A-Law Audio Compression

Ryland Nezil, UVic Dept. of ECE, V00157326 rnezil@uvic.ca

David Van Acken, UVic Dept. of ECE, V00998703 dvanacken@uvic.ca

.....

Table of Contents

Introduction

Audio compression[1] is the process of reducing the dynamic range of a signal. In layman's terms, this means making the loud parts quieter and the quiet parts louder while preserving the original sound as best as possible. This process has several applications, including modifying the sound of electrified musical instruments and minimizing the storage required for sound files; the latter is the focus of this project. Moreover, UVic students Ryland Nezil and David Van Acken have collaborated on the development of an optimized program for compressing WAV files[2] via the A-law algorithm[3]. This algorithm defines rules for compressing and expanding (often called "companding"[3]) digital audio signals. Compressed WAV files must undergo an expansion process (i.e. reconstructing the original signal from its codified A-law representation) before they can be played back. This process is simply the inverse of the compression process.

Theoretical Background

Familiarity with analog and digital audio signals and analog-to-digital conversion (ADC) is a prerequisite to understanding the A-law companding algorithm. Analog signals, by definition, are continuous; digital signals, on the other hand, are discrete. All "real" sounds, that is, sounds perceivable by the human ear, are analog. Ears pick up fluctuations in air pressure (called pressure waves) caused by, say, the vibration of a guitar string, and the brain converts this to sound. The volume and tone of the sound that is ultimately heard depends, respectively, on the amplitudes and frequencies that make up the pressure wave. Therefore, all sounds are defined entirely by their waveform. One major implication of this fact is that sounds can be encoded as electrical signals; as a voltage wave instead of a pressure wave, and thus in a format compatible with computers. Computers, however, cannot directly store analog signals, as doing so would require the preservation of an infinite number of samples recorded at differentially small time intervals. Engineers overcome this problem through the digitization of audio signals, which amounts to approximating the original signal by capturing samples at a set frequency. Each sample gets encoded as an N-bit integer, whose sign and magnitude are proportional to the amplitude of the original signal at the time the sample was recorded. The value of N varies from one ADC unit to another, but 16 bits is common for audio and thus was used in this project. It follows that, for a sampling frequency f, every second of recorded audio produces $16f$ bits of raw data. For a typical sampling frequency of 44.1KHz, that is equivalent to 88.2 kilobytes per second, or 5.3 megabytes per minute. Large files are often undesirable, thus the need for file compression arises.

The A-law companding algorithm, which produces 8-bit compressed samples, is compatible with sample sizes equal to or greater than 13 bits (samples greater than 13 bits will be truncated by chopping lower bits). For a sample x, the A-law representation $F(x)$ is given by[3]:

$$F(x) = \operatorname{sgn}(x) \begin{cases} \dfrac{A|x|}{1+\ln(A)}, & |x| < \dfrac{1}{A}, \\ \dfrac{1+\ln(A|x|)}{1+\ln(A)}, & \dfrac{1}{A} \leq |x| \leq 1, \end{cases}$$

Figure 1. A-law compression definition.

The value A in *Figure 1* is called the compression parameter, and its standard value is 87.6. In practice, it is cumbersome to apply this equation to each sample. Instead, compressed samples are computed according to *Table 1* below[4]:

Biased Input Values in Signed-Magnitude Representation (sign bit, 12 magnitude bits)													Compressed Code Word (sign bit, 3 chord bits, 4 step bits)							
s	11	10	9	8	7	6	5	4	3	2	1	0	s	6	5	4	3	2	1	0	Chord
0/1	0	0	0	0	0	0	0	a	b	c	d	×	1/0	0	0	0	a	b	c	d	1st
0/1	0	0	0	0	0	0	1	a	b	c	d	×	1/0	0	0	1	a	b	c	d	2nd
0/1	0	0	0	0	0	1	a	b	c	d	×	×	1/0	0	1	0	a	b	c	d	3rd
0/1	0	0	0	0	1	a	b	c	d	×	×	×	1/0	0	1	1	a	b	c	d	4th
0/1	0	0	0	1	a	b	c	d	×	×	×	×	1/0	1	0	0	a	b	c	d	5th
0/1	0	0	1	a	b	c	d	×	×	×	×	×	1/0	1	0	1	a	b	c	d	6th
0/1	0	1	a	b	c	d	×	×	×	×	×	×	1/0	1	1	0	a	b	c	d	7th
0/1	1	a	b	c	d	×	×	×	×	×	×	×	1/0	1	1	1	a	b	c	d	8th

Table 1. A-law compression table.

As shown above, the resultant compressed sample consists of a:

- sign;
- chord; and
- step.

The chord is a function of the number of leading zeros, the step is a copy of the 4 bits following the leading zeros, and the sign bit is simply the negation of the original sign. Despite only being 4 bits, the step effectively encodes 5 bits as it is known that, except in the case of 7 leading zeros, the first bit following the leading zeros will hold the value 1.

Design Process

The design process consisted of five distinct phases:

1. devising specifications;
2. learning to work with WAV format;
3. implementing the A-law in C;
4. optimizing the code; and
5. optimizing the architecture.

These are detailed below.

Specifications

First and foremost, the program must be able to generate a compressed WAV file from any given compressible WAV file, and then be able to decompress that file. Compression and decompression must be performed using the A-law algorithm. The compressed file must retain enough quality that, when played back, it is immediately recognizable as a copy of the original audio. For example, if the original audio is someone speaking ten clearly audible words, then those ten words must also be audible and clear enough to understand in the compressed audio. Finally, the program must be optimized for execution on the ARM Cortex A15 ARMv7-A architecture.

WAV Format

The WAV (pronounced "wave") file format was developed in the early 1990's by Microsoft and IBM for the purpose of storing uncompressed audio. WAV files consist of a short header followed by the samples, which are sequentially stored such that the last sample of the file corresponds to the last sample recorded by the ADC unit. The header contains information about the file, including the:

- file size;
- header size;
- number of bits per sample;
- average bytes per second;
- sample format (for example, A-law compressed);
- number of audio channels; and
- sample rate.

This information is stored in binary, not ASCII, so a hex editor was frequently needed for manual inspection of these headers. Outputting well-formed WAV headers was initially challenging, but this problem was eventually resolved through byte-by-byte debugging. It was found that, for compressing a WAV file with N-bit samples and M channels, the only changes necessary to the header were:

1. 'file size' = ('file size' - 'header size') / (N / 8) + 'header size'

2. 'number of bits per sample' = 8;
3. 'average bytes per second' = 'sample rate' * M; and
4. 'sample format' = 6, which corresponds to A-law compressed format.

With reliable mechanisms in place for dealing with headers, all that was left in regards to WAV navigation was parsing audio samples. As mentioned earlier, WAV format stores audio samples sequentially, where the beginning and end of the audio signal correspond respectively to the first and last audio samples stored in the file. This enabled a straightforward parsing process which amounted to iterating forward over the file, beginning at the first sample.

A-Law in C

After finding consistent success in the parsing of WAV files, work began on implementing the A-law compression algorithm. This implementation was found to require 8 steps:

1. read sample from input;
2. truncate sample to 13 bits;
3. extract sign;
4. compute number of leading zeros;
5. compute chord;
6. extract step;
7. prepare compressed sample; and
8. write compressed sample to output.

Decompression was then easily realized by performing the inverse of the compression process. None of the steps needed were particularly complicated, and thus the details of how they were implemented are not discussed. The code to decompress is loosely based on the The ITU-T Software Tool Library [5], with modifications made in order to increase speed when executing on ARM hardware. Several steps benefit greatly from certain optimizations, and these are discussed in the following sections. The program itself was organized according to the UML diagram shown below in *Figure 2*.

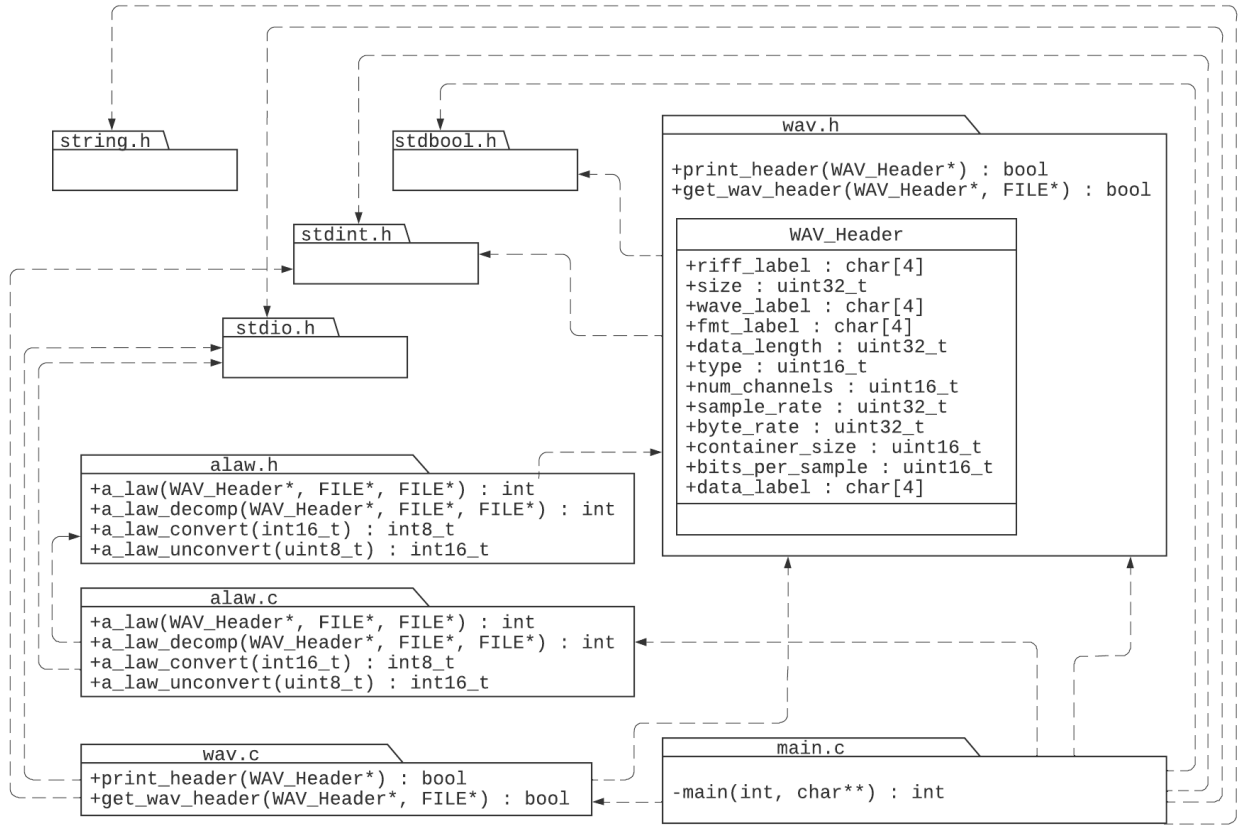

Figure 2. UML diagram showing the organization of the A-law program.

As shown above, the main executable is compiled with the wav.h and alaw.h headers, and linked against the alaw.c and wav.c libraries. The wav.h header defines the WAV_Header type, which allows for a software representation of WAV file headers. wav.h also declares four functions:

1. print_header, which prints the data of a WAV_Header object to stdout (this function is only used for debugging); and
2. get_wav_header, which, given a file, populates a WAV_Header object with the data from the header of a WAV file.

The wav.c library defines both of these functions. The alaw.h header declares four functions:

1. a_law, which handles file I/O and sample parsing for the compression process;
2. a_law_decomp, which handles file I/O sample parsing for the decompression process;
3. a_law_convert, which compress a given sample; and
4. a_law_unconvert, which decompresses a given sample.

These functions are defined in the a_law.c library.

Software Optimization

Given the inherent simplicity of the algorithm, optimizations were chiefly gained from fine-tuning certain operations in the main sample processing loop. For instance, it was discovered that the targeted processor architecture supports the CLZ instruction[6], which computes the number of leading zeros in a given register.

Function Inlining

Initially, each of steps 3 through 7 relied on calling specific functions that would complete the given step and then return. This setup facilitated debugging, but once the program was consistently performing as intended these functions were all manually inlined into the main loop. As a result, the number of function calls required to process a sample was reduced from 7 to 2, the remaining 2 being non-negotiable function calls which read and write the input and output files. Function calls compile to unconditional branches, which cost 3 clock cycles[7]; therefore, eliminating 5 branches saves 15 clock cycles per iteration.

Getting Sign

The sign of the data can be extracted in several ways. The most simple, but not the fastest, is an if/else check, as seen in *Figure 3*. This method is excellent for readability, however it will be compiled down to branch instructions, as seen in *Figure 4*. The result of this is that the "bge" instruction will take somewhere between 5 and 7 clock cycles if the incorrect branch is predicted.

```
static uint8_t get_sign(int16_t num) {

        if(num < 0)
                return 0x00;
        else
                return 0x80;
}
```

Figure 3: If/else to find sign.

```
        cmp     r3, #0
        bge     .L16
        mov     r3, #0
        b       .L17
.L16:
        mov     r3, #128
.L17:
        mov     r0, r3
        add     sp, fp, #0
```

Figure 4: If/else compiled to branch instructions.

One way to speed up this section of code is to assign 0 to the sign variable to start, then replace the if/else statements with inline assembly that assigns 0x80 to the sign variable value if the value is positive. The resulting assembly instructions are found in *Figure 5*. The "ANDS" instruction will perform an AND operation on the provided number and a bitmask, in this case 0x8000. This uppermost bit will be a 1 if the number is negative, and a 0 if it is positive since the number is stored in 2's complement. The 'S' at the end of the "ANDS" means that the condition flags will be set with this operation. Consequently, a predicate operation can be used on the "MOV" instruction so that the value is only moved if the "ANDS" results in a non-zero number.

```
static inline uint8_t get_sign(register int16_t num) {
        register uint32_t result;

        __asm volatile (
                "ANDS %[result], %[num], %[sign_mask] \n"
                "       MOVEQ %[result], #0x80 \n"
                : [result] "=r" (result)
                : [num] "r" (num), [sign_mask] "r" (sign_mask)
                : "cc"
        );

        return result;
}
```

Figure 5: get_sign function using inline assembly.

Finally, the function was made inline in order to remove the overhead of calling the function. Special attention was paid to ensuring that inlining this function would not affect any other conditional instructions. The result of this change can be observed in *Figure 6*. Overall, this change will speed up the process of finding the sign significantly.

```
@ 164 "src/alaw.c" 1
        ANDS r3, r3, r2
        MOVEQ r3, #0x80

@ 0 "" 2
```

Figure 6: Inline assembly instructions to replace get_sign function call.

Getting the sign in this way reduces the instruction count from 9 instructions, 3 of which are branches, 1 of which are conditional, to 3 instructions (1 'UXTB' instruction needed due to data

size). This will reduce the cycle count from between 19 and 21 cycles to 10 cycles, according to the ARM technical reference [8].

Calculating the Chord

To calculate the chord, the number of leading zeros must be calculated. This can be thought of in two ways. One way would be to calculate the number of leading zeros on the number directly. The second way to calculate the chord is to perform a $\log_2$ operation, which could be done with the built in C math functions, or through other methods.

The first method of calculating the chord, counting the number of leading zeros, is relatively simple to implement in software, although the time required to perform the calculation is long since it requires checking if the correct bit is set, and if not, shifting the number up, and repeating. The code to perform this operation can be seen in *Figure 7*.

```
static uint8_t get_leading_zero_chord(int16_t val) {
        // Get chord according to upper bits
        uint8_t chord = 7;
        while( (!(val & 0x4000)) && chord != 0 )
        {
                val = val << 1;
                chord --;
        }

        return chord;
}
```

Figure 7: Calculating chord by counting zeros manually.

Some processors, like the one used in this project, include a faster way to perform this calculation in the form of a "count leading zeros" instruction. This instruction can be implemented for this purpose by getting the number of leading zeros once the value has been made positive, and the sign bit removed. This must be done since if the number is negative, the number of leading zeros will always be zero. Once the number of leading zeros is found, the chord can be calculated by taking 7, and subtracting the number of leading zeros plus the number of bits that are not relevant to the calculation. In the end, the equation works out to:

$$chord = 7 - (num\ zeros - 17) = 24 - num\ zeros$$

One more consideration for this calculation is if the input number is very small, and the number of leading zeros is greater than 7, then the chord should be zero. This was implemented in assembly by checking if the result is bigger than the maximum allowed value, and if so, setting it

to the maximum value. The whole function to perform all of these operations is shown in *Figure 8*.

```
static int8_t get_leading_zeros(register uint32_t val) {
        register uint32_t num_zeros = 0;

        __asm volatile (
                "CLZ    %[result],      %[input] \n"
        "       CMP     %[result],      #24 \n"
        "       MOVGT   %[result],      #24 \n"
                : [result] "=r" (num_zeros)
                : [input] "r" (val)
                : "cc"
        );

        return 24 - num_zeros;
}
```

Figure 8: Calculating chord using CLZ assembly instruction.

One final way to calculate the chord would be to calculate the $\log_2$ of the number. While it would be possible to implement this using C standard library math functions, this is unlikely to be performant enough. An alternative is to use a lookup table for the values. One issue that could be faced with this approach is that the lookup table would have to be very large in order to cover all 2^{16} possible values, however the lookup table can be reduced. By shifting the input values down, the lookup table can be reduced to just 128 values, as can be seen in *Figure 9*. Calculating the chord this way does come with the downside of having to check if the input value is above a threshold of 255, since anything below that will always look at the first value in the lookup table, which is intended for when the input is 256, so the result will be incorrect. This can be compensated for by performing a check, and setting the chord to zero if the value is below 256, as shown in *Figure 10*.

```
const static int8_t log_table[128] =
{
        1,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
        6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
        7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
        7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7
};
```

Figure 9: Reduced lookup table.

```
// If value will appear on log table (below 256 will always return 1 which should be 0)
if(val >= 256) {
        // Get leading zero chord from value
        chord = log_table[(val >> 8) & 0x7F];
        // Get step data from correct position, ignore other data (+3 because 16-bit)
        step = (val >> (chord + 3) ) & 0x0F;
        // Get the converted value
        converted = sign | (chord <<  4) | step;
}

// If value is less than 256, just take 4 lower bits of 13-bit number
else
        converted = sign | (val >> 4);
```

Figure 10: Finding chord using lookup table with if/else.

One further optimization can be performed with this method by checking if the value is less than 256, and clearing the chord bits if so. This optimization removes the need for the if/else check. The code for this can be seen in *Figure 11*. The LUT could also be extended, and 8 bits looked at instead of 7, but the increase in memory usage did not seem like a worthwhile tradeoff.

```
    // Get leading zero chord from value
    chord = log_table[(val >> 8) & 0x7F];
    // Get step data from correct position, ignore other data (+3 because 16-bit)
    step = (val >> (chord + 3) ) & 0x0F;

    // If the value is less than 256 clear the chord
    register uint32_t chord_long = chord;
    __asm volatile (
            "CMP    %[val],         #256 \n"
    "       MOVLT   %[chord],       #0 \n"
    : [chord] "=r" (chord_long)
    : [val] "r" (val)
    : "cc"
    );

    // Get the converted value
    converted = sign | (chord <<  4) | step;
```

Figure 11: Finding chord with if/else removed.

Architectural Optimization

Hardware optimizations were analyzed by simulating the addition of a custom instruction. The instruction, ALAW, computes the A-law compression of a sample stored in register RS and stores the sample in register RD. Inspection of the assembly code resulting from this instruction revealed that the salient bottleneck, that is, the sample processing loop, could be reduced from approximately 350 instructions per iteration down to 27. This yields an instruction reduction factor or nearly 13, which evidently provides a mighty performance boost. Such a boost was found to be unachievable by software optimizations alone, which implies that architectural optimizations are theoretically superior for A-law audio compression. Unfortunately, actually implementing these optimizations would likely be both extremely time consuming and extremely expensive, as the processor's datapath would need to be redesigned. This undertaking would undoubtedly demand a great deal of testing, and furthermore would incur the costs associated with producing new silicon.

Functional Overview

The program begins by enforcing that the 'bits per sample' field of the input file header is equal to 16, since the calculations assume a data length of 16 bits. If successful, the program then enters the main processing loop, which iterates over each sample in the input file. Every iteration results in the production of one compressed sample, which is written to the output file before the beginning of the next iteration. Once all samples have been processed, the compressed WAV header is generated and written to the output file. The program exits successfully upon writing the output header.

Performance/Cost Evaluation

The performance of the optimized versus the non-optimized code was analyzed using Valgrind. The optimized code saw a significant improvement over the non-optimized code. The total speed up can be calculated:

$$Speedup = \frac{4{,}300{,}315 - 3{,}331{,}290}{4{,}300{,}315} \times 100\% = 22.5\%$$

This speedup is mostly attributed to speedups in finding the sign of the value, and improvements to the calculation of the chord. The improvements to the magnitude function appear to be minimal since the number of assembly instructions required to compute the negative of a number is relatively high. The computation speed between using the CLZ instruction to find the number of leading zeros and using a lookup table to calculate the $\log_2$ appear to be similar. *Figures 12* and *13* below show results from the unoptimized and optimized code respectively.

In this case, the cost of the improvements is mostly a reduction in portability. The code is written for a specific architecture, ARMv7, and the code is incompatible with others such as x86 or even other ARM architectures. There are also disadvantages from a readability and maintainability point of view. Inlining assembly instructions makes the code harder to read, and has a higher potential to lead to issues that would be hard to track down, for example inline assembly that changes the state of the status registers at an inconvenient time. While the compiler should be able to handle these issues by keeping track of when changes to the status registers will have an impact on the program, it will ultimately be up to the programmer to ensure that the compiler generates valid assembly.

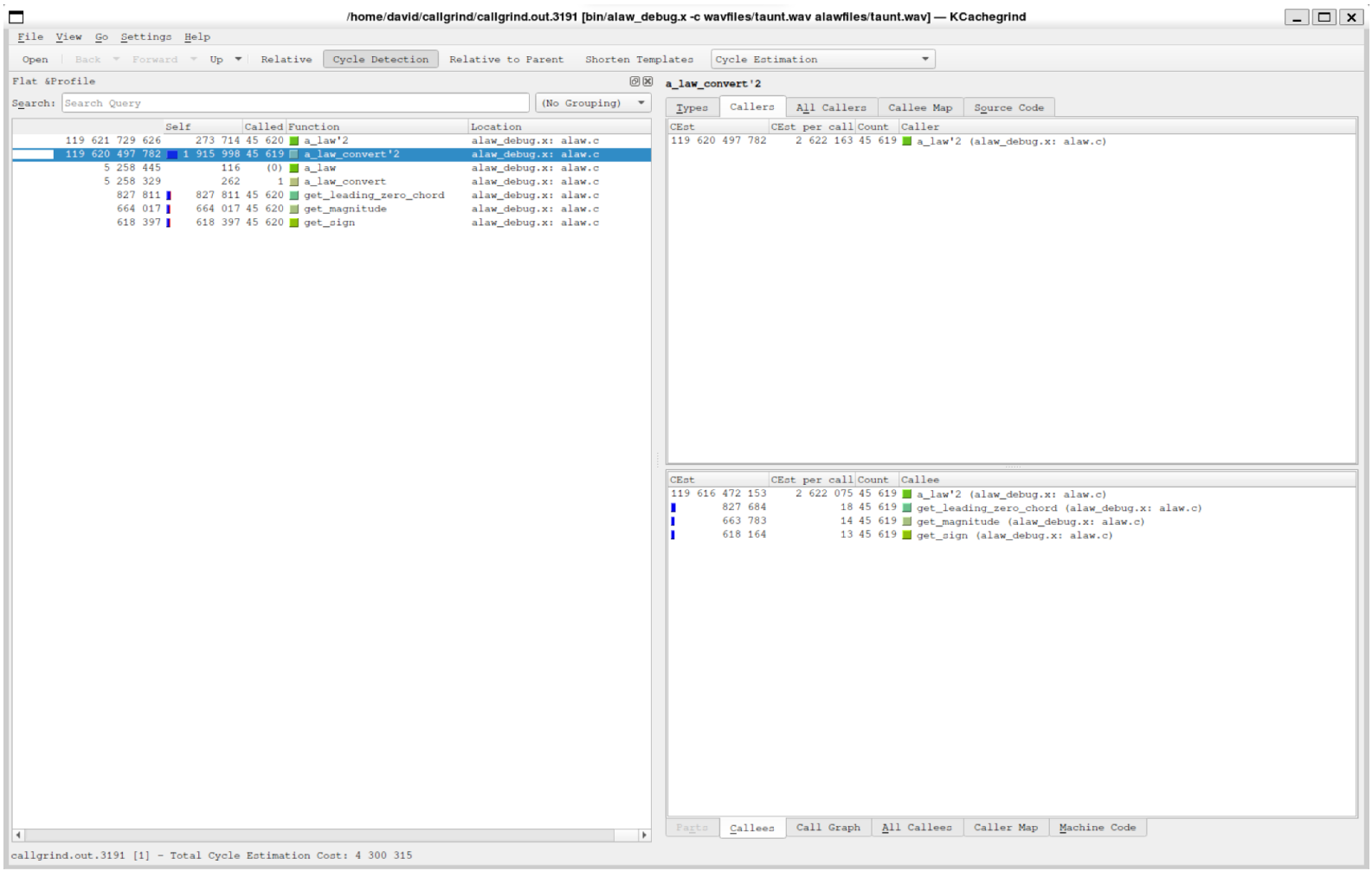

Figure 12. Unoptimized results.

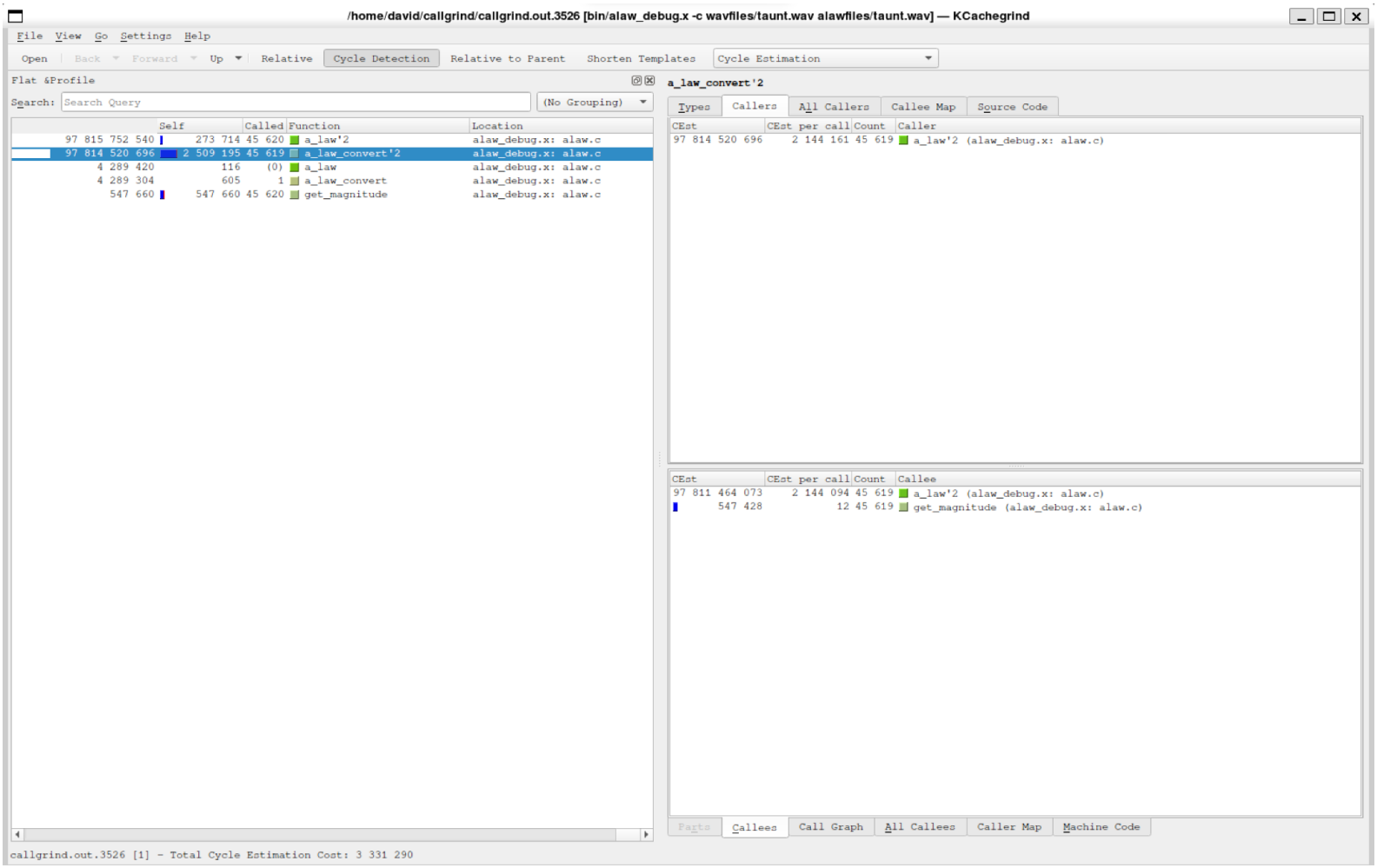

Figure 13. Optimized results.

Conclusions

Compression Quality

The quality loss incurred from using the A-law companding algorithm to compress WAV format audio files was significant. However, given that audio ADC is typically done with at least 16 bits, it follows that A-law compression can offer a 2x reduction in file size. This justifies the usage of compression in many practical applications, including file transfer and storage.

The reduction in quality is most apparent in the volume of the noise after conversion, and with the clarity of loud noises, such as emphasized words or letters. If a small, inaudible amount of noise is present in an uncompressed file, the noise is amplified to an audible level after conversion, and loud noises appear to be muffled, or clipped. Both of these phenomena are due to the lower number of bits being used after compression.

Hardware Support vs Firmware Support

While the performance gained from hardware optimizations proved superior to that gained from software optimizations, the resource cost associated with realizing such optimizations makes doing so much less appealing. Whereas Nezil and Van Acken were able to implement a multitude of effective software optimizations in a relatively short time period with no funding whatsoever, realizing the hardware optimizations described in this report would likely require a team of highly qualified, well-paid engineers. It follows that hardware optimizations should be restricted to adequately profitable ventures, and that software optimizations should be applied in all cases. In conclusion, it has been shown that while hardware optimizations provide notably better results, software optimizations shine in their simplicity and affordability, and are therefore applicable in any performance-focused context.

Bibliography

[1]https://www.researchgate.net/publication/2615393_Digital_Audio_Compression

[2]https://www.mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html

[3]https://www.dspguide.com/ch22/5.htm

[5]https://github.com/openitu/STL/blob/dev/src/g711/g711.c

[4]https://bright.uvic.ca/d2l/le/content/279730/viewContent/2269071/View

[6]https://developer.arm.com/documentation/dui0802/a/A32-and-T32-Instructions/CLZ

[7]https://developer.arm.com/documentation/ddi0222/b/instruction-cycle-times/instruction-cycle-count-summary

[8]https://developer.arm.com/documentation/ddi0301/h/cycle-timings-and-interlock-behavior/branches