A Rust Implementation of the Web Audio API

Otto Rottier, Benjamin Matuszewski

A Rust Implementation of the Web Audio API

Decoupling the Web Audio API from the web

Otto Rottier
Utrecht, The Netherlands
ottorottier@gmail.com

Benjamin Matuszewski
STMS Ircam-CNRS-Sorbonne Université
Paris, France
benjamin.matuszewski@ircam.fr

ABSTRACT

In this paper we present a novel implementation of the Web Audio API specification written in Rust. While still in its early stage, the library already proposes a stabilized API and an important subset of the specification. We think this novel implementation has the potential to fill two complementary gaps. From a Web Audio API perspective, it proposes to decouple the API from the web (and web browsers), potentially opening new application areas and helping to widen its community of users. From a Rust perspective, it could provide an intermediary and extensible solution for audio applications that is not yet available in the ecosystem. This paper describes the general design of the library, expliciting and justifying the trade-offs that have been made to the specification in regards to the specificities of the Rust language. This general picture is completed with some examples of usage of the library and a discussion on its current performance. Additionally, a related JavaScript package that proposes Node.js bindings to the core Rust library is introduced. The project is open-source and released under the MIT License.

CCS Concepts

•**Applied computing** → **Sound and music computing;** •**Software and its engineering** → *Software libraries and repositories;* •**Information systems** → World Wide Web;

Keywords

Web Audio API, Rust

1. INTRODUCTION

In the last decade, the Web Audio API first proposed by Chris Rogers in 2011 and released as a W3C Recommendation in 2021 [12], has gained increasing attention by developers, artists and researchers. Indeed the possibility of doing advanced audio processing and synthesis natively on the Web platform has unfolded a number of novel possibilities in different domains such as music performance and creation, gaming or online conferencing. As of today, the Web Audio API appears to have reached a point where its growing community of users, number of existing applications and amount of documentation and tutorials makes it an interesting alternative for someone willing to develop an audio application. Despite this interest and the large application domains, we believe adoption of the Web Audio API is limited by the sandboxed and constrained environments that are Web browsers. With its stable specification and comprehensive documentation, the Web Audio API could offer an interesting alternative for native applications. In this paper we present `web-audio-api-rs`[1], a novel implementation of the Web Audio API written in the Rust language [20] that we think could fill this gap.

The choice of the Rust language has been motivated by several reasons. Primarily, Rust is a low-level, memory-efficient language much like C++. There is no runtime and no garbage collector which means we can build real-time audio components with predictive timing and high performance. Unlike higher level languages such as Python or JavaScript, Rust allows us to use memory efficient primitives (float, double, atomic, reference-counted etc), to choose between stack and heap allocation and to define custom memory allocators. Compared to C++, Rust introduces the concept of data ownership which prevents the entire class of race conditions and concurrency issues that may arise in multi-threaded execution. This gives us two advantages when implementing audio systems: we can eliminate some needless copies of audio buffers and it gives us a compile-time guarantee that the concurrent execution of the control thread and render thread is free of data races. Moreover, the Rust ecosystem (compiler toolchain, dependency management, error messages, auto-generated documentation) is modern, easy-to-use and beginner-friendly, potentially opening the domain of low level audio processing to developers that are primarily invested in web programming [5]. As such, our project also welcomes contributors that are well versed in using the Web Audio API and want to get their hands dirty in low level audio processing.

Beyond the personal learning experience of implementing such an API from scratch, our objectives for the library are twofold. First, we aim for full compliance with the specification and, for deviating from it only in specific, justified and therefore predictable cases. Second, and deriving from this first point, our goal is to provide an API that is both easy to use when coming from a JavaScript background with only a few adaptations to the Rust coding style and specificities, and inversely, can leverage on existing JavaScript documentation and tutorials for Rust users. As such, the library could open interesting perspectives in several application domains. In artistic contexts, it could open novel possibilities in several areas such as distributed music systems or digital musical instruments (DMI). For start-ups and industry, it

Web Audio Conference WAC-2022, July 6–8, 2022, Cannes, France.

[1] https://github.com/orottier/web-audio-api-rs

could provide a novel tool to support the path from prototyping to production (e.g. simple prototyping in JavaScript easily ported to Rust for performance) in many domains such as game development or embedded devices creation.

After a presentation of the related works (Section 2), we will present in Section 3 the main design and implementation aspects of the library. This general presentation will be followed in Section 4 by a few examples of its usage and by a discussion about its performances in Section 5. Finally, we will introduce in Section 6 a related JavaScript package that provides Node.js bindings to the core library.

2. RELATED WORKS

Few attempts have been made over the years to implement the Web Audio API as an autonomous library that could be used outside Web browsers. For example, the *LabSound* project [7] proposes an open-source C++ library originally forked from the *Webkit* implementation. However, while the library appears to be maintained, the authors state themselves in the `README` of the project that they do not aim at maintaining full compatibility with the specification. Furthermore, they state that "*LabSound* has deliberately deviated from the spec for performance or API usability reasons", which is "expected to continue into the future as new functionality is added to the engine." Derived from this project, the *node-audio* library [8] proposed Node.js bindings built on top of Labsound. However, the project is presented by its author as in an "extremely experimental state" and does not seem to be maintained (i.e. last commit 4 years ago).

The *web-audio-engine* project [13] proposed an alternative approach by proposing an implementation of the Web Audio API written in pure JavaScript, and could therefore be used within Node.js applications. Despite this amusing and didactical approach, the project is obviously tied to the threading limitations of JavaScript and could not have reached any serious performance comparison with low-level implementations. Additionally, the project has been archived by its author and will not receive any further support.

Finally, we can cite the *servo-media* component [9] created in the context of the *Servo* project backed-up by *Mozilla* and implemented in Rust. While much larger in scope, the library proposed a low-level implementation of the Web Audio API. However, we can observe that the public API differs in many regards with the JavaScript API, preventing reuse of acquired knowledge or to easily port code from one language to the other. Additionally, since the abandonment of the *Servo* project by *Mozilla* in August 2020, the library appears unmaintained and hasn't received any updates.

In the growing Rust ecosystem, a number of audio libraries with different scopes and goals have also been proposed. For example, the *dasp* project [3] offers a number of components that provide low-level abstractions for working with digital audio signals. Inside this suite, the *dasp-graph* component aims at creating modular and dynamic audio graphs. However, the component is very general purpose and low-level, and does not provide higher-level building blocks as in the Web Audio API.

On the other side of the spectrum, the *Kira* project [6] provides a high-level "audio library designed to help create expressive audio for games". The library appears to be well maintained and is even recommended as a community plugin alternative for the *Bevy* game engine. However, this focus on game development also means the library may be not suited for more general purpose audio processing such as the Web Audio API.

Inside this frame, we consider that our library has therefore the potential of filling two gaps. First, from a Web Audio API perspective, it provides a native solution decoupled from Web browsers, potentially opening new application areas and helping to widen its community of users. Second, from a Rust perspective, it proposes an intermediary and extensible solution for advanced audio applications that is not yet available in the ecosystem.

3. DESIGN & IMPLEMENTATION

On multi-core processor systems, dynamic audio libraries typically split up work between a *control thread* and a *render thread*. This approach can be seen as a variation of the client / server architecture widely implemented in computer music oriented languages and platforms [17, 22, 23]. The Web Audio API, constrained also by the specifics of the JavaScript language, makes no exception and requires the implementation of this pattern[2]. In such an architecture, the *render thread* has the sole responsibility of rendering the audio graph and shipping the samples to the OS so they can be played by the hardware. This thread therefore has hard real-time constraints: if it is unable to serve up the next block of audio samples in time (128 in the case of the Web Audio API), users will hear catastrophic clicks and pops. The *control thread* on the other hand, is user-facing and orchestrates all changes to the audio graph. It allows the user of the library to add and remove nodes and change their settings. Additionally, other threads can be spawned to handle tasks that could block the control thread for too long. For example, the specification requires a media decoder[3] thread. Our implementation follows this model and uses lock-free message passing for cross thread communication.

It is important to realize that the objects we colloquially call *audio nodes* are distinct from the objects that are placed inside the render thread, which we call *audio processors*. As in figure 1, the *audio node* and the *audio processor* are therefore always created as a pair.

`AudioNode`
: User facing object that implements the `AudioNode` interface from the W3C spec. It does not perform any audio processing, but allows the user to mutate the audio graph, by adding or removing connections or changing the settings of its related `AudioProcessor`.

`AudioProcessor`
: Object that is placed on the render thread, which produces the actual audio frames. It cannot be directly manipulated by the user and relies on instructions received from its corresponding `AudioNode` to change its own behavior.

3.1 Control thread

On the control thread, audio nodes are created, mutated and (dis-)connected. Method calls on the audio context, audio nodes or audio params typically spin off a control message to be handled by the render thread. Hence these methods operate synchronously and do not block, they are therefore safe to use on a UI thread.

In our implementation, following both the specification and the JavaScript behavior, an `AudioNode` can go out of scope (i.e. to `Drop` in Rust idiom) in the control thread, while its processing

[2] https://www.w3.org/TR/webaudio/#control-thread-and-rendering-thread
[3] https://www.w3.org/TR/webaudio/#dom-baseaudiocontext-decoding-thread

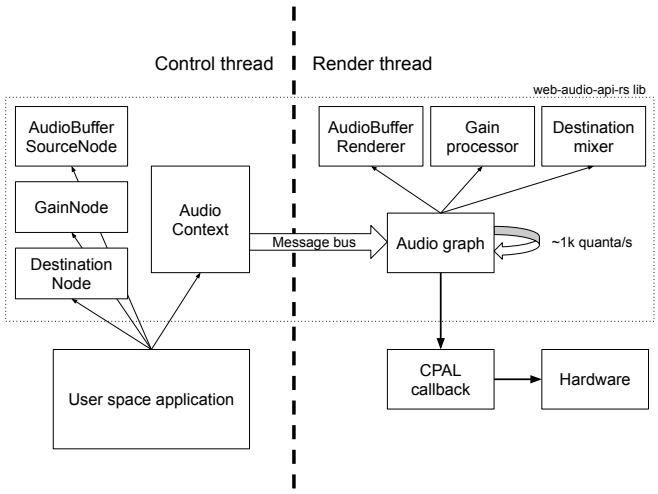

Figure 1: Architecture of the library, showing the components living in the control thread (audio nodes and the audio context) and components that live in the render thread (the audio graph and the audio processors).

counterpart continues to run inside the render thread. In such a case, the renderer will be released only when it has finished processing (governed by the `tailTime` behavior). On the contrary, if the `AudioContext` itself is dropped in the control thread, the corresponding render thread is immediately halted and all resources are released.

3.2 Render thread

Our implementation makes use of the `cpal` library [2] as the audio backend. This library will set up a real-time high priority OS thread to collect the audio samples. This is the thread in which the `AudioProcessor`s run. The entry point of `cpal` is a callback that expects us to fill an array of float or integer interleaved audio samples (i.e. a `FnMut(&mut [f32|u16|i16])` in Rust idiom). While the desired size of the output array can be specified (e.g. 128 samples per channel in the Web Audio API specification), in practice this value is not always available or allowed by the underlying system. In such case we still render the audio graph piecewise in blocks of 128 samples (maintaining therefore the accuracy and regularity of the audio time), but perform internal buffering to align the buffer sizes. For example, if the `cpal` callback data size is 192, two buffers of size 128 will be rendered on one over two `cpal` calls.

The `AudioGraph` is responsible to store the `AudioProcessors` and to ensure that they are rendered in the right order by performing a topological sort of the audio nodes as defined in the specification[4]. This algorithm ensures that when node A has an outgoing connection to node B, the processor of A is called before B. Therefore, when A has rendered, its resulting samples are copied and mixed appropriately to serve as input for processor B. Each time a node is added or removed from the graph, or when the connections between them are altered, the previous ordering is cleared and recalculated. Additionally, during this topological sort, a cycle detection is performed to mute every node that is part of the cycle unless a `DelayNode` is present. To implement the *dynamic lifetime* of the audio nodes described in Sec. 3.1, the `AudioGraph` will also keep track of the processors that have finished their rendering and will consequently clean up branches of the audio graph that will no longer emit output.

[4]https://www.w3.org/TR/webaudio/#rendering-loop. Note that there are subtle differences in our implementation because we are storing edge information in a different way. The resulting sort order conforms to the specification though.

The design of the `AudioProcessor` interface is very much inspired by the `AudioWorkletProcessor` interface as defined in the specification [16]. An `AudioProcessor` is therefore a stateful object that will execute on every render tick a callback of the following form:

```
callback AudioWorkletProcessCallback = boolean (
    FrozenArray<FrozenArray<Float32Array>> inputs,
    FrozenArray<FrozenArray<Float32Array>> outputs,
    object parameters
);
```

Here, `inputs[n][m]` and `outputs[n][m]` follow a planar layout where arrays of audio samples are stored in the `m`th channel of the `n`th input or output. The `parameters` object contains the computed values for each `AudioParam` of the `AudioProcessor` for this rendering quantum. The return value corresponds to the `tailTime` behavior, which allows the `AudioProcessor` to be dropped by the `AudioGraph` when it has finished rendering (i.e. when returning `false`) and has no input connections. In our Rust implementation, this IDL has been adapted to:

```
fn process(
    &mut self,
    inputs: &[AudioRenderQuantum],
    outputs: &mut [AudioRenderQuantum],
    params: AudioParamValues,
    scope: &RenderScope,
) -> bool;
```

The `AudioRenderQuantum` type is a specialized container type for the sample frames, which uses fixed sized, reference counted arrays with *copy on write* semantics. This allows for very efficient implementation of up/down-mixing, fan-in/out of channels and, in a general way, for moving input and output buffers around without making copies and allocating memory.

The `RenderScope` reference is modeled after the specification of the `AudioWorkletGlobalScope`. It contains the current time, sample frame and sample rate, and in the future it will allow processors to share code and data (such as a wavetable or impulse response) within the render thread.

3.3 Message passing

A single, asynchronous communication channel is established from the control thread to the render thread. Following the Web Audio API specification, this is called the *control message queue*. The queue is a FIFO queue where items are ordered by time of insertion, the oldest message being at the front of the queue. The reason a single message bus is used instead of multiple ones, for example from every node to its corresponding processor, is to ensure that audio graph updates cannot be applied out of order. This is best shown with an example:

```
// create a new OscillatorNode
let osc = context.create_oscillator();
// set the frequency to 500Hz (mutate renderer)
osc.frequency().set_value(500.);
// connect oscillator to destination (mutate graph)
osc.connect(&context.destination);
```

We can see here that if the audio graph topology messages are handled out of order with the audio node processor setting messages, it could be possible for the `OscillatorNode` to play at the wrong frequency (i.e. 440 Hz) for a single render quantum.

3.4 Tradeoffs Between Spec and Language

While an important goal for our library is to adhere to the specification as much as possible, some differences are impossible to avoid in translating an API targeted at JavaScript to Rust. The most obvious deviations are that we chose to 1. implement all `AudioNodes` in a thread-safe way and 2. to conform to the Rust code style standard, which enforce the use of `snake_case` for methods and of `CamelCase` for `enum` variants. [5]

The most important differences lie in the implementation of some object-oriented programming concepts, which are very common in the specification but that are not supported by the language. First, Rust does not allow us to override property getters and setters. Therefore, instead of directly exposing the `AudioBufferSourceNode::buffer` attribute, we offer two methods: `AudioBufferSourceNode::buffer() -> AudioBuffer` and `AudioBufferSourceNode::set_buffer(buffer: AudioBuffer)`[6]. Second, Rust is strongly designed toward composition rather than inheritance. Therefore to model inheritance, we decided to use `Traits` for parent / extendable interfaces such as `AudioNode`, `AudioScheduledSourceNode` or `BaseAudioContext`. For dictionaries, we decided to use composition of `Structs` to model inheritance, for example for the `AudioNodeOptions`. Third, Rust does not provide any mechanism for method overloading. In such cases, rather than providing a single method with `Option` parameters, we chose to expose several specialized methods. For example `AudioScheduledSourceNode::start([time])` is derived as `start()` and `start_at(time: f64)`.

A concept that is very specific to Rust is the concept of data *ownership*. Ownership is a mechanism to prevent concurrency bugs and data races that is enforced at compile time so there is no runtime overhead. This means we can avoid needless copies of data because the language can guarantee us we have a non-aliased, mutable reference of data. For example, the `AudioBufferSourceNode` takes an `AudioBuffer` by value so we do not need to implement the idea of "acquiring the content" as described in the specification[7]. Of course, it is still possible to reference and reuse an `AudioBuffer` but the user needs to be explicit and clone the data before feeding it into the source node. Note that such clones are generally cheap in terms of memory because the buffers are reference counted containers implementing a *copy-on-write* semantics, and will therefore be lazily copied only if mutated through `copyToChannel`.

Finally, the Web Audio API is relatively isolated from other web standards which makes it quite simple to decouple it from the Web. However, this does not hold for some nodes such as the `MediaStreamAudioSourceNode` (input) or the `MediaStreamAudioDestinationNode` (output) nodes. We consider these I/O possibilities to be essential for building rich apps such as video conferencing or online gaming. Therefore, we tried to come up with an API boundary interface as simple and flexible as possible to tackle this question, and hence modeled the media streams as a Rust style iterator of fallible buffer items: the `MediaStream` trait.

[5] Fortunately, the search function of the auto-generated documentation is not sensitive to these discrepancies, meaning users should have no issue in finding the right spelling of a method or enum.

[6] Some attribute names such as `type` and `loop` are reserved keyword in Rust, in these cases we decided to append an underscore to the getter, e.g. `OscillatorNode::type_`

[7] https://www.w3.org/TR/webaudio/#acquire-the-content

3.5 AudioWorklet and Extensibility

The question of extensibility is an important aspect that has been handled in the specification with the introduction of the *AudioWorklet* interface [16]. Such an interface is required in JavaScript because of the particular nature of the language and of the necessity to run arbitrary code in the high-priority audio thread. In a Rust context, this question is however posed differently as these limitations and constraints do not hold anymore.

As our implementation of audio processors closely resembles the signature of the `AudioWorkletProcessor` callback (see Section 5.1), we have thus decided to not implement the entirety of the `AudioWorklet` interface. Instead, we expose our `AudioProcessor` trait in the public API of our library and users are advised to use it to build their own custom nodes. Over time we may decide to still implement a spec compliant interface for worklets if such a feature appears important for users. However, the performances will remain suboptimal compared to implementing the lower level interface.

3.6 Current Known Limitations

Two important concepts of the Web Audio API are currently missing in our implementation: asynchronous functions (methods that return a `Promise`) and event handling (attributes of type `EventHandler`).

The Rust equivalent of a function returning a `Promise` is an `async` function. The `async` execution model landed in the core language in 2019 [1]. This opened the possibility for libraries that perform I/O to offer asynchronous versions of their functionality. Users should bring in their own executor of choice (typically `tokio` [10]) which will poll the pending I/O operation concurrently. Our implementation currently only offers a synchronous version of the async functions in the specification. Our progress on this point is also dependent on downstream libraries (mainly the media decoder) as they must support asynchronous execution before we can tap into it.

The concept of event handling has no equivalent in the Rust language, so we will have to come up with our own adaptation. Analogous to decisions made regarding the interface of other web standards discussed in 3.4, we will eventually come up with a simplified interface that allows us to hook a callback into the event targets. Event propagation and cancellation are probably not meaningful in our context.

Besides these general questions, a few limitations of our implementation at time of writing are worth mentioning. First, the `ConvolverNode` and `DynamicsCompressorNode` are not yet implemented, as well as some important features of the `PannerNode` such as HRTF support. Second, the `OscillatorNode` does not fully adhere to the specification: the built-in oscillators are not based on `PeriodicWave` and only applies rudimentary strategy to prevent aliasing (i.e. polyBLEP). Finally, the `DelayNode` does not yet allow sub-quantum delays when not in a cycle.

4. EXAMPLE USE

In this section, we present two examples illustrating the current state of the public API and some possibilities already offered by the library[8].

4.1 Feedback Delay

The example in Figure 2 shows a simple implementation of a feedback delay implemented with our library. The

[8] A number of examples are available in the `examples` directory of the repository

example also highlights different automation methods of the `AudioParam`, as well as the use of the scheduling methods of the `AudioScheduledSourceNode` interface.

```rust
use std::{thread, time};
use web_audio_api::context::{AudioContext, BaseAudioContext};
use web_audio_api::node::{AudioNode, AudioScheduledSourceNode};

fn main() {
    let audio_context = AudioContext::new(None);

    // feedback delay chain
    let delay = audio_context.create_delay(1.);
    delay.delay_time().set_value(0.3);
    delay.connect(&audio_context.destination());

    let feedback = audio_context.create_gain();
    feedback.gain().set_value(0.85);
    feedback.connect(&delay);
    delay.connect(&feedback);

    // trigger an osc with an envelop
    let now = audio_context.current_time();

    let env = audio_context.create_gain();
    env.connect(&audio_context.destination());
    env.connect(&feedback);
    env.gain()
      .set_value(0.)
      .linear_ramp_to_value_at_time(0.5, now)
      .exponential_ramp_to_value_at_time(0.0001, now + 1.);

    let osc = audio_context.create_oscillator();
    osc.connect(&env);
    osc.frequency().set_value(200.);
    osc.start_at(now);
    osc.stop_at(now + 1.);

    thread::sleep(time::Duration::from_secs(15));
}
```

Figure 2: Example code of a simple feedback delay

4.2 Granular Scrub

The example in Figure 3 presents a scrubbing effect realized using a granular synthesis approach. The example also shows how a given *File* can be decoded into an `AudioBuffer` that is then consumed by `AudioBufferSourceNode`s scheduled to read back and forth the `AudioBuffer` at half speed.

5. PERFORMANCE

Audio processing systems typically face hard real-time constraints: when the production of new sample frames lags behind the actual playback, users will be faced with catastrophic clicks and pops. Therefore the performance of the render thread must be optimized. If the overhead of the audio processing library is low, users can build richer audio graphs without running into issues. The performance of the control thread operations is of lesser concern because it will have no influence on audio playback. UI thread operations should never take more than a few milliseconds though.

The `web-audio-api-rs` library currently has not been profiled for performance extensively. Some initial results are presented in this section but we expect them to change significantly over time.

5.1 Render Thread Considerations

So far we have invested in three classes of performance optimizations.

The first and foremost goal is to prevent allocator calls in the render loop by reusing heap allocated objects. Allocator calls in Rust acquire a program-wide lock on the system allocator and are unpredictable in their timing, especially in contended use cases. To deal with this, we have built a custom allocator on top of the default Rust allocator in order to reuse channel sample vectors as they propagate through the audio graph. More work is still to be done: whenever nodes are added or removed, or audio param automation events are handled, (de)allocations may still occur. Eventually we will add

```rust
use std::fs::File;
use std::{thread, time};
use web_audio_api::buffer::AudioBuffer;
use web_audio_api::context::{AudioContext, BaseAudioContext};
use web_audio_api::node::{AudioNode, AudioScheduledSourceNode};

fn trigger_grain(
  audio_context: &AudioContext,
  audio_buffer: &AudioBuffer,
  position: f64,
  duration: f64,
) {
  let start_time = audio_context.current_time();

  let env = audio_context.create_gain();
  env.connect(&audio_context.destination());
  env.gain()
    .set_value(0.)
    .set_value_at_time(0., start_time)
    .linear_ramp_to_value_at_time(1., start_time + duration / 2.)
    .linear_ramp_to_value_at_time(0., start_time + duration);

  let src = audio_context.create_buffer_source();
  src.set_buffer(audio_buffer.clone());
  src.connect(&env);
  src.start_at_with_offset(start_time, position);
  src.stop_at(start_time + duration);
}

fn main() {
  let audio_context = AudioContext::new(None);
  // grab an AudioBuffer from file
  let file = File::open("samples/sample.wav").unwrap();
  let audio_buffer = audio_context.decode_audio_data_sync(file).unwrap();
  // launch granular scrub process
  let period = 0.05;
  let duration = 0.2;
  let mut position = 0.;
  let mut incr = period / 2.;

  loop {
    trigger_grain(&audio_context, &audio_buffer, position, duration);

    if position + incr > audio_buffer.duration() - (duration * 2.)
      || position + incr < 0.
    {
        incr *= -1.;
    }
    position += incr;
    thread::sleep(time::Duration::from_millis((period * 1000.) as u64));
  }
}
```

Figure 3: Example code of a scrubbing effect realized with a granular synthesis approach.

a constructor for the `AudioContext` that can specify its required capacity upfront and will run guaranteed allocation-free.

Second we avoid operations that block, such as acquiring a lock on a mutex. When high-priority render thread blocks on a resource that is held by a lower priority thread, priority inversion [18] may occur on some systems. This will lead to non permissible delays in producing the next sample frames. Instead of utilizing mutexes to share state between threads, we use lock-free message passing and atomic primitive types (e.g. `AtomicU64`).

Last, we attempt to write all audio processor code in such a way that the compiler can apply auto-vectorization of transformations of the sample vectors. This means that the following simplified snippet for applying a volume gain will be compiled to SIMD instructions:

```rust
pub fn apply_gain(
    input: &[f32],
    output: &mut [f32],
    gain: f32,
) {
    input
        .iter()
        .zip(output.iter_mut())
        .for_each(|(i, o)| *o = i * gain)
}
```

We can verify this optimization takes place with the Godbolt compiler explorer using the right optimization flags (`-C opt-level=3 -C target-cpu=native`) [4]. We are also working on an optimization that detects if all channels in an audio buffer are equal. This is very common in Web Audio API programs because the default behavior of almost all nodes is to upmix their input to a 2-

channel speaker output. For example, when a mono signal is fed to a `GainNode` with default settings, the mono signal will be upmixed to stereo and the gain will be applied to both identical channels. In our implementation, upmixed buffers will only apply the transformation once.

5.2 Benchmarks

To evaluate current performances of the library, we ported the benchmarks developped by P. Adenot from [15] into Rust. The main idea of these benchmarks is to calculate an `AudioBuffer` (generally of 120 seconds) using an `OfflineAudioContext`, measure the time needed to calculate the buffer (i.e. between the start and the end of the rendering) and compare this value to the duration of the buffer, giving therefore an estimation of the performances according to real-time. From the available benchmarks, we could port all of them except the one regarding the `ConvolverNode` which is not yet implemented in our library. We consider this approach interesting as it should minimize the JavaScript overhead in browsers and therefore allow us to almost compare against the bare-bone C++ implementations.

The benchmarks have been run on two different computers: a MacBook Pro 2019 with a 2,3 GHz 8-Core Intel Core i9 processor and 16 GB of 2400 MHz DDR4 RAM, and a MacBook Pro 2020 with a Apple M1 processor (ARM) and 8 GB of LPDDR4X-4266 MHz RAM. For each tested platform (Google Chrome, Firefox, and our library) the benchmarks have been run 5 times. Reported results are the mean of these 5 runs and are the speed-up compared to real-time (i.e. $duration/processingTime$), so the higher the better.

As shown in the results presented in Table 1 and 2 in the appendix, we can consider that our library has room for improvement compared to these state of the art implementations, but is not completely out of the game: 2.8 slower than Chrome and 5,7 times slower than Firefox on the MacBook Pro 2019.

We think these results, while not completely fair for all cases (see in particular Section 3.6) are quite encouraging, especially considering that our work so far has been focused on compliance and API stabilization rather than performance. Moreover, while developing the library we have not heavily studied the source code of the Firefox and Chrome implementations. These implementations have been optimized for years and will allow us to cherry pick relevant improvements, such as node pre-assignment and pooling, for our own codebase.

6. NODE.JS BINDINGS

In this section, we present `node-web-audio-api-rs`[9], a related `npm` package that provides Node.js bindings to `web-audio-api-rs`. We present first the general goals and the technical approach we followed, and then illustrate its usage with an example. The library is still in its early developments and does not yet cover all the features implemented in `web-audio-api-rs`.

6.1 General Approach and Goals

Our main goal in developing this bindings is to propose a perspective we could call *Isomorphic Web Audio*. We believe such an approach could prove to be interesting contribution in several areas such as distributed music systems [21] or more generally for the field of Internet of Musical Things [25]. Indeed, it could allow to seamlessly develop JavaScript audio components able to run both in the browser and on headless nano-computers such as the Raspberry PI. Another potential of the package is to enable the testing of the core `web-audio-api-rs` library against the official Web Audio test suite [14], therefore contributing back to its development and compliance.

The package is build using `napi-rs`[10], which propose Rust to `Node-API` bindings, and should therefore garantuee forward compatibility with Node.js. The approach we choose to follow is to derive and generate most of bindings code from specification IDL, using JavaScript as a parser and a template engine for the Rust code. While still in its early stage of development, we hope this approach will help to maintain the package up-to-date according to the core library. Finally, we took care of hiding the specifities of the Rust API described in Section 3.4 to provide an interface that is exactly similar to the one exposed in Web browser. Such an approach could, at least in theory, allow to simply reuse a whole set of existing higher-level libraries [24] [19] in a Node.js context.

6.2 Example Use

```
const path = require('path');
const { Scheduler } = require('waves-masters');
const { AudioContext, load } = require('node-web-audio-api-rs');

const audioContext = new AudioContext();
const scheduler = new Scheduler(() => audioContext.currentTime);
const file = load(path.join(__dirname, 'sample.wav'));
const buffer = audioContext.decodeAudioData(file);

const period = 0.05;
const duration = 0.2;
let incr = period / 2;
let position = 0;

const engine = {
  advanceTime(currentTime) {
    if (
      position + incr > buffer.duration - 2 * duration ||
      position + incr < 0
    ) {
      incr *= -1;
    }

    const env = audioContext.createGain();
    env.connect(audioContext.destination);
    env.gain.value = 0;
    env.gain.setValueAtTime(0, currentTime);
    env.gain.linearRampToValueAtTime(1, currentTime + duration / 2);
    env.gain.linearRampToValueAtTime(0, currentTime + duration);

    const src = audioContext.createBufferSource();
    src.connect(env);
    src.detune.value = Math.random() * 2 * detune - detune;
    src.buffer = buffer;
    src.start(currentTime, position);
    src.stop(currentTime + duration);

    position += incr;

    return currentTime + period;
  }
}

scheduler.add(engine);
```

Figure 4: Example code of the granular scrub example rewritten back to JavaScript.

The example code in Figure 4 shows the granular scrub example presented in Section 4.2, rewritten back in JavaScript. The example also highlights how an existing component, i.e. the lookahead `Scheduler` provided by the `waves-masters` library [11], can be simply reused in this context.

7. CONCLUSION

In this paper we have presented `web-audio-api-rs`, a novel implementation of the Web Audio API specification implemented in the Rust language. We first described the general design of the library, expliciting and justifying the trade-offs that have been made to the specification in regards to the specificities of the Rust language. We then completed this general presentation with some examples and a picture of its current performance. Finally, we introduced a related JavaScript package that proposes Node.js

[9] https://github.com/b-ma/node-web-audio-api-rs

[10] https://napi.rs/

bindings to the core library.

While still in its early stage, the library already proposes a stabilized API and implements an important subset of the specification. We think this novel implementation has the potential to provide an interesting solution in several directions: open new application areas for the Web Audio API community by decoupling the library from the Web, and provide an intermediary and extensible solution for audio applications that is not yet available in the Rust ecosystem. We hope that future evolutions of the library, backed by new and diverse contributors, will allow us to mitigate the limitations described in this paper and offer a worthy alternative to the in-browser implementations.

8. ACKNOWLEDGMENTS

We would like to thank our contributors, especially *Jerboas86* for his precious contributions to the project. We would also like to thank our colleagues at IRCAM for their support on this project.

9. REFERENCES

[1] Announcing rust 1.39.0. https://blog.rust-lang.org/2019/11/07/Rust-1.39.0.html.

[2] CPAL - Cross-Platform Audio Library. https://github.com/rustaudio/cpal.

[3] dasp - digital audio signal processing in rust. https://github.com/RustAudio/dasp.

[4] Godbolt compiler explorer - simd optimization. https://rust.godbolt.org/z/qv3YfToW9.

[5] JetBrains - the state of developer ecosystem 2021 - rust. https://www.jetbrains.com/lp/devecosystem-2021/rust/.

[6] Kira - expressive audio library for games. https://github.com/tesselode/kira.

[7] Labsound - graph-based audio engine. https://github.com/LabSound/LabSound.

[8] node-audio - graph-based audio api for node.js based on labsound and juce. https://github.com/ramirezd42/node-audio.

[9] Servo Media. https://github.com/servo/media.

[10] Tokio - a runtime for writing reliable asynchronous applications with rust. https://tokio.rs/.

[11] waves-masters - low level components for transport and scheduling. https://github.com/wavesjs/waves-masters.

[12] Web Audio API Specification. https://www.w3.org/TR/webaudio/.

[13] web-audio-engine - pure js implementation of the web audio api. https://github.com/mohayonao/web-audio-engine.

[14] The web-platform-tests project - webaudio. https://github.com/web-platform-tests/wpt/tree/master/webaudio.

[15] webaudio-benchmark. https://github.com/padenot/webaudio-benchmark.

[16] H. Choi. AudioWorklet: The future of web audio. In *Proceedings of the International Computer Music Conference*, Daegu, South Korea, 2018.

[17] F. Déchelle, R. Borghesi, M. De Cecco, E. Maggi, R. Butch, and N. Schnell. jMax: An Environment for Real-Time Musical Applications. *Computer Music Journal*, 23(3):50–58, 1999.

[18] J. Kacur. Realtime kernel for audio and visual application. In *Proceedings of the Linux Audio Conference 2010*, Utrecht, The Netherlands, 2010.

[19] Y. Mann. Interactive Music with Tone.js. In *Proceedings of the 1rst Web Audio Conference*, Paris, 2014.

[20] N. D. Matsakis and F. S. Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, Oct. 2014.

[21] B. Matuszewski and F. Bevilacqua. Toward a Web of Audio Things. In *Proceedings of the 2018 Sound and Music Computing Conference*, Limassol, Cyprus, 2018.

[22] J. McCartney. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68, Dec. 2002.

[23] M. Puckette. FTS: A Real-time Monitor for Multiprocessor Music Synthesis. *Computer Music Journal*, 15(3):58–67, 1991.

[24] N. Schnell, V. Saiz, K. Barkati, and S. Goldszmidt. Of Time Engines and Masters An API for Scheduling and Synchronizing the Generation and Playback of Event Sequences and Media Streams for the Web Audio API. In *Proceedings of the 1rst Web Audio Conference*, Paris, France, 2015.

[25] L. Turchet, C. Fischione, G. Essl, D. Keller, and M. Barthet. Internet of Musical Things: Vision and Challenges. *IEEE Access*, 6, 2018.

APPENDIX

Benchmark	Chrome	Firefox	Rust
Empty testcase	4377.6	15685.8	4178
Simple source test without resampling (120s)	3348.6	5885.6	944.6
Simple source test without resampling (Stereo) (120s)	2635.6	4029.4	734,2
Simple source test without resampling (Stereo and positional) (120s)	1252.8	2959	370.8
Simple source test with resampling (Mono) (120s)	1935.4	2958.2	986,6.4
Simple source test with resampling (Stereo) (120s)	1490	1667	732,6
Simple source test with resampling (Stereo and positional) (120s)	903	1474.8	357
Upmix without resampling (Mono -> Stereo) (120s)	2715	3924.2	869.2
Downmix without resampling (Stereo -> Mono) (120s)	3162.2	3046.2	811.2
Simple mixing (100x same buffer) (30s)	32.6	49.4	10.6
Simple mixing (100 different buffers) (30s)	31.8	46.4	11
Simple mixing with gains (120s)	548.6	1202.4	251.6
Granular synthesis (7.5s)	7	32	1.5
Synth (120s)	82.8	307.6	21.3
Substractive synth (120s)	408.2	639	217.2
Stereo Panning (120s)	1928.2	3707.2	549.6
Stereo Panning with Automation (120s)	1826.4	1349.2	546.6
Periodic Wave with Automation (120s)	1511.6	1001.8	1488.2

Table 1: Benchmark results for the MacBook Pro Intel 2019. The values denote the speedup compared to live audio playback, higher is better. Standard deviation never exceeded 7.5%

Benchmark	Chrome	Firefox	Rust
Empty testcase	10209.8	22400	7058.8
Simple source test without resampling (120s)	7235.4	12266.6	1807.9
Simple source test without resampling (Stereo) (120s)	5714.0	8835	1562.7
Simple source test without resampling (Stereo and positional) (120s)	2765.4	5828.4	548.0
Simple source test with resampling (Mono) (120s)	4286.0	3516	2166.2
Simple source test with resampling (Stereo) (120s)	3046.2	2055.2	1524.7
Simple source test with resampling (Stereo and positional) (120s)	1929.0	1829.4	551.1
Upmix without resampling (Mono -> Stereo) (120s)	6000.0	7703.4	1945.3
Downmix without resampling (Stereo -> Mono) (120s)	6667.0	2299.4	1666.9
Simple mixing (100x same buffer) (30s)	73.0	45.6	23.6
Simple mixing (100 different buffers) (30s)	70.0	42	23.3
Simple mixing with gains (120s)	1138.6	1481.2	379.5
Granular synthesis (7.5s)	12.0	44.2	2.1
Synth (120s)	123.0	419.8	27.5
Substractive synth (120s)	513.0	1111	473.2
Stereo Panning (120s)	4084.6	7914.2	1043.9
Stereo Panning with Automation (120s)	3898.4	2371.8	1010.3
Periodic Wave with Automation (120s)	1968.4	1667.4	1987.3

Table 2: Benchmark results for the MacBook Pro M1 2020. The values denote the speedup compared to live audio playback, higher is better. Standard deviation never exceeded 5%