Multithreaded Real-Time Audio Processor in C++

Pablo-Andres Pedicino, Evan Goldsmith, Arthur Lookshin, Daniel Gonzalez, Sreelekshmi Sreekumar
Undergraduate Students, Department of Computer Science
University of Central Florida
Orlando, FL, USA
{pa123047, ev565011, ar275475, da810538, sr165748}@ucf.edu

***Abstract*—This project presents a multithreaded real-time audio processor designed to enhance microphone input by applying various effects such as noise gating, equalization, de-essing, and limiting. Utilizing C++ threading and digital signal processing (DSP) techniques, we implement a strategic two-thread architecture to achieve low-latency, high-quality audio processing. DSP is crucial in transforming audio signals with precision, with effects applied sequentially in an optimized order through a dedicated processing thread, while audio I/O is managed separately. Our approach leverages modern multi-core processors to address the computational challenges of real-time audio processing, providing a robust solution that is suitable for practical applications. Performance is evaluated through benchmarks focusing on latency, audio quality, thread efficiency, and system load.**

***Index Terms*—Multithreading, real-time audio processing, C++, DSP, low-latency**

I. Introduction

An audio processor is a system that takes raw audio input and applies various effects and transformations to generate a modified output channel. These effects can include noise gates, equalizers, limiters, and much more. The main goal of an audio processor is to enhance the quality of audio in real time without interruption of the audio output.

Originally, audio was processed with analog hardware components like vacuum tubes and transistors, which were used to modify and enhance outputs. As digital technology advanced, software-based audio processing became the preferred method for modifying audio signals. The introduction of digital signal processing (DSP) greatly enhanced the performance of these software-based processors. Now, these digital audio processors run on operating systems and are integral to music production and broadcasting software.

Processing audio in real-time presents many computational challenges due to the number of continuous mathematical operations needing to be performed on a constant audio stream. It is important to maintain low-latency performance to prevent any artifacts, delays, or glitches in the audio. With the advent of multi-core processors, strategic use of multithreading has emerged as an effective approach to efficiently handle these performance demands. Multithreading allows modern audio processors to better organize tasks, separating intensive processing from time-critical I/O operations.

By leveraging C++ threading and audio libraries, we introduce an audio processing pipeline that applies various effects to enhance microphone input. A noise gate, 3-Band EQ, De-esser, and Limiter are applied to the audio input in a carefully designed sequential chain through a dedicated processing thread, while audio I/O is handled separately. This architecture uses thread-safe buffer queues with condition variables to manage the efficient handling of audio data. By separating time-critical I/O operations from intensive audio processing, we provide a low-latency, high-quality audio processor suitable for real-world applications.

To test the performance of our application, we use several benchmarks and tools to measure efficiency. The tests consist of four areas of focus: latency, audio quality, thread efficiency, and system load.

II. Methodology

A. Algorithms

Our audio processor incorporates interrelated DSP algorithms and a balanced architecture to process incoming audio streams in real-time. As a general strategy, we organize audio I/O and DSP processing into dedicated threads to reduce latency while maintaining processing integrity [1]. Our audio pipeline employs a main processing thread that sequentially applies effects in an intentional order: noise gating, equalization, de-essing, and limiting. This approach ensures each effect benefits from the processing of previous stages while minimizing thread synchronization overhead and maintaining phase coherence between processing stages.

A high-level overview of the algorithms involved for achieving each effect is as follows:

1) Noise Gate: Attenuates background noise below a defined threshold by analyzing 4 frequency bands [2]. Frequencies below the threshold in targeted bands are reduced. Attack and release coefficients smooth transitions and minimize artifacts.

2) 3-Band EQ: Splits the audio spectrum into low, mid, and high bands using bandpass filters. Biquad filters provide steeper cutoffs and smoother band transitions. Users can adjust the gain for each band independently [3] before the output is recombined.

3) De-esser: Targets sibilance typically found in the 4-10 kHz range. A bandpass filter detects frequencies in this range, and gain reduction is applied via a coefficient when the threshold is exceeded [4].

4) Limiter: To prevent audio clipping past a certain amplitude, we implement a look-ahead buffer to anticipate harsh audio peaks. Like the de-esser, we multiply the audio input by a gain reduction coefficient to avoid loud sounds [5].

Algorithm 1 Noise Gate

```
1: procedure CALCULATEGATEGAIN($inputBuffer$, $numFrames$)
2:     Copy input to FFT time-domain buffer
3:     Execute FFT to convert to frequency domain
4:     Initialize $bandEnergies$ array for NUM_BANDS
5:     for each frequency bin do
6:         Calculate band index based on logarithmic distribution
7:         Add squared magnitude to corresponding $bandEnergy$
8:     end for
9:     Calculate average energy across all bands
10:    Normalize energy based on FFT size
11:    if normalized energy $> threshold^2$ then
12:        return 1.0 (open gate)
13:    else
14:        return 0.0 (close gate)
15:    end if
16:    Apply attack/release smoothing to target gain
17: end procedure
18: procedure PROCESS($inputBuffer$, $outputBuffer$, $numFrames$)
19:    Calculate target gain
20:    for each sample do
21:        if $targetGain > currentGain$ then
22:            Apply attack smoothing
23:        else
24:            Apply release smoothing
25:        end if
26:        Apply gain to input sample
27:    end for
28: end procedure
```

Algorithm 2 3-Band EQ

```
1: procedure PROCESS($inputBuffer$, $outputBuffer$, $numFrames$)
2:     Shift previous input for overlap
3:     Copy new input to processing buffer
4:     Apply window function (Hann window)
5:     Execute forward FFT
6:     for each frequency bin including DC and Nyquist do
7:         Calculate frequency in Hz
8:         Determine gain using smoothGain function:
9:             For transitions between bands, apply cosine interpolation
10:        Calculate magnitude and phase
11:        Apply gain to magnitude
12:        Reconstruct real/imaginary components
13:    end for
14:    Execute inverse FFT
15:    Apply overlap-add with previous frame
16:    Output processed audio with 50% overlap
17: end procedure
```

Algorithm 3 De-Esser

```
1: procedure APPLYDEESSER($samples$, $sampleRate$, $startFreq$, $endFreq$, $reductionDB$)
2:     Calculate linear reduction factor from dB value
3:     Allocate FFT complex input/output arrays
4:     Create forward and inverse FFT plans
5:     for each frame in input do
6:         Fill input array with current frame (zero-pad if needed)
7:         Execute forward FFT
8:         for each frequency bin do
9:             Calculate corresponding frequency
10:            if frequency is in target range ($startFreq$-$endFreq$) then
11:                Multiply bin by reduction factor
12:                Apply same reduction to mirror bin (conjugate symmetry)
13:            end if
14:        end for
15:        Execute inverse FFT
16:        Normalize output and copy to result buffer
17:    end for
18:    Clean up FFT resources
19: end procedure
```

Algorithm 4 Limiter

```
1: procedure PROCESS($inputBuffer$, $outputBuffer$, $bufferSize$)
2:     If limiter disabled, copy input to output and return
3:     for each sample do
4:         Calculate sample absolute value
5:         Determine target gain:
6:             If below threshold: 1.0 (no limiting)
7:             If above threshold: $threshold/sampleValue$
8:         Apply gain smoothing:
9:         if $target < current$ then
10:            Attack phase: $currentGain = attackCoeff \times currentGain + (1 - attackCoeff) \times targetGain$
11:        else
12:            Release phase: $currentGain = releaseCoeff \times currentGain + (1 - releaseCoeff) \times targetGain$
13:        end if
14:        Apply gain to input sample: $outputBuffer[i] = inputBuffer[i] \times currentGain$
15:    end for
16: end procedure
```

B. Testing and Analysis

To ensure our audio processing algorithms are robust, we will use benchmarking and profiling tools (such as Valgrind, gprof, or perf) to measure execution time, thread efficiency, and memory usage.

We will test for:

1) **Latency:** Measuring the delay between input and output processing.
2) **Thread Scalability:** Observing performance on single-core vs. multi-core execution.
3) **Audio Quality:** Ensuring no unintended distortions or artifacts are introduced.
4) **System Load:** Monitoring CPU and memory usage to prevent overloading.

Our testing will involve real-world audio inputs, such as human speech and music, to evaluate the effectiveness of each effect. Additionally, we will compare our implementation's performance to existing real-time audio processing software.

III. Background

Audio processing has significantly evolved from its analog roots to more sophisticated digital methods [6]. Initially, it relied on physical devices ranging from vacuum tubes to transistors. These devices would directly manipulate electrical audio signals and modify their amplitude, frequency, and other properties to create specific effects. While these methods were effective, they were limited in various ways, such as noise, size constraints, and component degradation over time.

The leap from analog to digital audio processing signaled a major achievement for this technology [7]. Digital audio processing enables more precise, flexible, and reliable manipulation of audio signals through a method called Digital Signal Processing (DSP). DSP involves converting analog audio signals into digital format, represented by numerical values that can be algorithmically manipulated to achieve precise transformations. Digital processing allows for easy replication, scalability, and significantly reduced noise compared to analog methods [8].

Real-time audio processing presents tough and specific challenges, one of the most notable being latency management and computation load. Latency, the delay between raw audio input and processed output, must be minimized to maintain audio fidelity and synchronization. Imagine trying to have a conversation with a friend or colleague, only to have them hear you seconds after you speak, and then waiting seconds to hear their response. This challenge is particularly critical in live applications such as broadcasting, live performances, or conferencing, where latency can introduce noticeable disruptions and degrade the user experience.

Modern processing units typically contain multicore architectures, which have helped facilitate new methods of handling computation demands and overcoming challenges through parallelism. Multithreading, a parallel computing technique where multiple tasks are executed concurrently, significantly improves the efficiency and responsiveness of audio processing applications. By distributing computational tasks such as filtering, equalizing, and compressing audio across multiple threads, real-time audio processors can simultaneously manage numerous operations, drastically reducing latency and optimizing resource usage.

Incorporating advanced data structures, such as lock-free queues, can further enhance multithreaded audio processing by eliminating bottlenecks associated with traditional thread synchronization methods. These lock-free approaches minimize delays caused by waiting for locks, supporting a smoother and more efficient flow of audio data between processing threads.

Overall, the continued advancement in digital signal processing, multithreading techniques, lock-free data structures, and hardware developments forms a robust foundation. This allows us to build an efficient, low-latency, real-time audio processor capable of sophisticated and high-quality audio modifications suitable for diverse practical applications.

IV. Approach

Our approach to developing a real-time audio processor prioritizes efficiency, low latency, and modularity. We rely on open-source tools and libraries while using consumer-grade hardware for implementation and testing.

We use C++ as the primary programming language due to its performance advantages and suitability for real-time applications. For handling audio input and output (I/O), we utilize the RtAudio library, which provides cross-platform compatibility and direct access to audio hardware interfaces.

Our threading model implements a dedicated audio processing thread separated from I/O operations using the C++ standard library's `std::thread`. This architecture aims to provide an optimal balance between processing efficiency and low-latency performance. Synchronization between threads is managed through our `BufferQueue` implementation, which uses mutexes and condition variables (`std::mutex` and `std::condition_variable`) to ensure thread-safe data transfer while minimizing contention.

To manage real-time constraints effectively, we prioritize efficient buffer handling within an optimized processing pipeline. Our system processes incoming audio in fixed-size frames. Each frame undergoes a sequential chain of effects—noise gating, equalization, de-essing, and limiting—before being queued for output.

Effect Processing Pipeline

Our system implements several real-time audio effects designed to enhance microphone input. These effects are applied sequentially in a carefully designed processing chain to maximize audio quality and ensure predictable behavior.

The Noise Gate uses spectral analysis via the Fast Fourier Transform (FFT) to identify and reduce unwanted background noise. It attenuates frequency components below a set threshold. This implementation includes attack and release smoothing parameters to prevent audible artifacts during transitions between gated and non-gated states.

The 3-Band Equalizer (EQ) employs overlap-add processing with Hann windowing to divide the audio spectrum into three adjustable frequency bands—low, mid, and high. Our implementation includes smooth transitions between bands to prevent frequency ”shelving” artifacts. This technique allows precise frequency manipulation while maintaining phase coherence across the spectrum.

Our De-esser implementation targets the typical sibilance range (approximately 4 kHz – 10 kHz). Using FFT-based spectral processing, it selectively reduces the amplitude of these frequencies only when they exceed a defined threshold, resulting in smoother speech audio without significantly affecting overall clarity or brightness.

The Limiter serves as the final stage in our processing chain, preventing signal clipping by dynamically controlling gain. It employs carefully tuned attack and release coefficients to respond quickly to transients while aiming for transparent operation on sustained audio content, maximizing loudness without introducing distortion.

Each effect is designed as a modular component that can potentially be enabled or disabled individually. By organizing these effects in a sequential pipeline executed on a dedicated processing thread, we achieve both processing efficiency and high audio quality.

Hardware Setup

The hardware requirements for running and testing the processor are minimal and consist of standard consumer-grade components:

- A standard consumer-grade microphone for audio input.
- A computer with a multi-core processor (e.g., an Intel i5/i7 or AMD Ryzen 5/7 or equivalent) capable of handling real-time processing demands.
- A standard pair of speakers or headphones for monitoring the audio output.
- Optional: Different USB audio interfaces can be used to test variations in audio input quality and driver performance.

Multithreading and Parallel Processing Strategy

To optimize performance while maintaining audio integrity, we implement a strategic multi-threaded approach:

- **Audio Callback Thread:** Managed by the RtAudio library, this high-priority thread handles real-time audio capture from the microphone and playback to the output device [9]. It interacts directly with the hardware audio buffers and manages strict timing constraints imposed by the audio driver.
- **Effects Processing Thread:** A dedicated thread retrieves audio data from an input buffer, processes it through the complete effects chain (noise gating, EQ, de-essing, limiting), and places the processed data into an output buffer. This thread sequentially applies effects in a carefully determined order.
- **Thread Communication:** We implement thread-safe `BufferQueue` objects utilizing mutexes and condition variables for inter-thread communication. This mechanism allows efficient and safe passing of audio data buffers between the audio callback thread and the effects processing thread, preventing race conditions and minimizing the risk of buffer underruns or overruns.

This architecture provides an optimal balance between leveraging parallel processing benefits and maintaining the integrity and sequential nature of the audio signal chain. By separating time-critical I/O operations from potentially CPU-intensive processing, we reduce the risk of audio dropouts (glitches) while ensuring all effects are applied consistently to the audio stream.

V. Results

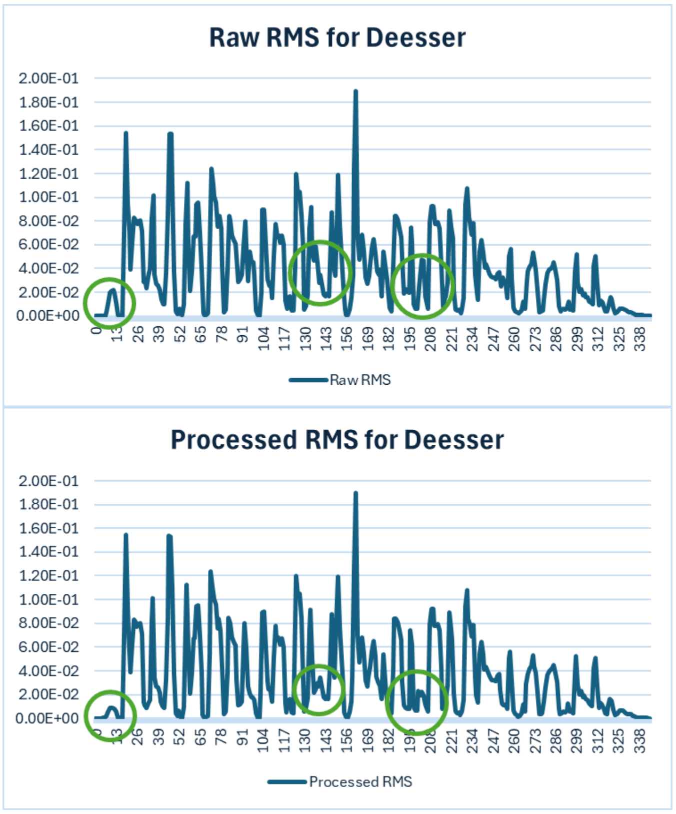

Fig. 1. Raw vs. processed RMS for de-esser sample audio.

The charts above compare the root mean square (RMS) values of audio frames before and after de-essing. Overall, the processed audio exhibits a consistent reduction in RMS energy in frames that likely contain strong sibilant content. Three key regions have been highlighted to illustrate this change more clearly.

In the early frames (e.g., frames 0–30), a noticeable drop in RMS amplitude can be observed in the processed output, reflecting the suppression of harsh high-frequency transients, which are typical indicators of sibilant sounds such as ”s” and ”sh.” The mid-range frames (e.g., 130–160) show a similar pattern: energy that peaks in the raw RMS data has been attenuated in the processed version. This suggests that the de-esser is successfully identifying and reducing frequency components in the 4–10 kHz range, where sibilance is most prominent. Likewise, in the later frames (around 190–210), the processed RMS values are consistently lower than their raw counterparts, indicating continued effective suppression. Importantly, the overall shape of the RMS curve remains preserved, which implies that the de-esser is targeting only the sibilant frequencies without introducing distortion or disrupting the natural dynamics of the audio.

These results demonstrate that the de-esser is functioning as intended — selectively reducing excessive high-frequency energy to produce a smoother, less harsh output while maintaining the integrity of the original audio content.

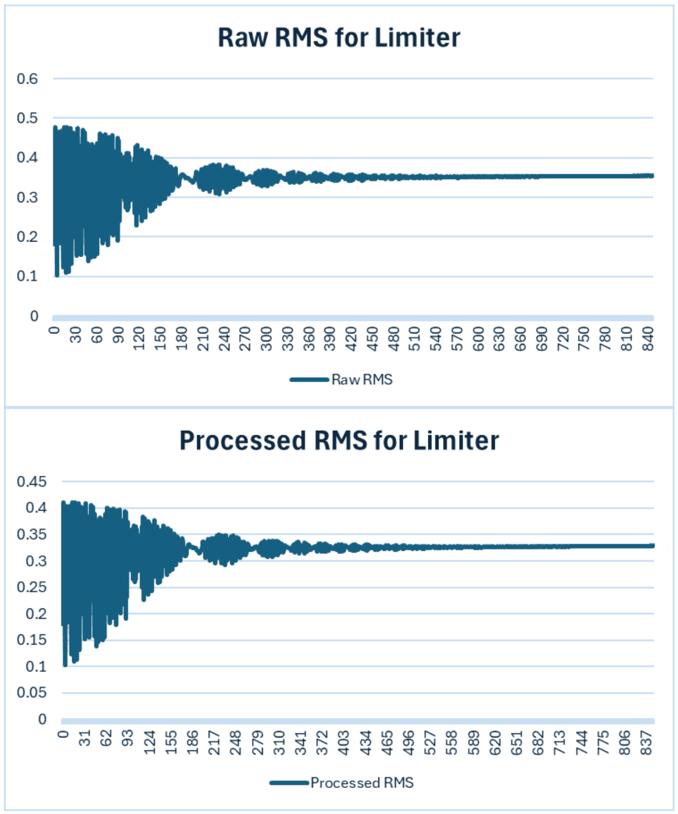

Fig. 2. Raw vs. processed RMS for limiter sample audio.

The RMS graphs for the limiter clearly illustrate the impact of dynamic range compression, particularly in the early frames where the audio signal is at its peak intensity. In the raw RMS plot, the signal initially fluctuates around 0.4 to 0.5 before naturally tapering off. This indicates the presence of high-amplitude content — likely transient peaks or loud passages — in the unprocessed audio.

In contrast, the processed RMS plot reveals a more tightly controlled signal, especially within the first 150 frames. The limiter effectively suppresses peaks that would otherwise exceed a defined threshold, producing a flattened upper envelope of RMS values around 0.35–0.4. The differences are most pronounced in the earliest frames, where amplitude limiting has reduced the RMS level relative to the raw input. However, the limiter retains the shape and flow of the original signal in quieter sections, as shown by the overlapping curves in later frames (after frame 300), where both raw and processed RMS values converge.

This behavior confirms the limiter is acting as expected: attenuating louder sections without affecting the lower-amplitude content, thereby reducing dynamic range while maintaining the overall temporal structure of the signal. The output is more uniform and consistent in loudness, which is desirable in many production contexts such as podcasting, broadcasting, or music mastering.

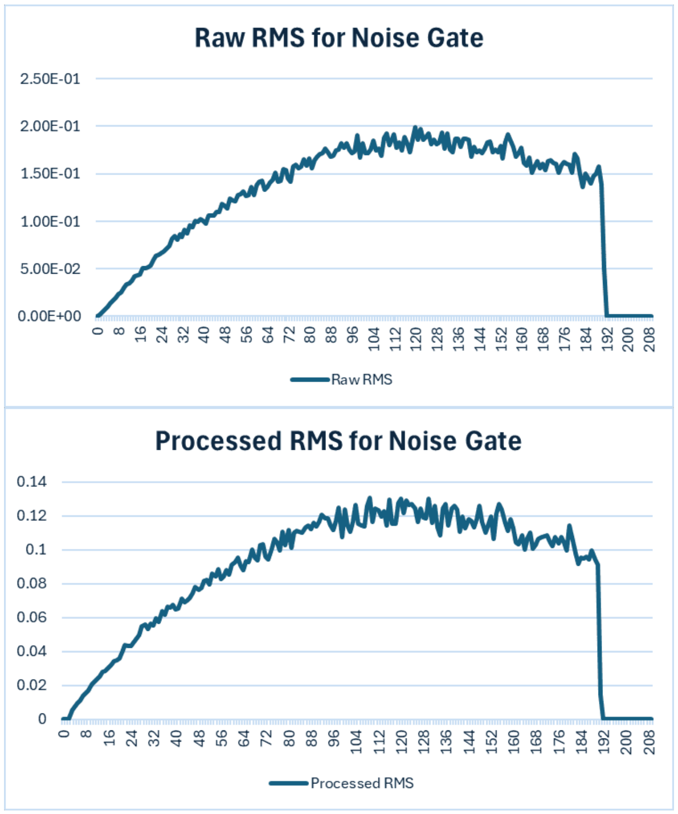

Fig. 3. Raw vs. processed RMS for noise gate audio.

The charts above compare the RMS values of the raw and noise-gated audio signal. A noise gate is designed to suppress low-level background noise by attenuating signals that fall below a specified threshold. The effect of this gating behavior is clearly visible in the processed RMS curve, particularly during the initial and final portions of the signal.

In the raw RMS plot, the energy builds smoothly from silence to a peak and then tapers off, with minor fluctuations reflecting the natural variation in the signal's dynamics. The processed RMS plot, however, shows that the noise gate delays the onset of energy slightly and reduces the amplitude across most of the signal. This reflects the gate's threshold-based behavior — weaker input frames are attenuated or muted, especially early on where the RMS was previously below 0.02.

Furthermore, during the decay phase, the processed signal cuts off more sharply, indicating that the gate is closing aggressively as the signal falls below the threshold. Compared to the raw output, the noise-gated version maintains a more constrained and cleaner signal, especially in quieter regions.

These results demonstrate that the noise gate is functioning effectively by suppressing low-level content and emphasizing stronger signal components, which is especially useful in reducing background noise or hiss during silent pauses in dialogue or recordings.

The equalizer functionality was evaluated by applying a 2× gain to each of the three primary frequency bands—bass, middle, and treble—individually, using an instrumental audio sample. The raw and processed root mean square (RMS) values were plotted frame-by-frame to assess how the gain adjustments influenced overall energy distribution in the signal. The instrumental nature of the sample provided a clean frequency spread, allowing the effects of band-specific amplification to be clearly observed without interference from vocals.

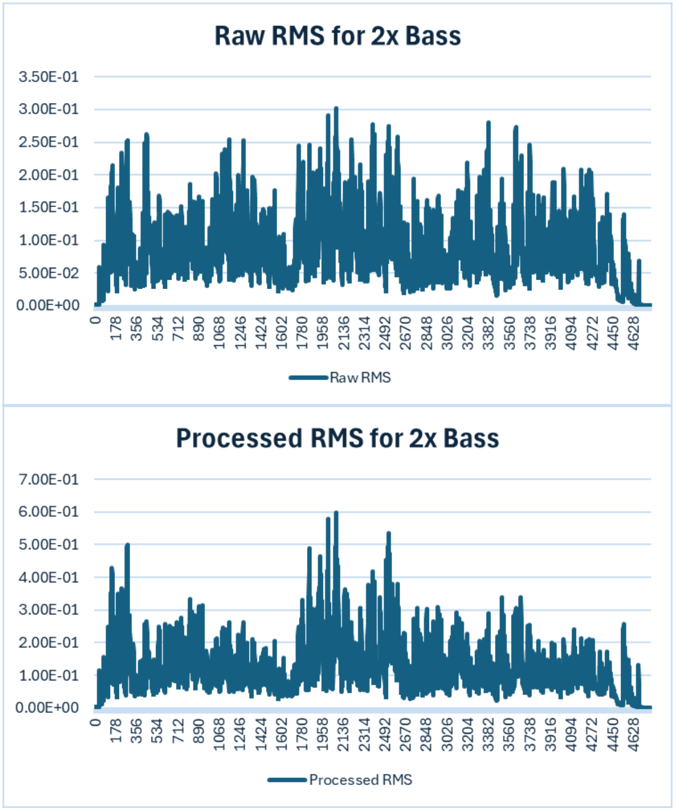

Fig. 4. Raw vs. processed RMS for doubled bass frequencies.

In the bass-boosted version, the processed RMS values exhibited a significant increase in low-frequency-dominant sections, confirming that the equalizer effectively enhanced the lower register of the track. This boost amplified the rhythmic weight and depth of the signal, particularly in frames featuring kick drums or bass lines. The resulting effect would translate perceptually to a warmer and more grounded audio profile.

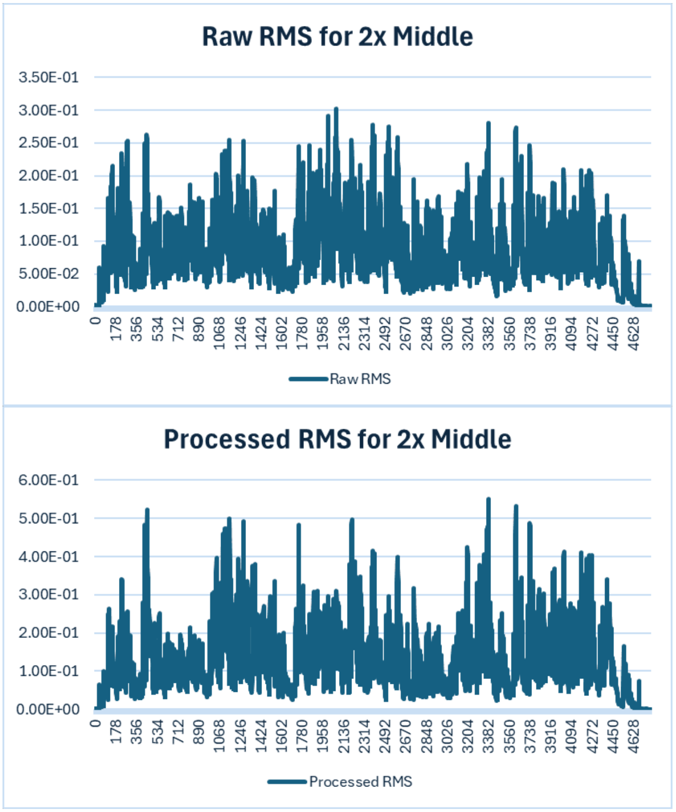

Fig. 5. Raw vs. processed RMS for doubled bass frequencies.

The midrange boost had a more consistent impact across the entire track. As mid frequencies in instrumental music often carry melodic content—such as guitars, pianos, and harmonic overtones—the RMS increase was distributed evenly, reflecting a stronger and more present core. The processed signal showed up to a 2× increase in RMS in musically active frames, indicating that the mid-band gain successfully elevated the body and tonal richness of the composition.

With the treble boost, the processed RMS plot revealed modest but targeted increases in high-frequency content. The changes were most pronounced in sections likely containing cymbals, string harmonics, or transient high-end detail. Although the RMS increase was not as large as in the bass or mid bands, it was still sufficient to enhance brightness and articulation. The treble gain contributed clarity and definition without overwhelming the overall mix.

Across all cases, the equalizer preserved the natural dynamics and temporal shape of the audio while selectively amplifying the designated frequency bands. The resulting RMS changes confirm that the processor is capable of making effective tonal adjustments in a musically transparent way—boosting desired elements without introducing artifacts or distortion. The instrumental sample proved especially useful in isolating the frequency band effects and validating the equalizer's smooth transition zones and gain precision.

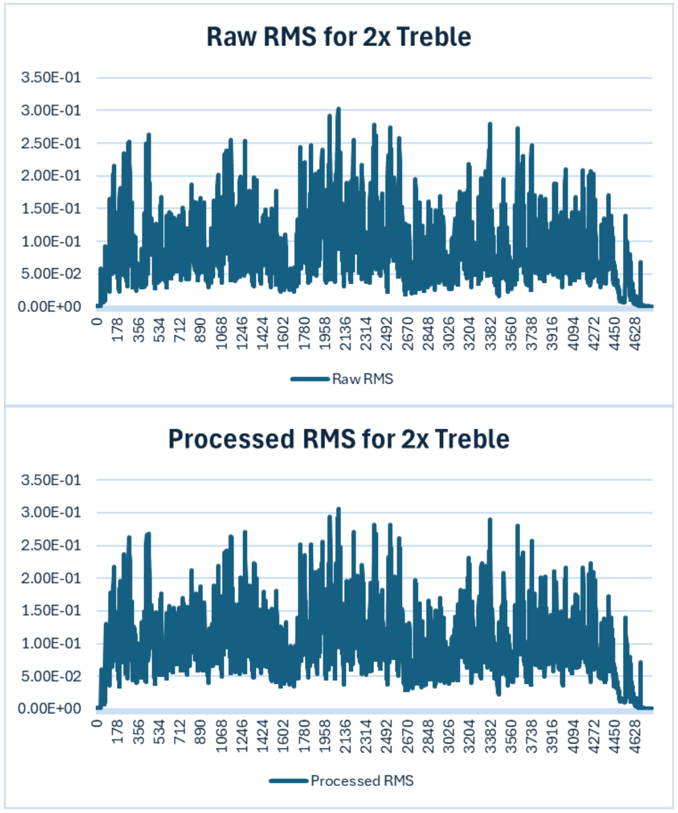

Fig. 6. Raw vs. processed RMS for doubled treble frequencies.

VI. Conclusion

This real-time audio processing implementation demonstrates an effective and efficient approach to enhancing audio signals through multiple effects. By implementing a sequential chain of a noise gate, equalizer, de-esser, and limiter, the system selectively attenuates unwanted noise, shapes frequency content, reduces sibilance, and prevents clipping, while preserving the clarity and tonal characteristics of the original signal. The use of the Fast Fourier Transform (FFT) enables precise manipulation in the frequency domain, and the inverse FFT ensures that the resulting time-domain signal remains phase-accurate and natural sounding [10].

The modular structure of the processing pipeline, with dedicated stages for noise gating, equalization, de-essing, and limiting, allows for isolated tuning and facilitates real-time interactivity. We initially experimented with assigning each effect to its own processing thread, coordinated by a dedicated mixing thread responsible for combining audio and writing to the output buffer. However, this approach introduced unnecessary complexity and ultimately increased audio latency. Further research into digital signal processing principles revealed that the sequential order of applying effects is integral to both functionality and quality. The chosen order maximizes performance; for example, applying the noise gate first ensures subsequent effects do not process audio components that are ultimately removed.

Ultimately, the most efficient parallel processing strategy involves one dedicated thread for all effects processing, separate from the threads handling audio input and output. The integration of thread-safe buffer queues utilizing condition variables enables continuous, low-latency operation, making the system robust for demanding real-time applications. Additionally, the graphical user interface provides intuitive control over each effect's parameters, enhancing user engagement and making the system adaptable for different voice types and specific use cases.

This approach offers a more transparent and targeted alternative to traditional static processing methods. It preserves transient information effectively and avoids undesirable side effects such as over-compression or unnatural tone shaping. Its performance characteristics and responsiveness further highlight its applicability in live audio workflows, including broadcasting, streaming, and recording scenarios. The current design lays a solid foundation for future development, such as incorporating adaptive thresholding, exploring multi-band processing techniques, or implementing expanded user interface enhancements.

References

[1] M. Geier, T. Hohn, and S. Spors, "An open-source C++ framework for multithreaded realtime multichannel audio applications," in *Proc. Linux Audio Conf. (LAC)*, Stanford, CA, 2012. [Online]. Available: http://lac.linuxaudio.org/2012/papers/19.pdf

[2] X. Sang and X. Chen, "Design, simulation and measurement of split-band digital audio expander and noise-gate," in *Proc. 7th Int. Conf. Information, Communications and Signal Processing (ICICS)*, Dec. 2009. [Online]. Available: https://ieeexplore.ieee.org/document/5397498

[3] V. Välimäki and J. D. Reiss, "All About Audio Equalization: Solutions and Frontiers," *Applied Sciences*, vol. 6, no. 5, art. 129, 2016. [Online]. Available: https://doi.org/10.3390/app6050129

[4] K. Linhard, P. Bulling, and A. Wolf, "Frequency Domain De-essing for Hands-free Applications," in *Proc. DAGA 2017 (43rd German Conf. Acoustics)*, Kiel, Germany, 2017, pp. 315–318. [Online]. Available: https://pub.dega-akustik.de/DAGA$_2017/data/articles/000071.pdf$

[5] D. Giannoulis, M. Massberg, and J. D. Reiss, "Digital Dynamic Range Compressor Design—A Tutorial and Analysis," *J. Audio Eng. Soc.*, vol. 60, no. 6, pp. 399–408, 2012. [Online]. Available: http://eecs.qmul.ac.uk/ josh/documents/2012/GiannoulisMassbergReiss-dynamicrangecompression-JAES2012.pdf

[6] H. Bode, "History of Electronic Sound Modification," *J. Audio Eng. Soc.*, vol. 32, no. 10, pp. 730–739, 1984. [Online]. Available: http://www.vasulka.org/archive/Artists1/Bode%2CHarald/History.pdf

[7] J. A. Moorer, "Audio in the New Millennium," *J. Audio Eng. Soc.*, vol. 48, no. 5, pp. 490–498, 2000. [Online]. Available: https://secure.aes.org/forum/pubs/journal/?elib=12062

[8] T. Wilmering *et al.*, "A History of Audio Effects," *Applied Sciences*, vol. 10, no. 3, art. 791, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/3/791

[9] G. P. Scavone, "RtAudio: A Cross-Platform C++ Class for Realtime Audio Input/Output," in *Proc. Int. Computer Music Conf. (ICMC)*, Gothenburg, Sweden, 2002. [Online]. Available: https://quod.lib.umich.edu/i/icmc/bbp2372.2002.041

[10] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965. [Online]. Available: https://www.cs.jhu.edu/ misha/ReadingSeminar/Papers/Cooley65.pdf