Enhancing Media Playback Security without Sacrificing Performance

A basic audio codec implemented in Rust

JACK O'REILLY, Stanford University, USA

Many problem domains require software solutions that have specific constraints on performance to be useful. One particular subset of these problems is the set of problems in the real-time domain, which require computation to complete within a timeframe associated with some physical world frame of reference. For example, systems in the domain of software radio, transportation control, and media encoding and playback all must be written as to make decisions within a hard time limit.

The constraints in the latter domain – media playback – are obvious. If software is unable to render media content as a raw signal in less time than the signal's duration, the software is effectively useless. This constitutes one of the reasons that media codec technologies have historically been implemented using low level languages like C and C++. In the modern era of software development, Rust purports to be a tool that can fulfill real-time requirements while allowing for reduced risk of software instability or insecurity.

In this paper, I hope to understand the development complexity of writing a media codec application in Rust. To do so, I describe the process of porting an existing audio codec (written for Stanford's MUSIC 422 course, "Perceptual Audio Coding") to a Rust implementation of same. Factors under consideration include ease of implementation, maturity of tooling and ecosystem, performance, and ease of validation.

CCS Concepts: • **Software and its engineering** → Imperative languages;

Additional Key Words and Phrases: Rust, audio codec, media playback, numpy, memory safety

ACM Reference Format:
Jack O'Reilly. 2017. Enhancing Media Playback Security without Sacrificing Performance: A basic audio codec implemented in Rust. 1, 1 (December 2017), 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Media transport and playback applications pose an interesting challenge with respect to software design. The performance requirements for software packages like media codecs in particular have historically precluded the transitions to managed and memory-safe languages weâĂŹve seen with some other application types. Real-time constraints on encode and decode processes have meant that many widely used codecs are implemented in C or C++, with all the power and potential for error that accompanies them.

Adding to the complexity of this domain is the fact that compressed elementary media bitstreams generally contain in-band sizing and control data. This combined with the fact that the input data may come from an untrusted source has meant that media playback applications have historically been a common target for security vulnerabilities. Rust provides a unique combination of performance and safety to address these domain constraints.

1.1 Context: the MUSIC 422 audio codec

The Rust audio codec implemented in this project is based on a Python implementation called the ACHE audio codec which I implemented for Stanford's MUSIC 422 course in collaboration with project partner Nitish Padmanaban. This codec uses a lossy compression algorithm based on a psychoacoustic model, allowing us to identify frequency elements which can be removed from the original signal with minimal perceptual difference in

Author's address: Jack O'Reilly, Stanford University, 450 Serra Mall, Stanford, CA, 94305, USA, oreilly@stanford.edu.

the resulting signal after the encode / decode process. The methods for signal windowing [1], analysis, masking, and quantization used in the codec are drawn from [2]. The reference audio codec is implemented in Python, using NumPy for numerical computations.

1.2 High level encoder architecture

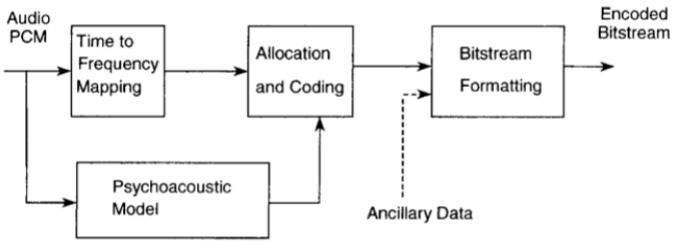

Fig. 1. Basic encoder architecture [2]

The steps in the procedure (see Figure 1 for converting a set of raw multi-channel PCM audio data time samples to a compressed encoded representation are as follows:

For each audio channel:

(1) Compute the modified discrete cosine transform (MDCT) of the time samples
(2) Compute absolute sound power levels across the signal frequency spectrum
(3) Compute the overall signal masking curve
  (a) Identify masking sources by searching for signal peaks
  (b) Add the raw intensities[3] from contributing masking sources according to psychoacoustic model
  (c) Add the 'threshold in quiet' [2] base masking curve
(4) Section the frequency data into a subset of Zwicker critical subbands [4]
(5) Compute a bit allocation per critical band based on the difference between max signal power in the band and the overall masking curve
(6) Write encoded block-level metadata to bit array
(7) For each critical band
  (a) Perform block floating point quantization on MDCT mantissas using bit allocation assignment
  (b) Write encoded mantissas to bit array

From a computation perspective, one of the most significant differences from the decoding process is that the encoder must effectively perform a search per audio block to determine the optimal subband bit allocation in order to optimize perceptual quality. Besides this step and the computation of the masking curve, the decode process is effectively the reverse of the steps listed above.

1.3 High level decoder architecture

The decode process steps (see 2, which convert the compressed encoded representation back to raw PCM audio data, are as follows:

For each audio channel (channel count read from file-level metadata):

(1) Read critical band count and other block-level metadata from bitstream
(2) For each subband:
  (a) Perform block floating point dequantization using bit allocation metadata

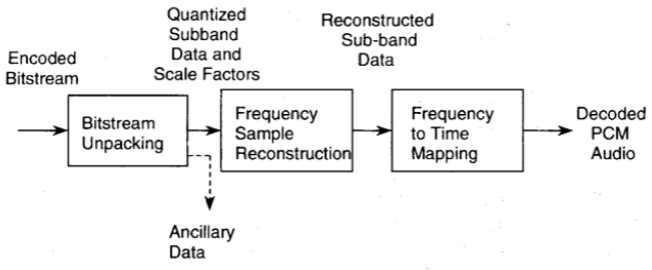

Fig. 2. Basic decoder architecture Bosi and Goldberg [2]

Module	Purpose	Function symbols exported
bitalloc	Algorithms for calculating optimal bit allocation	1
bitpack	Subroutines for efficiently packing binary data at the bit level	N/A: no FFI testing
codec	File-format agnostic encoding and decoding logic	2
lib	Top-level public codec library interface	N/A: no FFI testing
mdct	Implements the Modified Discrete Cosine Transform algorithm	2
pacfile	Routines for reading and writing encoded data files	N/A: no FFI testing
psychoac	Algorithms relating to the codec psychoacoustic model	17
quantize	Algorithms for performing quantization of sample types	7
wavfile	Routines for reading and writing WAV data files	N/A: no FFI testing
window	Routines for windowing sample blocks for analysis	3

Table 1. Code modules and associated purposes

(3) Overlap and add time samples from the previous decoded block in order to reconstitute original signal

2 APPROACH

2.1 Implementation

Codecs are not uncomplicated pieces of technology. In many cases, complex mathematical operations as well as low-level bit twiddling is necessary to achieve the goals of the software. Such methods are error-prone, and given my relative inexperience with Rust's rules for manipulation of primitive types and numerical operations, I chose to take a test-driven approach for implementing the codec.

2.1.1 Project organization. I chose to separate the project into three main sections. They are:

(1) The reference Python implementation, including:
- reference algorithm implementations
- automated test driver code for comparing reference to Rust via FFI
- Python-side benchmarking code

(2) Python-to-Rust FFI code
(3) Rust library code, including:
- unsafe FFI layer coercing raw C types to slices
- safe algorithm layer operating on slices of primitives

Source file	Purpose
python/quantize.py	Reference Python implementation, parameterized unit test driver calling Python FFI module and comparing results
ffi/quantize_ffi.py	Python FFI module for invoking exported Rust lib functions
ache_codec/src/quantize_ffi.rs	FFI Rust module exporting function symbols; unsafe coercion to slices
ache_codec/src/quantize.rs	Algorithm implementation operating on slices; no unsafe code

Table 2. Source code file responsibilities for a particular processing module

2.1.2 Differences from reference implementation. From the onset of implementation, there were some design decisions made for the Rust implementation that caused differences in implementation details from the Python reference. Probably the most significant was heavy use of pre-allocated buffers for numerical operations within the Rust implementation. An audio codec which operates on fixed size frames can typically be designed to use a known maximum amount of scratch buffer space within the context of a frame, and this codec is no exception. Therefore, to improve performance, the processing functions in the Rust back end are designed to work on pre-allocated output arrays rather than returning references to new memory. The fine-grained control over memory allocation and management possible through Rust affords significant opportunity for improved performance; this is reflected in the benchmark results.

Static typing for primitives: much of the computation done on the NumPy side resolves to appropriate types dynamically. On the Rust side, it was necessary to determine static types for each point along the processing chain.

2.2 Testing

As noted above, it was essential that I be able to compare computation results between the Rust implementation and the reference Python implementation directly. To do so, I considered creating a serialized format for comparing test output in a file-based manner. However, in order to more easily reuse testing code for benchmarking and to reduce possibility of error, I decided to pass computation results back to the Python context through Rust's FFI.

2.2.1 Testing to reference implementation with the Rust FFI. To form the codec as a whole, it was necessary to implement various processing modules (see Table 1). In order to allow automated validation, each processing module consisted of multiple source files handling various interface layers (see Table 2 for the source files relating to the quantization module, e.g.). For each module in the original Python implementation, I created a separate interface implementation which resolved to the Rust back end through FFI. In the Rust library, each module contained a base layer of safe code and an FFI-only layer (not used in the final Rust-only executable) to encapsulate unsafe conversions and facilitate testing.

Python was particularly appropriate for the test driver end for a few reasons. First, duck typing made it very straightforward to parameterize the existing automated test infrastructure so that the Rust back end results would also be compared to the reference. Additionally, it appears to be significantly simpler to call into Rust from Python than vice versa, as the former has stronger guarantees about memory layouts for primitives and vector data. Output vector parameters were passed to the Rust back end as blank mutable buffers of appropriate size and were written to without allocation. Besides the Rust-backed implementation of psychoac::ScaleFactorBands, all

memory allocation for validation testing was managed on the Python side and was mutated within Rust without ownership transfer.

2.2.2 Testing using Quickcheck for Rust. Some modules, such as the bitpack module, were implemented in a heavily object-oriented fashion on the Python side. For these modules, in order to reduce FFI boilerplate and allow for easier random input generation, I used Rust's quickcheck crate. QuickCheck is a powerful automated testing tool that was immensely helpful for ferreting out implementation bugs in this module.

One item I found somewhat difficult in Rust was implementing a QuickCheck shrinker for vector-based data input. In dynamic languages, it's easier to do so since it's not necessary to track ownership (or disposal) of previously used test-input variations.

2.3 Benchmarking

Each processing module was supplemented with a simple benchmarking test suite. Most operations are quick enough in single iterations that my intuition had me worried about timing precision, so the benchmarks first generate a configurable number of iterations of random input, then perform the operation in question for that many iterations, taking care not to include the input generation as part of the timing measurement. As noted previously, due to enhanced capabilities with Rust to minimize allocations in a well-designed codec, the benchmarks can reflect the benefits of a difference in allocation strategy.

Both Python and Rust provided straightforward methods for benchmarking code; the former through integration with the unittest API and the latter through the 'benchmark tests' functionality provided by default in Cargo-managed projects.

3 CONCLUSIONS

3.1 Rust development effort

Rust and the associated tooling provided some significant benefits for implementing real-time software. In addition to having very useful standardized testing and benchmarking frameworks (almost certainly adding to the average quality of available crates), it also made project build configuration a breeze. My significant experience with C++ made the functionality provided by Cargo for linking and generating binaries seem space-age by comparison.

There were a few drawbacks that I was mostly able to work around. One notable language limitation is the lack of variable-length stack-allocated arrays, which are sometimes used in native code for allocating high-performance scoped scratch space without touching the stack. It was unclear to me whether the Rust compiler would be smart enough to optimize a function-local Vec<T> into an equivalent operation, or if doing so is simply inherently unsafe due to possibility of exceeding stack size limits. Additionally, it would have been nice to have slice-based 'broadcasting' for arithmetic vector operation as is possible with languages like Matlab, NumPy, etc., although it may be that I simply didn't find a crate that was available for that purpose.

3.2 Benchmarking

The benchmarking results measured for various processing modules can be found in Table 3. The Rust implementation outperforms the Python + NumPy implementation for all measured functions, although the gain is modest in some cases. It is notable that this performance is achieved in Rust without the use of any SIMD operations – all vector operations are achieved through naive linear iterator manipulation.

Module, function	Iterations	Python time (sec)	Rust time (sec)
bitalloc, BitAllocOptimal	1000	0.11663	0.00183
mdct, MDCT	10000	1.09096	0.53801
psychoac, CalcSMRs	1000	4.50631	1.42410
quantize, vQuantize	10000	0.62038	0.07915
window, SineWindow	100000	1.3727	1.31334
window, SineWindow (Rust pre-alloc)	100000	1.3727	0.5471

Table 3. Benchmarking results

3.3 Possible improvements

There are a few items that could be improved in the Rust codec implementation. It would be possible to eke out additional performance gains by pre-computing window coefficients at file init time rather than per windowing operation; likewise pre-computations for FFT operations. Currently in both the Python and Rust implementations, these operations are performed more often than necessary due to unnecessarily limited scoping of the associated data.

It would also be interesting to investigate the possibility of templating the processing operations in the Rust back end as much as possible. It would significantly complicate the FFI validation code, but could provide a more flexible and/or scalable experience for future codec feature additions.

Finally, I undertook no effort to implement vectorized / SIMD operations in the data processing for the Rust codec. Basic research shows that said functionality is available, which could result in significant performance gains on supported platforms. This is notable, especially considering that NumPy already makes use of these operations internally.

4 REFERENCES

4.1 Software dependencies

- num = "0.1.41"
- num-complex = "0.1.41"
- rand = "0.3.18"
- time = "0.1.38"
- quickcheck = "0.5.0"
- rustfft = "2.0"
- byteorder = "1.2.1"
- clap = "2.29.0"

4.2 Additional reading

- Block floating point quantization: http://oldweb.mit.bme.hu/books/quantization/floating-point.pdf
- Quickcheck – originally implemented for Haskell: https://hackage.haskell.org/package/QuickCheck
- Quickcheck "Shrink" and "Arbitrary": https://stackoverflow.com/questions/16968549/what-is-a-shrink-with-regard-to-haskells-quickcheck

REFERENCES

[1] Marina Bosi and Grant Davidson. 1992. High-quality, low-rate audio transform coding for transmission and multimedia applications. In *Audio Engineering Society Convention 93*. Audio Engineering Society.

[2] Marina Bosi and Richard E. Goldberg. 2002. *Introduction to Digital Audio Coding and Standards.* Kluwer Academic Publishers, Norwell, MA, USA.

[3] Robbert G Van Der Waal and Raymond NJ Veldhuis. 1991. Subband coding of stereophonic digital audio signals. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on.* IEEE, 3601–3604.

[4] E. Zwicker. 1961. Subdivision of the Audible Frequency Range into Critical Bands (Frequenzgruppen). *The Journal of the Acoustical Society of America* 33, 2 (1961), 248–248. https://doi.org/10.1121/1.1908630