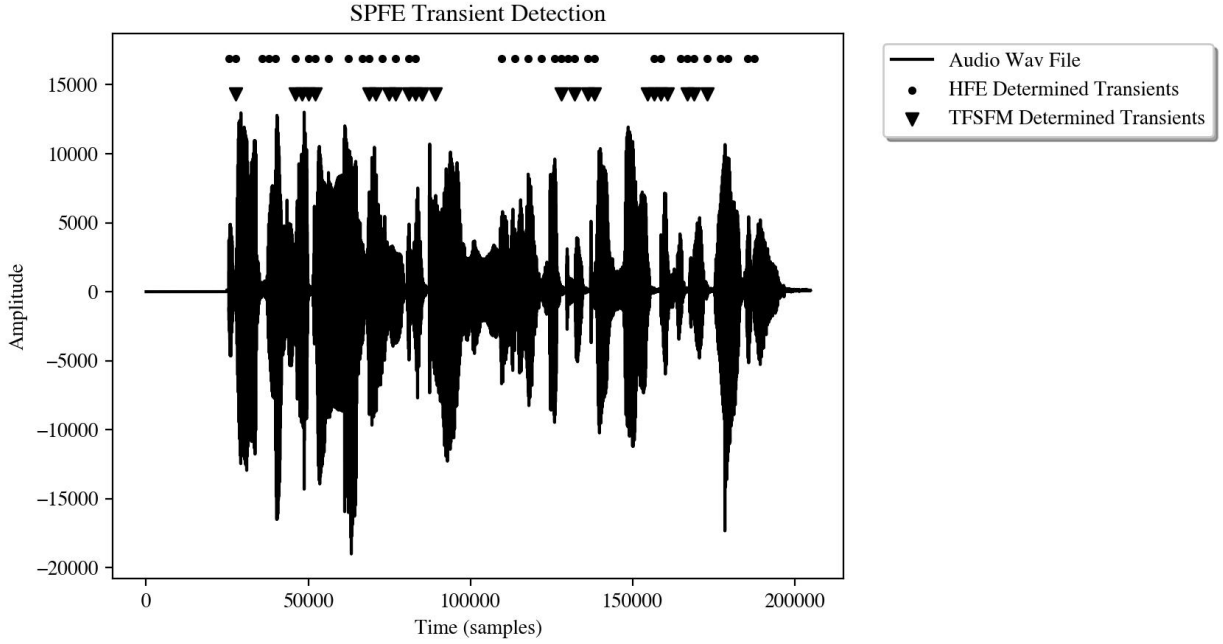

Fig. 5. Transient detection on the English speaker critical item

next block is read in, the previous block and current block state gets updated and stored accordingly.

There are several values which need to be updated based on the block size. For example, during a transition window the number of MDCT lines changes to be the average of the number of MDCT lines for long and short blocks. This changes the sfBands variable throughout the code as well. As a result, the new nMDCTLines and sfBands variables needs to be changed in the codingParams and the headers. Therefore, we are now passing more information into the headers which then need to be parsed out. Additionally, the block state needs to be saved before the sfBands information is saved to

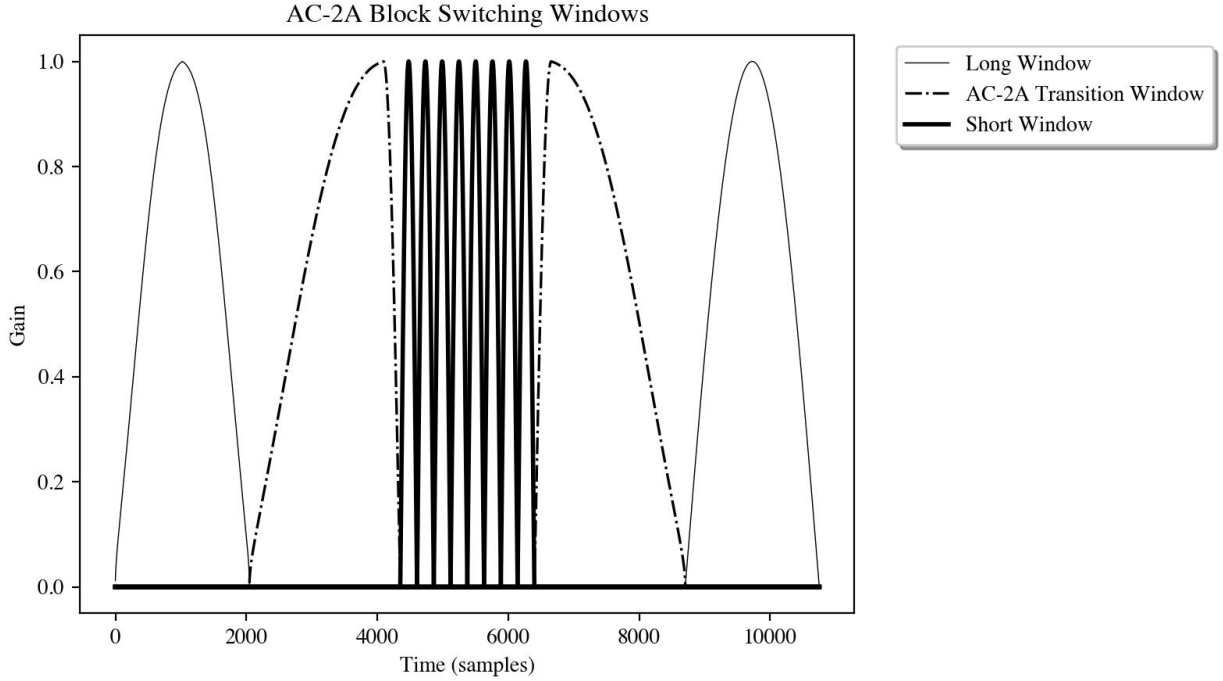

Fig. 6. AC-2A Window Switching Explanation

TABLE III
BLOCK STATE KEY

Block State Enum Value	Window Type
0	Long
2	Long to Short
3	Short
4	Short to Long

the header because this will then allow the correct number of MDCT lines and scale factor bands to be chosen when decoding. This information is saved as a two bit value since there are four possible states as described in Table III.

C. Entropy Coding

Initially, we wanted to implement both Arithmetic Coding and Huffman Coding and compare the two to see which performs better. However, since we had limited time during our project, we wanted to try to implement Arithmetic Coding with our baseline perceptual audio codec since this would be a new feature previous iterations of the class have not implemented. However, as we continued to try and implement Arithmetic Coding, we saw that it would require a more rigorous overhaul of our baseline codec which would require more time than the allocated project timeline. As such, we were more confident that Huffman Coding would be better to implement given our timeframe and baseline codec.

Arithmetic Coding Attempts

For Arithmetic coding, we attempted to implement this entropy coding algorithm with our baseline codec. However, we were met with many tribulations and logistical difficulties.

We first implemented our own Arithmetic coder in which we adapted from the Stanford Compression Library. All three of us have worked with this codebase before and knew extensively how it worked. [10]

We rigorously tested how our Arithmetic coder behaves, such as how it encodes and decodes different symbols and how we could retrieve them without losing information. We were confident that our Arithmetic coder works well, but when we tried integrating our Arithmetic coder with the baseline codec, we encountered many roadblocks.

For our first attempt, we tried integrating through "codec.py" itself with encoding on a per-symbol basis. Since Arithmetic coding relies on encoding each symbol based on its probability of occurrence and using the probabilities as the bit string, we pre-calculated the probabilities, and used these pre-

calculated probabilities to encode and decode all of the symbols. Then, once we know the length of the bit string per block per channel, we would designate that as the number of bits we needed to allocate when passing it to write the bits. However, we noticed multiple issues that arose:

1) Since the probabilities of the mantissas occurring within that block had to be encoded as one single bitstring, it would result in a huge bitstring. This was especially true for blocks that had mantissa values with a large range and heavily varied frequencies of occurrences. Because of this, we noticed that Arithmetic coding might not save us bits. Especially since we noticed that the bit string for the probabilities was already over 1000 bits long for only one channel in one block compared to the baseline codec which allocated a little over 200 bits for a block.
2) We also noticed that it was difficult to write the bits since the baseline codec relied heavily on writing the mantissa values on a per band and per line basis. With this methodology, we did not know which line and which band the mantissa values came from, so when we had to decode these values, we were unsure of how to essentially slot these values in the correct places. Since we could not distinguish what was our Arithmetic encoded bits and which was our baseline codec's bit allocated bits, it more difficult to switch between methodologies if Arithmetic coding was not saving us bits.

Another methdology we tried was to interact with the read and write bits directly. Once we got the mantissa values from codec.py's Encode, we would ignore the pre-allocated bitalloc from BitAlloc.py and create our own bit allocation strategy to write bits to using our Arithmetic coding. We thought this method would work better as we would know which band and line the mantissa values came from, so it would be easier to interact with the read and write bit functions themselves. However, we ran into many issues with encoding way too many bits since the Arithmetic coder is still encoding the probabilities of all of the mantissa values it's trying to encode in that block. One major issue we ran into was with bitpacking our bits. For example, when we would try to encode the large values obtained from the Arithmetic encoder, the bitpack functions would sometimes wait until the least significant bit to write the information to and would then throw an error. The codec would think that there the most significant bits were full and that no more information could be written to them, so it would think that it would have to write the really large value to the least significant bit.

Our third method was trying to write the Arithmetic coded value as a floating point value to represent the binary instead of an integer value to represent the binary. We thought that this method could possibly reduce the amount of bits needed to write to the PAC file in bitpack.py. However, we then ran into an issue of floating point precision error. Where, given that our probability tables were very large, it could not code the probabilities properly given that the float64 datatype can only hold 64 bits. So, when we tried to have the Arithmetic coder encode the floating point value from the probability tables, we could not achieve an accurate enough precision for the Arithmetic coder to encode and decode the values properly.

After many attempts, we decided to switch to Huffman Coding for our Entropy-based compression.

Huffman Coding

We implemented our own Huffman coder in which we adapted from the Stanford Compression library [10]. We have worked with the codebase for this coder and knew how it worked. We tested this coder to make sure we could encode and decode a set of symbols given that we provided it a codebook containing the frequency of occurrences of these symbols. From the frequency of occurrences table, we could generate a probability distribution of these symbols as our Huffman tables, and can losslessly get back our encoded symbols.

When integrating the Huffman coder, we tried a couple of methods. Fortunately, because we tried implementing our Arithmetic coder prior to attempting integrating our Huffman coder, we had a better understanding of how to write bits to the PAC file.

For our first methodology, we wanted to encode each symbol using the Huffman tables. Since we knew that we would not have all of the mantissa values in our tables, we knew that we would have to switch from Huffman encoding for bit allocation to the base codec bit allocation. Initially, we thought

it would be easiest to save the locations of where we switch from Huffman coding bit allocation to the baseline codec bit allocation (and vice versa). We thought that this would be easier since we would have the exact locations. However, we realized that saving these values externally would not represent Huffman coding well.

So, for our second methodology, we decided to write our Huffman encoded mantissa bits and our baseline encoded mantissa bits to the bitstream together, but with a known Huffman encoded symbol to know when to switch between the two bit allocation methods.

This symbol is what we call an "Escape Code" and we encode this symbol using Huffman coding whenever we have to switch from encoding in Huffman to the baseline codec's bit allocation method. [3] Since we pre-calculated the tables beforehand, we knew the counts of the mantissa values and were able to create a frequency table and symbol dictionary for most of our audio samples. We could not use Huffman coding for all of our mantissa values, because it would cause us to have too large of a table resulting in longer bit strings when encoding. This lead us to inserting our own escape code into each table with the respective count. This count is determined by the mantissa values we are not including in our table and their associated counts. The table for our different Huffman table sounds and the escape code count can be found in Table IV. After inserting these escape codes into their respective tables, we can generate the probability distribution for the Huffman coding's use. Using these tables, we know that whenever an escape code was decoded, we switch from Huffman decoding to the baseline codec's bit allocation decoding. With this method, we did not have to save which block, channel, band, and line we encoded the mantissa from since all of the information would flow linearly from encode to decode.

TABLE IV
HUFFMAN CODING ESCAPE CODE

Sound	Escape Code Count
Castanets	119375
SPGM	2806
Oboe	2584
Harpsichord	5026
Glockenspiel	5191
Quartet	4567
SPFE	4053

Once we have the escape codes within our bitstream, we were able to switch between our baseline codec's bit allocation method and our Huffman based bit allocation and representation method. [3] However, since we are switching between the baseline codec's bit allocation and the Huffman bits that we're writing to the PAC file, we would also need to adjust how many bits we're expecting in the PAC file. Since we are performing Huffman encoding and decoding during the bit writing stream itself, we are unable to determine the number of bits ahead of time without encoding our stream of mantissa values and then obtaining the length of that bit string. Therefore, our implementation calculates the Huffman encoded representation and length before writing the number of bits to the PAC file.

In our current implementation, we have manually picked which Huffman table we are using to encode our data. We created a pre-trained Huffman table for each of our audio samples and used the corresponding Huffman table for the encoding and decoding of all of our audio samples. Eventually, the goal is to have the audio coder automatically select the best Huffman table to use. This would be done by calculating the number of bits it would take to encode a whole block for each table, and choosing the table that would take the fewest number of bits to encode the block. This would ensure that we are always picking the best Huffman table to encode with and would allow us to get maximal compression. The reason we chose not to implement this is because our code already takes a long time to run when just comparing Huffman encoding to non-Huffman encoding per block, so adding the additional calculation of choosing which Huffman table to use for encoding would significantly increase the runtime of our coder.

With this implementation, we are able to fully encode and decode all of the mantissa values for each block via Huffman coding. We were also able to verify that using Huffman coding ensured that we losslessly retrieved the symbols that we wanted to retrieve, and that it sounded similar to our baseline codec.

VII. RESULTS

We ran a MUSHRA test to compare the results of our improved audio coder with our baseline audio coder for the a 96 kbps data rate and 128 kbps data

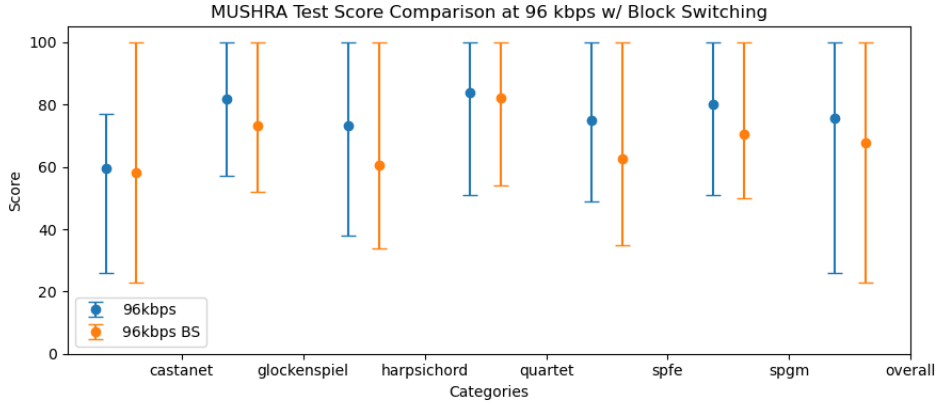

Fig. 7. MUSHRA scores for 96 Kbps with and without block switching and Huffman Coding

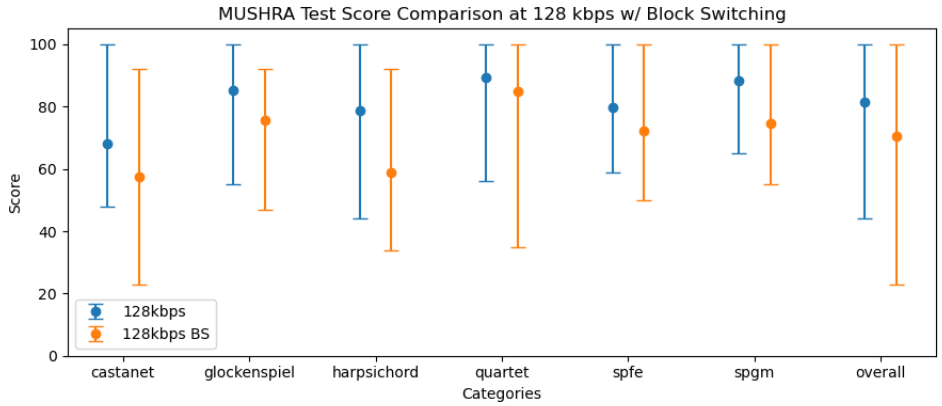

Fig. 8. MUSHRA scores for 128 Kbps with and without block switching and Huffman Coding

rate. We used 6 different audio samples for our test (castanets, glockenspiel, harpsichord, quartet, spfe, spgm) and encoded each audio sample in the following 4 ways:

1) 96 kbps without block switching and Huffman coding
2) 96 kbps with block switching and Huffman coding
3) 128 kbps without block switching and Huffman coding
4) 128 kbps with block switching and Huffman coding

The demographics of our MUSHRA tests consisted of 4 females and 7 males some of whom have not taken our MUSHRA test previously. The ages ranged from 23 to 28 and have average hearing. None of the participants are expert listeners. The test performed with high quality headphones that were not noise cancelling. The participants received detailed instructions before the test started on how to rate each of the audio samples in comparison to the reference. The participants sat in a quiet room, but it was not completely devoid of noise due to external noise sources outside of our control.

The results of our MUSHRA tests have been consolidated into plots for the 92 kbps and 128 kbps comparing block switching and Huffman Coding to our original baseline codec. These results are shown in Figures 7 and 8. From our results, we can see that block switching and Huffman coding did not end up improving our audio quality significantly. However, we did hear that especially for castanets, the majority of the perceptible pre-echo is removed, but there were several instances where the transients were not detected accurately which resulted in a perceptible ”glitch” in the audio. This was very clearly evident to the users despite the majority of the pre-echo being removed. Therefore, we suspect this is why the block switching was sometimes rated lower than the baseline coded audios. We do see that

the variance increases when block switching and Huffman are used. This could be due to people's preferences on how the "glitching" noise affects the audio quality.

The next aspect of our perceptual audio codec we looked at was the compression ratios between using Huffman coding and block switching to our baseline codec that does not have these features.

TABLE V
COMPARISON OF COMPRESSION RATIOS AT 96 KBPS

Sound	Huffman + Block-Switch	Baseline
Castanets	7.088	7.088
SPGM	7.16	7.22
Harpsichord	7.5	7.17
Glockenspiel	7.222	7.458
Quartet	7.12	7.23
SPFE	7.12	7.20

TABLE VI
COMPARISON OF COMPRESSION RATIOS AT 128 KBPS

Sound	Huffman + Block-Switch	Baseline
Castanets	5.32	5.31
SPGM	5.34	5.38
Harpsichord	5.65	5.38
Glockenspiel	5.3	5.57
Quartet	5.27	5.41
SPFE	5.32	5.37

It seems that Huffman coding only improved our compression ratio for a few audio samples, mainly harpsichord. We suspect this is because block switching is a very bit intensive process, so since harpsichord does not contain transients, we had no need to use block switching, thus allowing us to get a better compression ratio. For audio samples with lots of transients such as castanets, we did not see an improved compression ratio due to the large amount of transients present in the audio. The more transients detected in the audio increases the amount times we have to use our block switching, thus increasing the size of our PAC file.

VIII. CONCLUSIONS

Overall, we are able to create a successful perceptual audio codec that implements three additional features from the baseline codec. We implemented transient detection, block switching, and Huffman coding (as well attempted to implement Arithmetic coding).

Although our test results indicate that some of our test takers were able to distinguish the difference between block switching with Huffman coding compared to one without in both 96 kbps and 128 kbps data rates, we have a working perceptual audio codec that implemented the additional features and was able to encode and decode the audio successfully with slightly minimal artifacts given the time constraints.

IX. FUTURE WORK

In the future, we would like to optimize our code so it runs faster. The first step would be to optimize the Huffman coding implementation. Currently, the code runs Huffman encoding twice: first to encode the symbols and calculate the length of the encoded symbols' bit strings, and secondly to encode the symbols and write them to the PAC file. This is an inefficient approach and can be more optimized if the number of bits needed for the symbols in the current block can be calculated ahead of time. Another optimization we could make is in terms of block switching. Currently, we read the data blocks of different sizes multiple times in a "larger" block. However, since we run our Huffman encoding algorithm inside of our write and read data block methods, this would cause the code to run Huffman encoding multiple times which could take even longer. One way to improve this is to optimize how many times we call write and read data block or another way is to improve how Huffman coding encoded and decoded its symbols. Additionally, we would like to experiment more with transient detection methods to make sure that our transient detection is more robust and accurate.

REFERENCES

[1] Markus Erne. Perceptual audio coders quot;what to listen forquot;. In *Audio Engineering Society Convention 111*, Nov 2001.

[2] David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[3] M. Bosi and R. E. Goldberg. *Introduction to Digital Audio Coding and Standards*. I. Kluwer, 2003.

[4] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[5] John G. Cleary Ian H. Witten, Radford M. Neal. Arihtmetic coding for data compression. *Communications of the ACM*, 30(6), 1987.

[6] Senyuan Fan, Emily Kuo, Sneha Shah, and Marina Bosi. Transient detection methods for audio coding. In *Audio Engineering Society*, Oct 2023.

[7] A. Gray and J. Markel. A spectral-flatness measure for studying the autocorrelation method of linear prediction of speech analysis. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(3):207–217, 1974.

[8] B. Edler. Coding of audio signals with overlapping transform and adaptive window shape. *Frequenz*, 43(9):252–256, 1989.

[9] Davidson Bosi. High-quality, low-rate audio transform coding for transmission and multimedia applications. In *93rd AES Convention, J. Audio Eng. Soc. (Abstracts)*, Dec 1992.

[10] Pulkit Tandon Chacha Chaudhary Tao Jin Kedar Tatwawadi, Shubham Chandak. Stanford compression library, 2021.

X. Appendix

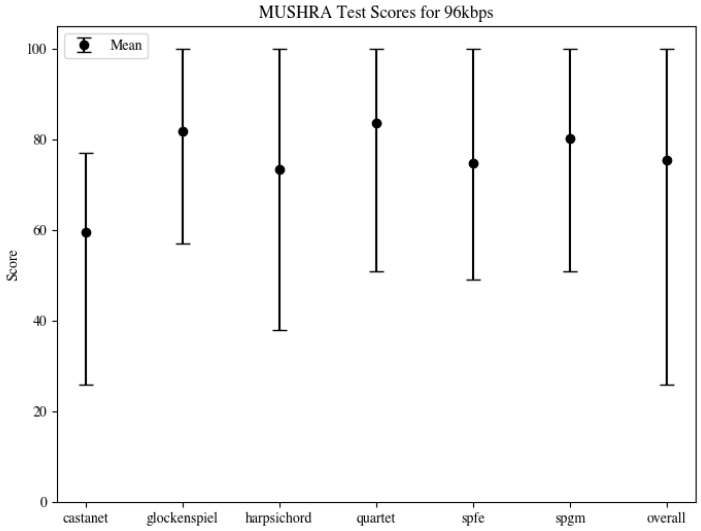

Fig. 9. MUSHRA scores for 96 Kbps

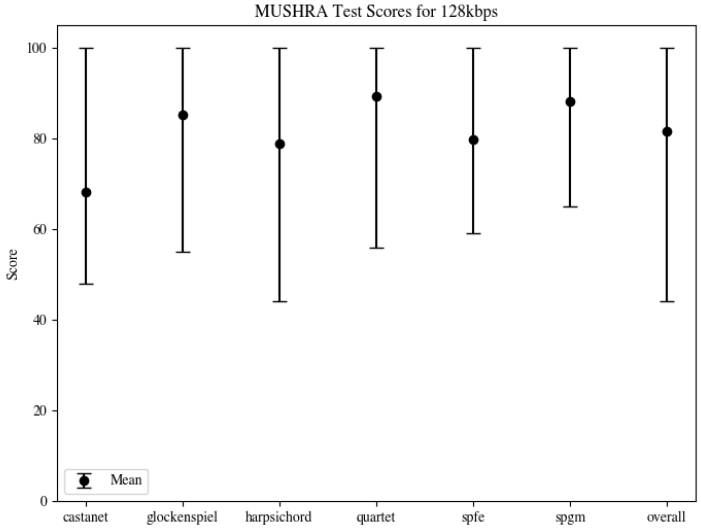

Fig. 10. MUSHRA scores for 128 Kbps

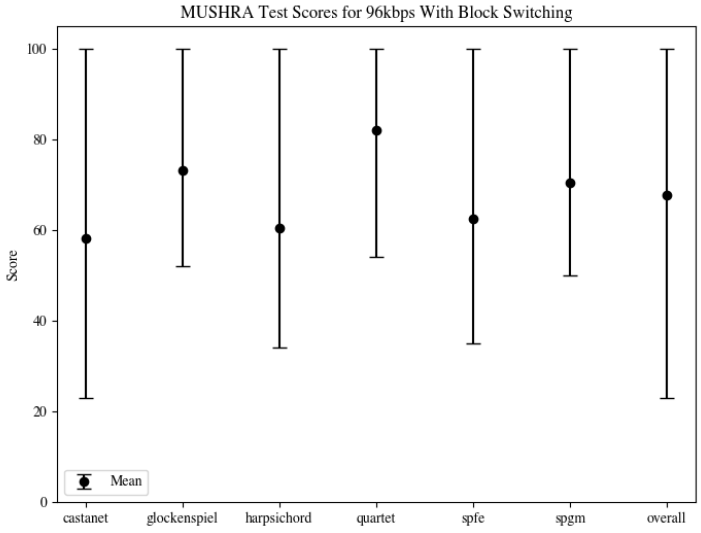

Fig. 11. MUSHRA scores for 96 Kbps with Block Switching

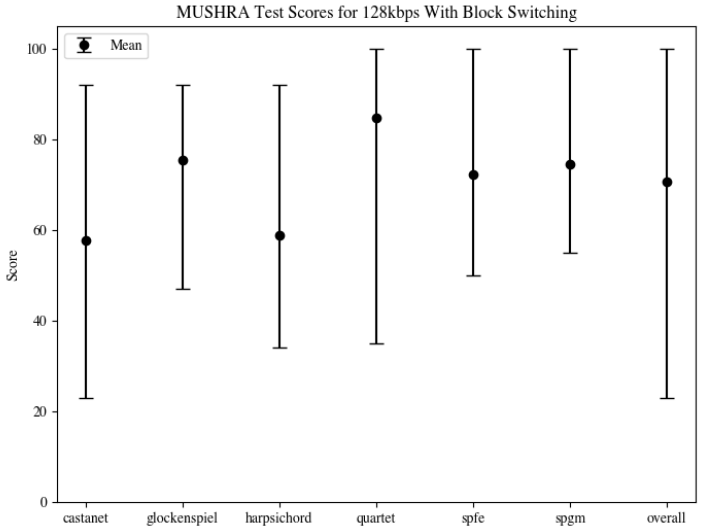

Fig. 12. MUSHRA scores for 128 Kbps with Block Switching