Proceedings of the 2nd International Faust Conference (IFC-20), Maison des Sciences de l'Homme Paris Nord, France, December 1-2, 2020

AUDIO ANYWHERE WITH FAUST

Benedict R. Gaster *

Computer Science Research Centre
University of West of England
Bristol, UK
benedict.gaster@uwe.ac.uk

Max Cole

University of West of England
Bristol, UK
maxhedleycole@gmail.com

ABSTRACT

This paper introduces *Audio Anywhere* (*AA*), a framework for working with audio plugins that are compiled once and run anywhere. At the heart of Audio Anywhere is an audio engine whose Digital Signal Processing (DSP) components are written in Faust and deployed with WebAssembly.

Unlike previous work, Audio Anywhere does not just run in the browser, in fact, the same portable plugin can run at close to native speed on desktop, tablets, and low-power micro controllers at the edge of the Internet of Musical Things network (IoMusT). It is not our intention to push another audio module or plugin format, rather we are developing Audio Anywhere as a proof that Faust to WebAssembly, combined with HTML5 is a viable platform for portable audio modules or plugins outside of the browser and even on tiny micro-controllers.

In this paper we focus on an example instance of the framework for the desktop, the use of Faust for DSP, lightweight Webviews for Graphical User Interfaces (GUIs), and Rust as a hosting language. The embedded audio platform Daisy is also targeted. We describe our modifications to the Faust compiler, utilizing Rust as an intermediate language to provide access to auto-vectorization of WebAssembly (128-SIMD). A number of example modules are discussed, demonstrating the utility of the framework.

1. INTRODUCTION

> Things would be different one day. But you had to start small, like oak trees.
>
> *Tiffany Aching*

The ideal of compile once and run anywhere[1] has been a dream in computer science for as long as it has been an area of research. From the early days of Lisp, through to Java and Python with its import ideal. However, to date these offerings, as amazing as they are, have failed to reach performance close to what system-based languages C and C++ can achieve. Outside of general purpose programming, certain Domain Specific Languages [1] have achieved excellent performance. For example, in the domain of graphics there are many, including GLSL [2], and in the audio domain the language Faust [3] is an exemplar. However, both OpenGL Shading Language (GLSL) and Faust fall into the system-based languages camps of having to be compiled for each individual platform. Faust is often translated into C++ code, that is then compiled offline for deployment, once for each target platform. GLSL is different as the OpenGL runtime provides routines for compiling to a particular Graphics Processing Unit (GPU) dynamically, and the most recent graphics Application Programming Interfaces (APIs), most notably Vulkan [4], provide an intermediate representation, called SPIR-V [5]. However, SPIR-V, like GLSL, is still designed to be compiled ahead of time and does not easily support JIT style compilation.

Figure 1: Left is a hacked Casio VL-1 with an Electro Smith's Daisy running a AA Module, emulating a version of the VL-1 audio engine. Right is the same emulation running in AA hosting application.

Over the last few years a new kid on the block has emerged as an interesting inflection point in the search for a compile once, run anywhere target for compiling system-based languages[2]. WebAssembly (or Wasm) is an open standard, originally developed by the four main browser vendors, specifying a portable format for executable programs, including interfaces for facilitating interactions between such programs and their host environment [6]. Wasm came from the ashes of projects such as Adobe's Flash and Google's Native Client, with the goal of providing a platform neutral environment for running non Javascript code, as close to native speed as possible, within a modern web-browser. Building on Emscripten's [7] success in compiling the systems languages C and C++ to JavaScript, WebAssembly provides a simple instruction set and sand boxed environment for compiling "low-level" languages, including C++, Go, and Rust to the web. Unlike the virtual platforms of Java, .NET, and Python WebAssembly does not provide garbage collection. Instead an abstraction of linear memory, a simply array of bytes, is provided, fitting naturally with the models of C and C++.

The web browser is an amazing platform and the goal of one day replacing desktop apps with web-apps running in the browser a lofty one, however, while there have been some major advances in real-time audio in the web, e.g. [8, 9, 10], latency limitations still exist. Moreover, Digital Audio Workstations (DAWs) and other

* This work was supported by UWE VC 2019 Award.

[1] A play on Sun Microsystems' slogan "Write once, run everywhere (WORE)".

[2] For the sake of this work we consider system-based languages to be in the guise of C, C++, and Rust, but avoid any formal definition.

audio software that support plugins still run outside the browser, while additionally there are many capable embedded systems, that are not suitable for a browser. The goal of this work is to begin to explore if WebAssembly is a suitable compilation target for real-time audio on the desktop, embedded desktops devices such as the Raspberry Pi or Bela's Beaglebone Black, and even embedded micro-controllers, such as Electrosmith's Daisy platform. As visualized in Figure 2, we are interested in compile once to WebAssembly vs the compile n-times to native machine code, where n is the number of platform OS/native instruction set combinations.

In this paper we introduce Audio Anywhere as a small step towards compile once, run anywhere for audio DSP code. Audio Anywhere combines Faust, for audio DSP code, and HTML5 to enable development of modern audio synthesis and effects tools. The Faust DSP code is complied once into WebAssembly, but unlike early work the resulting audio code is not hosted within a browser, but instead translated on the fly to native code running within a hosting application. This means that, in theory, an Audio Anywhere module can run anywhere, including platforms where browsers are not available. For the initial implementation of Audio Anywhere, User Interfaces (UIs) can be described and implemented in HTML5, but this is not a requirement, as the control component is independent of the audio component. This enables interfaces to range from conventional plugin GUIs, remote controllers connected via the Internet of Musical Things, to Embedded hardware interfaces.

To help validate Audio Anywhere's approach we have developed a standalone hosting app that runs on Windows, Mac OS, and Linux, a VST 2.x plugin host, and a tiny host that runs on the Daisy Seed. To demonstrate the versatility of our approach we have developed a number of example synths and effects, that are compiled once and run on all the platforms. For example, we developed a clone of the Casio VL-1, which runs with a HTML5 interface on the supported desktop platforms, but a variant also uses an actual VL-1 hacked to control a Daisy, as shown in Figure 1.

More details on Audio Anywhere and links to the source code are available from the project website[3].

The remainder of this paper is structured as follows:

- Section 2 takes a look at related and background work;
- In Section 3 we introduce Audio Anywhere and some example applications;
- Section 4, then describes the implementation of DSP algorithms using the Faust programming language and HTML5 for Audio Anywhere;
- Section 5 evaluates the WebAssembly approach to a portable DSP framework; and
- Finally, Section 6 concludes with pointers to future work.

2. BACKGROUND

There is a wide variety of work and technologies that are related to and has inspired this current work. This section considers a few to help provide context and plant seeds for future work, both in audio DSP and Digital Musical Instrument (DMI) design.

The history of the modern Web includes the standardization of what is now termed HTML5. Beginning with work by Mozilla,

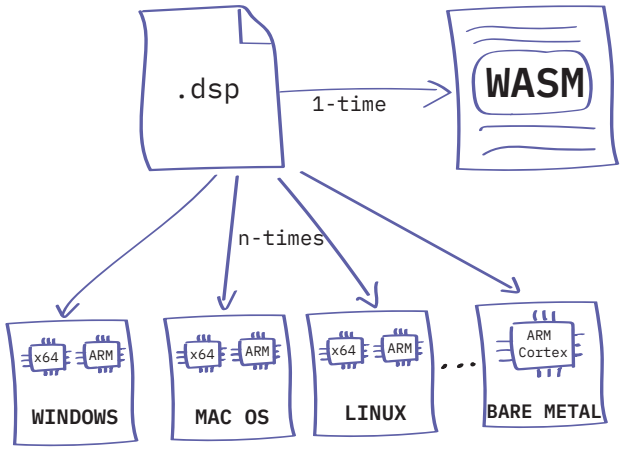

Figure 2: Traditional n-time compilation vs WebAssembly 1-time Compilation

Apple, and Opera begun in the mid 2000s, which pushed the Web towards technology capable of running 3D Games, real-time audio, and more. Javascript is at the heart of today's Web technology and Just In Time (JIT) compilers such as the V8 engine, provide amazing performance, however, it is not without its drawbacks and it is often hard to predict an application's performance. To address some of the performance unknowns of JIT JavaScript compilation Alon Zakai, at Mozilla, developed Emscripten [7], a compiler for C and C++ that targeted a subset of JavaScript, called Asm.js[4], which enabled performance characteristics closer to native code.

Building on Asm.js success, Mozilla and the other popular browser developers came together to specify WebAssembly[6], a binary instruction format for a stack-based virtual machine that targets the web. WebAssembly emerged at a similar time as other innovations in web technology, in particular, Worklets[5], that in combination enable low-level access to both the rendering pipeline and low-latency audio within the browser.

A wide selection of proposals for real-time audio on the web have emerged during this fruitful time for development and in particular, building on from work on the Web Audio API [8], recent proposals have introduced Native Web Audio API Plugins [10] and Faust for the Web [11]. Faust for the Web saw its compiler being extended with a new backend that specifically targeted WebAssembly for the audio DSP code. An important difference in our approach is that Audio Anywhere does not target the web, although it is easy to see that it could indeed be applied there. WebAssembly's development is tightly linked to that of the Rust programming language and there are a number of possible benefits that might be gained by utilizing Rust as an intermediate language for Faust, before compiling to WebAssembly, and the approach is investigated here. One feature that has been enabled by using Rust as an intermediate language is the ability to have Rust's LLVM compiler apply auto-vectoriazation to Faust's resulting compute loop.

The design of portable GUIs is difficult and in the development of Digital Musical Instruments often complicated by the use of

[3] https://muses-dmi.github.io/projects/.

[4] https://en.wikipedia.org/wiki/Asm.js.

[5] https://developer.mozilla.org/en-US/docs/Web/API/Worklet.

Figure 3: Audio Anywhere Module

tactile controls that go beyond the QWERTY keyboard and computer mouse. While C++ frameworks such as GTK and Qt have gone a long way to address early issues with GUI toolkits, they are still often limited to certain platforms and not easy to access for many programmers. The introduction of Electron[6], a cross-platform toolkit for building applications with JavaScript, HTML, and CSS, showed again the promise of Web technology for application development. In particular for GUI applications that could be written in modern web technologies, and customized to the look and feel required by the application developer, not, necessarily, matching the native look and feel of the host OS. A downside of Electron is the dependency on NodeJS and Chromium. With the development of Audio Anywhere we were keen to support desktop applications with modern custom GUIs, but at the same time wanted to avoid the heavy dependencies that Electron introduces. Instead we chose to use light weight Webviews[7], aiming to utilize a common HTML5 UI abstraction layer for the most widely used platforms.

3. OVERVIEW

In this section we give an overview of Audio Anywhere, introducing some of the tools and ideas. In general, an Audio Anywhere application is made up of a DSP component, that consumes and produces audio samples at the block level, a control component or UI, and a host.

The DSP component is delivered as WebAssembly, that conforms to a specified API and is the same for all platforms and hosts[8]. The code compiled to generate the WebAssembly can be from any language, however, the resulting WebAssembly can depend only on functions and values defined by the AA external library. This enables portability requirements to be stated for all AA modules. The control or UI components are independent of the DSP component and currently AA supports UIs developed in HTML5 for desktop platforms, and a simple physical control library for embedded Daisy platforms. This latter library currently provides support for buttons, potentiometers, and Bela Trill capacitive touch sensors[9].

An AA hosting application is not formally specified, other than to state that they should be compatible with loading AA DSP code and one of the supported UI models. For example, the standalone desktop host can dynamically load and unload AA modules, consisting of AA DSP Wasm and HTML5 interface, while the Daisy firmware loads a predetermined AA DSP Wasm and has a UI baked in directly. To enable dynamically loading and unloading of AA components a notion of AA module is defined. An AA module or bundle contains everything necessary for the instrument

[6] `https://www.electronjs.org/`.

[7] `https://github.com/webview/webview`.

[8] In fact as WebAssembly Single Instruction Multiple Data (SIMD) support is still limited, 128 vector SIMD is optionally supported.

[9] `https://bela.io/products/trill/`.

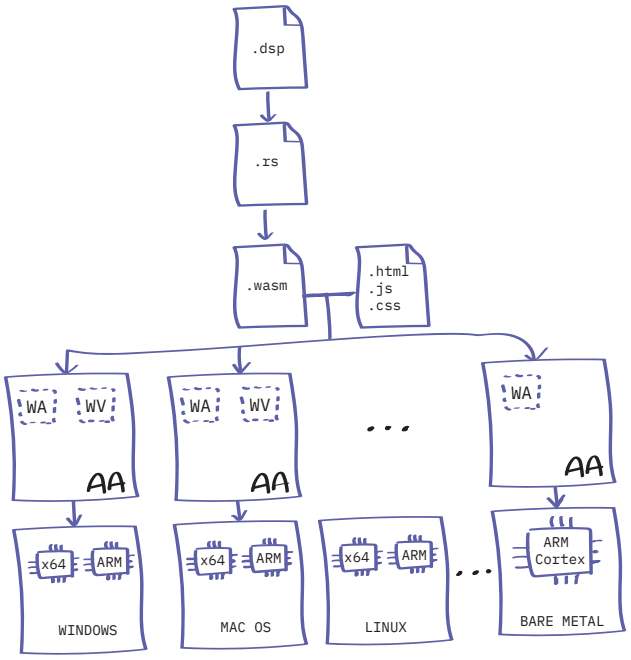

Figure 4: Audio Anywhere Module Compilation

to work within a specified host, in particular, the AA DSP Wasm and UI components, and a description of how they are connected together.

AA does not directly provide support for control standards such as MIDI or OSC, but the AA API provides entry points that can easily handle values produced by incoming control, note on, and off messages. All of the current hosting applications provide support for a subset of MIDI.

In general, AA DSP code can be written in any language suitable for compiling to WebAssembly, in this paper we focus on DSP code written using Faust. The following subsections provide more detail of our current design and implementation, beginning with AA modules, then looking at the use of HTML5, and finally introducing `faust2audioanywhere`, a tool for compiling Faust to AA DSP Wasm, via Rust. This is followed by details of the implementation, with particular focus on Faust to AA DSP Wasm, in the Section 4.

3.1. Audio Anywhere Modules

An AA module is currently defined in terms of its DSP code, a UI component, and meta-data used to load a module dynamically at runtime. While AA's module definition does not specify anything about how the UI component should be implemented, a particular AA hosting application is likely to do so. For example, we have currently implemented a standalone hosting application for desktop (Windows, Mac OS, and Linux) and Raspberry Pi (Linux) that requires UIs to be implemented in HTML5. An AA module in this case is encapsulated as a 'bundle' of all these components, as visualized in Figure 3.

An AA modules is defined in terms of these components and assuming they implement the defined AA Wasm and UI APIs, given below, then the modules themselves can be implemented with any framework or tools preferred by the user. To make the

```
{
    "wasm": [String],
    "gui": String,
    "info": {
        ...
        "inputs": int,
        "outputs": int,
        ...
    }
}
```

Figure 5: JSON Representation of a AA Bundle

initial development of AA modules faster and easier we have developed a framework for building them. For audio components, a modified version of the Faust compiler is used to implement audio DSP from Faust's DSP code, while the HTML5 UIs are written with a small, no dependence, JavaScript framework. Figure 4, visually captures the compilation workflow, with Faust DSP code compiled to Rust, which in turn is compiled to Wasm. The GUI code is written in pure JavaScript, along with any necessary CSS, and very minimal HTML. The resulting outputs are then combined to form an AA bundle, which can be loaded and executed using a WebAssembly virtual machine on a variety of different target platforms. For desktop and Raspberry Pi a tiny cross-platform Webview library is used to render the HTML5 components, while on embedded platforms, such as the Daisy[10], UI elements are connected to physical buttons, sliders, etc.

For simplicity a module's bundle is currently defined with JSON, a subset of which is shown in Figure 5. The key elements are defined as follows:

- URL: $module.wasm$, implementing the Wasm API for audio plugins;
- URL: $gui.html$, implementing the Javascript API for UI plugins; and
- Set of configuration parameters, which are defined as part of the module bundle itself and describe its capabilities, parameter mappings, and initialization constants.

3.2. Audio Anywhere Wasm API

An AA module must implement the AA API interface. The API is specified in Rust, but with a C ABI[11]. The reasons for a C ABI are twofold:

- Unlike native programming languages such as C++ or Rust, C's ABI is explicitly defined with no name mangling and complex types;
- WebAssembly supports only a basic export and linking mechanism, which C ABI maps directly, without renaming of symbols and so on. This avoids having to define the API in Wasm itself, benefiting from the high-level abstractions of C definitions.

As it is expected that not all uses of *AA* will utilize Rust, definitions of the API are also provided as a C header file. An exemplar

```
// initialize module
#[no_mangle]
pub fn init(sample_rate: f64)

// module meta data
#[no_mangle]
pub fn get_sample_rate() -> f64
#[no_mangle]
pub fn get_inputs() -> u32
#[no_mangle]
pub fn get_outputs() -> u32
#[no_mangle]
pub fn get_voices() -> i32

// parameters
#[no_mangle]
pub fn get_param_index(length: i32) -> i32
#[no_mangle]
pub fn get_num_params_float() -> u32
#[no_mangle]
pub fn set_param_float(index: u32, v: f32)
#[no_mangle]
pub fn get_param_float(index: u32) -> f32

#[no_mangle]
pub fn handle_note_on(mn: i32, vel: f32)
#[no_mangle]
pub fn handle_note_off(mn: i32, vel: f32)

// compute audio
#[no_mangle]
pub fn compute(frames: u32)

// input and output buffer management
#[no_mangle]
pub fn get_input(index: u32) -> u32
#[no_mangle]
pub fn get_output(index: u32) -> u32
#[no_mangle]
pub fn set_input(index: u32, offset: u32)
#[no_mangle]
pub fn set_output(index: u32, offset: u32)

// extern functions provided by AA Host
extern "C" pub fn set(index: u32, v: f32)
```

Figure 6: Subset of the Audio Anywhere Wasm API

[10] https://www.electro-smith.com/daisy/
[11] Abstract Binary Interface.

```
declare aavoices "2";

import("stdfaust.lib");

process = vgroup("voices",
                 par(n,
                     2,
                     vgroup("aavoice%n",
                         voice))) ;

voice = hgroup("midi", osc(freq))
with {
 freq = hslider("freq",200,50,1000,0.01);
 gain = hslider("gain",0.5,0,1,0.01);
 gate = button("gate");
 envelope =
   en.adsr(0.01,0.01,0.8,0.1,gate)*gain;

 osc(freq) = os.sawtooth(freq)*envelope;
};
```

Figure 7: Simple Faust DSP code for Synth

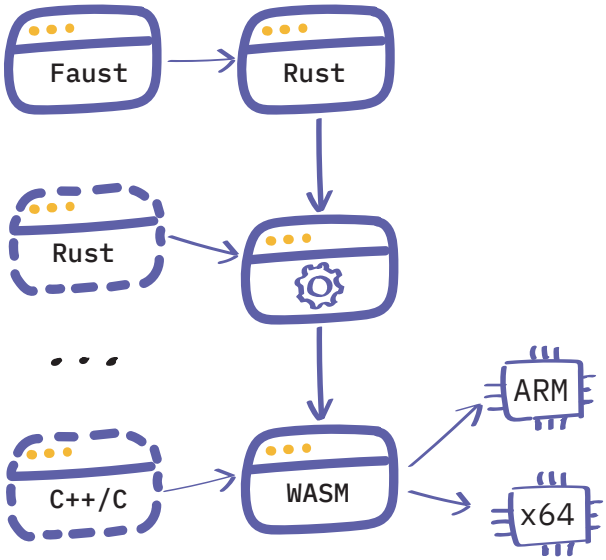

Figure 8: Compilation flow for AA DSP Code.

subset of the API is given in Figure 6. The particular subset is representative of the API as a whole and is also close to the corresponding Faust API, generated by our modified compiler, but also the standard Faust API itself.

3.3. Generating DSP Structs with *faust2audioanywhere*

`faust2audioanywhere` is a command-line tool that can be used to generate new DSP structures, in Rust, that are compatible with the Audio Anywhere API. It is invoked as follows:

```
faust2audioanywhere synth.dsp
```

which produces a Rust source file, `synth.rs`. Implementation details of the resulting Rust file are given in Section 4. By default, it is assumed that there is only a single voice, but polyphony can be enabled with an additional command line option. For example, the following assumes a synth with 2 voices:

```
faust2audioanywhere -voices 2 synth.dsp
```

It should be noted that unlike the approach to multiple voices described in [12], the number of requested voices must match that defined in the Faust DSP implementation. More details on our approach to polyphony is given in Section 4.2.

4. IMPLEMENTATION

This section gives details of the current prototype implementation of Audio Anywhere, with particular focus on the Faust to Rust compilation path for generating Wasm modules. Additionally, we describe AA module hosting applications for the desktop and embedded platform, including how AA Wasm modules are compiled and run on Electrosmith's Daisy.

For illustration throughout this section, the simple Faust synth in Figure 7, will be used for reference.

4.1. Compiling Faust To Rust

Figure 8, shows the path of compilation for DSP code into an AA Wasm module. In our particular case Faust, but, in general, the DSP code can be written in any programming language suitable for compilation to WebAssembly. The number of input and output buffers is known at compile time and thus the resulting code can vary on per-module basis. An additional set of arrays is allocated, whose elements contain pointers to input and output buffers, respectively. In practice this extra level of indirection is only necessary to support chaining of AA modules within the Wasm address space and is discussed in more detail in Section 4.3.

Figure 9 shows the general layout of memory for a module. All memory is allocated statically when a module is loaded, as the number of input and output buffers is known at compile time. For simplicity it is assumed that a maximum buffer size, in our case 512, is defined, and allocations are done to meet this maximum, even in the case when the actual required buffer size is less. In most of our tests we assume a buffer size of 64, for example. While this simplifies memory allocation, it does, in some cases lead to over allocation, however, in practice we have found that the trade of is quite small and compared to explicitly requiring a memory allocator, a much smaller price to pay. Of course, for an embedded target, where memory is often a scarce resource, this might not always be the best choice.

As AA modules are closed under the API defined in Figure 6 the code generated by the Faust compiler is not externally accessible and our implementation takes advantage of this with the introduction of an additional `compute` function. The sole purpose of this function is to setup the input and output buffers for the Faust generated `compute` function. The code generated by the AA Faust compiler for our running example is as follows:

```
pub fn compute_external(
        &mut self, count: i32) {
  let (output0, output1) = unsafe {
    (::std::slice::from_raw_parts_mut(
```

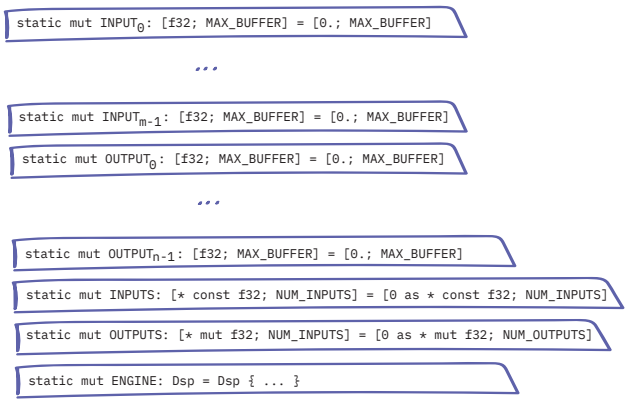

Figure 9: AA module static memory layout.

```
        OUTPUTS[0], count as usize),
    ::std::slice::from_raw_parts_mut(
        OUTPUTS[1], count as usize)) };
  unsafe { self.compute(count, &[], &mut [
      output0, output1]); }
}
```

Unlike the Faust `compute` function, `compute_external` is externally visible, called by the AA API implementation itself, and its only parameter is the number of samples to compute. It constructs the slices required for the Faust `compute` function and proceeds to call this function to actually perform the audio computation. In practice this function is marked as inline and is optimized away completely, by the Rust compiler.

The original Faust compiler supporting Rust as a target language produces a compute function that is hard to vectorize, at least automatically, and thus runs considerably slower that the corresponding C++ code. This issue and corresponding solution was discovered both during this work and also by Github user Bluenote10 (Fabian Keller)[12], and benchmark results in Section 5 demonstrate these performance differences.

Unlike C++, Rust provides certain safety guarantees including array bounds checking. As it is not possible, in general, to know the value for an index into an array statically, in the worse case Rust must check each array access, as Faust's compute function accesses input and output buffers in its hot loop. This bounds checking was a key factor in causing the Rust generated audio loop to be considerably slower than the equivalent C++, but it was not the only one. Bounds checking adds considerable overhead, in execution performance and also in code size, but it was also stopping the LLVM auto-vectorizer from vectorizing the audio loop! The solution to this performance bottleneck is well known to the Rust community; rather than using traditional indexed `for` loops, even ones that don't have loop carry dependencies, recasting the problem in terms of iterators and zippers allows the Rust compiler to avoid generating bounds checked access, which in turn enables LLVM to successfully apply its auto-vectorizer.

A subset of the generated compute function is given in Figure 10. Generated from our running example it highlights that the output channels are initially separated out, enabling them to be turned into iterators, mutable as they are outputs, followed by the

```
#[target_feature(enable = "simd128")]
unsafe fn compute(&mut self,
   count: i32, inputs: &[T], outputs: &mut
       [&mut [T];2]) {
  let [outputs0, outputs1] = outputs;
  let (outputs0, outputs1) = {
      let outputs0 =
          outputs0[..count as usize]
                  .iter_mut();
      let outputs1 =
          outputs1[..count as usize]
                  .iter_mut();
      (outputs0, outputs1)
  };
    ...
  let zipped_iterators = outputs0.zip(
      outputs1);
  for (output0, output1) in
      zipped_iterators {
    ...
    *output0 = ...
    ...
    *output1 = ...
    ...
  }
}
```

Figure 10: Subset of generated compute function.

process of (conceptually) zipping then into a single stream, which finally can be iterated over.

The compute function must be marked as a target for the auto-vectorization (enabled with the associated attribute). Additionally, the AA Faust to Rust code generator utilizes the knowledge of how many input and output channels the computation processes, seen above in the `outputs` parameter's array size. This plays an important role in the first line of the `compute` function, enabling a fixed pattern match to deference `outputs` into its known sub-components. This is necessary to avoid failures with the Rust borrow checker in the presence of mutable references, which provides guarantees about aliases, which in turn provides invariants that feed into LLVM's auto-vectorizer. Without prior knowledge of the number of input and output channels general pattern matching can be used, but this comes at some additional overhead, including handling the case when the pattern does not match.

A small library is provided by an AA host to give access to features outside of the Wasm world, for example, Figure 6 includes the function set that enables sending messages from the audio world to the GUI. This library is intended to be kept small as any AA host must at least provide entry points, although implementations may differ. Beyond this, it is assumed that all AA modules are self contained, no support is provided for additional external libraries, e.g. The WebAssembly System Interface (WASI) [13]. This is a key enabler for providing portability to embedded, bare metal targets. To help enforce this requirement the attribute $\#![no\_std]$ is added to the generated Rust code, removing support for Rust's standard library. Crates, such as `#wee_alloc`[13] are designed

[12] `https://github.com/bluenote10/RustFaustExperiments/tree/master/Benchmarks`.

[13] `https://github.com/rustwasm/wee_alloc`.

with Wasm and embedded targets in mind and provide the ability to be used without Rust's standard library, and further more keep code size to a minimum.

4.1.1. Parameters

Parameter handling is straightforward mapping "UI" definitions in the Faust DSP code to data members in the resulting Rust code. Following other Faust targets, parameters have setters and getters indexed by an integer. As it is not possible to know up front the mapping between parameters and indexes a function is generated by the compiler mapping symbolic names (i.e. strings) to indexes. To support polyphony this functionality is extended to handle parameters unique to each voice, see the following section for details.

Faust enables the definition of parameters to be constrained to a range of values, including an initialization value and a step function. The parameter info functions, an example of which is given in the following section, return this information enabling it to be past to a host for coordination and configuration of UI components, for example.

As already noted a key design goal was to avoid, as much as possible, dependency on external crates, and to this end careful design was required to enable global statics to not depend on runtime initialisation. This meant crates such as lazy_static[14] were avoided. An additional issue with statics initialised with lazy_static is it not possible define them as externally mutable, thus an additional crate, such as mut_static[15] is required. While the extra dependency is unwanted it is also worth nothing that mut_static introduces an additional level of indirection, which, in general, we wanted to avoid.

4.2. Polyphony

As AA's API is, in general, independent of Faust, our approach to polyphony requires the DSP developer to implement support for multiple voices directly in Faust and does not support the Faust's architecture file approach, as documented in [12], for example.

A new meta variable is added to Faust, `aavoices` declares the number of required voices and Faust's `vgroup`, named $aavoice_n$, where $0 \leq n < numVoices$, is combined with `par` to specify the voices.

The AA backend for the Faust compiler generator is modified to generate unique names for per voice controls, with a prefix representing the particular voice. Two new functions are added to the generated Rust implementation:

```
pub fn get_voices(&self) -> i32;

pub fn get_param_info(
    &mut self, name: &str) -> Param;
```

The first, `get_voices`, returns the number of voices, while the second, `get_para_info`, returns information about a given parameter, including its index, used for setting and getting parameter values, and its initial value, min and max bounds, and the step amount for incrementing and decrementing.

For example, the DSP code in Figure 7, specifies that the resulting AA module supports 2 voices and each of the standard Faust MIDI parameters, i.e. *freq*, *gain*, and *gate*, which is translated into the following:

```
fn get_param_info(
    &mut self, name: &str) -> Param {
    match name {
        "freq_v0" => Param {
            index: 0,
            range: ParamRange::new(
         200.0, 50.0, 1000.0, 0.01) },
        "gain_v0" => Param {
            index: 1,
            range: ParamRange::new(
                0.5, 0.0, 1.0, 0.01) },
        "gate_v0" => Param {
            index: 2,
            range: ParamRange::new(
                0.0,0.0,0.0,0.0) },
        "freq_v1" => Param {
            index: 3,
            range: ParamRange::new(
         200.0, 50.0, 1000.0, 0.01) },
        "gain_v1" => Param {
            index: 4,
            range: ParamRange::new(
            0.5, 0.0, 1.0, 0.01) },
        "gate_v1" => Param {
            index: 5,
            range: ParamRange::new(
            0.0,0.0,0.0,0.0) },
        _ => Param {
            index: -1,
            range: ParamRange::new(
            0.0, 0.0, 0.0, 0.0)}
    }
}
```

As is standard with Faust control parameters that are global, i.e. shared for all voices, can be defined using OSC notation to avoid being defined on a per voice basis. For example, the above envelope could be defined with controls for attach, decay, sustain, and release as follows:

```
voice = hgroup("midi", osc(freq))
with {
  ...
  a = hslider("/v:envelope/Attack",
              0.001, 0.001, 4, 0.001);
  d = hslider("/v:envelope/Decay",
              0.0, 0.0, 4, 0.001);
  s = hslider("/v:envelope/Sustain",
              1.0, 0.0, 1.0, 0.01);
  r = hslider("/v:envelope/Release",
              0.0, 0.0, 4.0, 0.01);
  envelope = en.adsr(a,d,s,r,gate)*gain;
  ...
}
```

A downside to the approach taken by AA is in the case when there is a single voice, i.e. monophonic. Currently the modified Faust compiler requires that user specifies that there is a single voice and wraps Faust's `par` construct. For example, the following code is taken directly from AA's VL-1 emulation:

[14] https://github.com/rust-lang-nursery/lazy-static.rs

[15] https://github.com/tyleo/mut_static

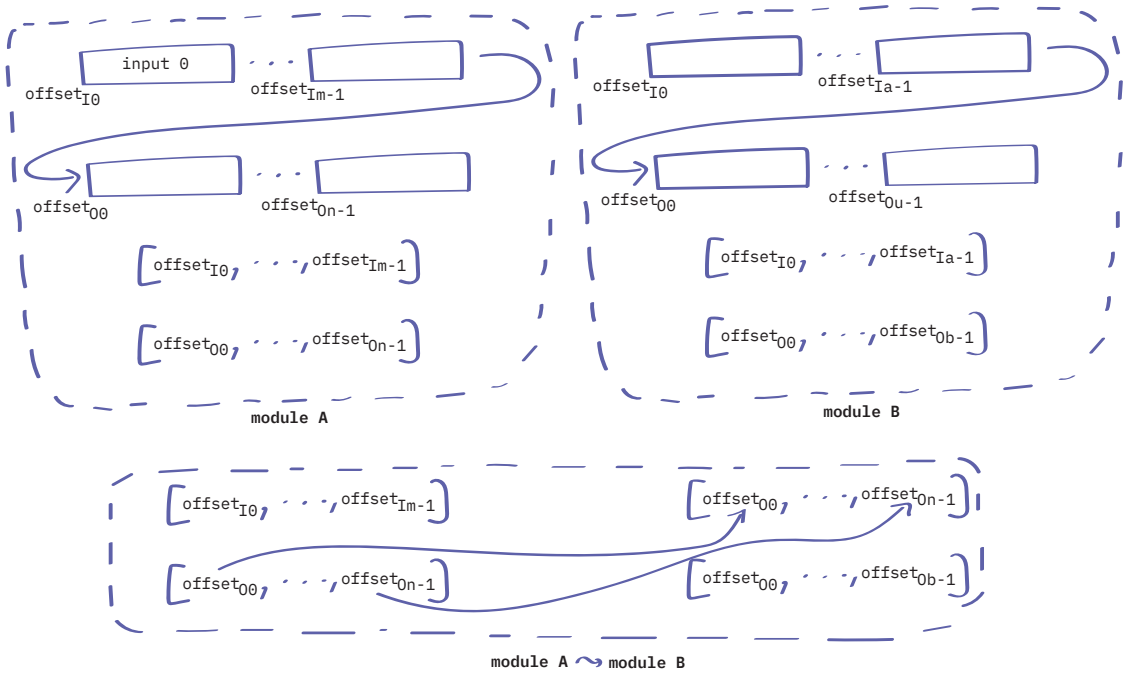

Figure 11: Composing AA modules A and B into an audio graph

```
process = vgroup(
    "voices",
    par(n, 1,
        vgroup("aavoice%n", waveforms))
  ) :> _ ;
```

Of course, the Faust compiler can optimize away any noise introduced by the use of the `par` construct, but it still means porting Faust mono synths requires work to fit with the AA model. It would not be a huge amount of work to modify the Faust compiler to avoid requiring this, although, to date we have not done so.

4.3. Composing Modules

It is possible to compose AA modules at the buffer level, i.e. the output from one module can be passed to another module for consumption, similar to how data is passed between plugins within DAWs, for example[16].

For AA modules to be composed together one module's output buffer(s) must become the input buffer(s) of another's, as highlighted in Figure 11. It is not possible for a particular module to know how it will be connected a priori, to other modules, and thus the hosting application must configure the audio graph, connecting module outputs to module inputs, via the set input and set output functions that all AA modules support, as can be seen at the bottom of Figure 6. To compose two modules the host application implements the following algorithm:

```
if mod1.get_outputs() == mod2.get_inputs()
    for i in 0..mod1.get_inputs()
        output = mod1.get_output(i)
        mod2.set_input(i, output)
    endfor
endif
```

While there is a cost of configuring a composed audio graph, this is done once when the graph is configured. Furthermore, the cost of copying external audio buffers in and the output buffers out of Wasm's linear memory is amortized more and more as the graph grows.

The resulting functionality of module composition is similar to that provided in DAWs, such as Ableton or Reaper, however, due to the isolation of Wasm's linear memory returning back to the host between each module computing audio can lead to unwanted memory copies, which in turn introduces overall latency into the audio pipeline.

It is worth noting that module composition is independent of how different UI elements are composed. For example, the AA Muses synth[17] has two audio modules, one for the 6 voice synthesis engine, and another for its reverb effect, but there is only a single UI.

4.4. Hosting AA modules on the Desktop

As part of Audio Anywhere initial research and development phase, we developed a standard alone desktop application that runs on Mac OS, Windows, and Linux (tested on Raspberry Pi), along with a VST 2.x plugin, currently only supported on Mac OS. Both the

[16] Per sample composition is not currently possible as it would require merging at the level of the compute functions, but as Wasm is compiled in place, it is reasonable to believe it is possible.

[17] `https://github.com/bgaster/aa_examples/tree/master/muses`.

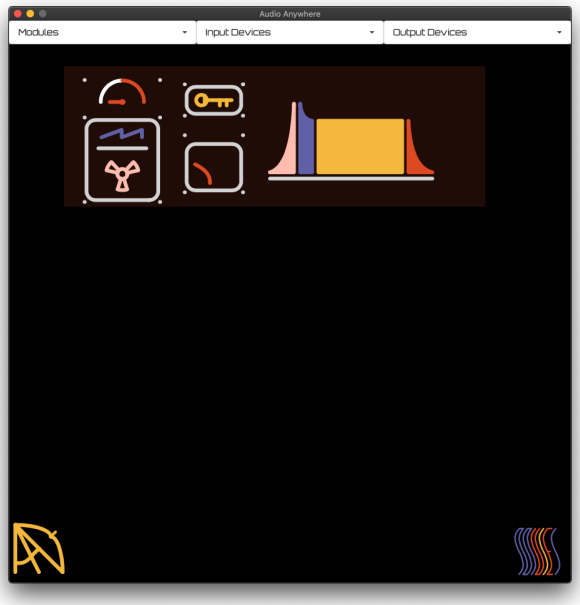

Figure 12: Standalone AA Hosting Application, with Muses Nuke synth loaded.

desktop application and plugin support for hot loading AA bundles from a server and modules can be swapped in and out live. Additionally, the desktop application supports chaining of modules to build more complex instruments. Figure 12, shows a screenshot of the desktop application running on macOS, with the Muses Nuke synth loaded.

The Wasm modules are Just In Time (JIT) compiled ahead-of-time when loaded, and currently our backend supports two WebAssembly virtual machines, Wasmtime[18], and Wasmer[19]. We initially began with supporting only Wasmer as the WebAssembly runtime of choice, however, we discovered that as the project evolved, in particular, as the Faust compiler was modified to support Rust code generation that could be automatically vectorized by LLVM and compiled to WebAssembly with SIMD-128 support, Wasmer failed to load the resulting WebAssembly. At the time of writing Wasmtime does not support the Raspberry Pi, but Wasmer does provide support for Raspberry Pi 4 and for this platform we disabled SIMD-128, due to stability issues.

In practice, we found Wasmer to be slightly faster than Wasmtime for small functions, but this changed as more Wasm code was executed per call. This is analysed further in Section 5.

4.5. Hosting AA modules on Daisy

Electrosmith's Daisy[20] is an embedded platform for creating high fidelity, 192kHz, 24-bit stereo audio. At its heart is a STM32 ARM Cortex-H7 running at 480Mhz, features including a 32-bit floating processing unit optimized for DSP, and 64MB of external SDRAM. An image of the Daisy is given in Figure 13.

[18] https://github.com/bytecodealliance/wasmtime.
[19] https://wasmer.io/.
[20] https://www.electro-smith.com/daisy.

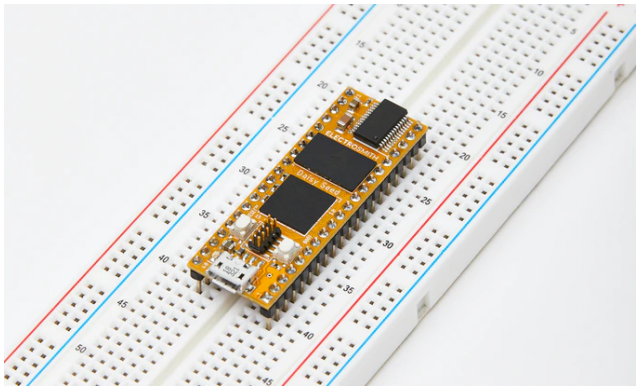

Figure 13: Electrosmith's Daisy

```
// wasm audio buffer
static float * wasm_out_buffer;
// other globals go here

static void AudioCallback(float *in, float
    *out, size_t size) {
    hw.DebounceControls();

    // handle VL-1 keyboard and button
        events
    // Wasm compute function
    aa_compute(size);

    for(size_t i = 0; i < size; i += 2) {
        float sig = wasm_out_buffer[i];
        out[i]     = sig;
        out[i + 1] = sig;
    }
}

void initSynth(float samplerate) {
    // setup Wasm runtime
    init();
    // initialize wasm AA module
    aa_init((double)samplerate);
}

int main(void) {
    float samplerate;
    hw.Init();
    samplerate = hw.AudioSampleRate();
    initSynth(samplerate);

    // start callbacks
    hw.StartAdc();
    hw.StartAudio(AudioCallback);

    while(1) {}
}
```

Figure 14: Daisy Audio Application

The Daisy can be programmed via the Arduino IDE, however, the recommended path is to use the ARM Cross compiler for the Cortex-M series, along with OpenOCD[21] for connecting, programming the flash, and debugging. `libDaisy` [14] provides a hardware abstraction library for the Daisy, which provides easy access to GPIOs, MIDI, and USB communication and can be used along side `DasiySP` [15], providing additional DSP functionality. The listing in Figure 14 shows basic setup for a Daisy AA application, in this case for the VL-1 demo, including place holders for connecting to the Wasm world.

To evaluate the feasibility of Wasm on the Daisy we developed two prototype approaches. The first utilizes Wasm3[22], which is a high performance WebAssembly interpreter written in C. The main benefit of Wasm3 is that it requires very little memory, in the order of 64k for code and 10kb for RAM, a key design factor when considering an embedded platform such as Daisy. The biggest downside of Wasm3 is one of performance, in this case Wasm3 is 4-5x slower than optimized Wasm JIT engines, and given that that the requirements for real-time audio are high this is considered a major drawback. That being said it was remarkably easy to get going and our VL-1 emulation was able to run in real-time. However, other, more compute intensive, AA modules clipped, even at 44.1kHz, when using Wasm3 on the Daisy.

The second approach uses `wasm2c` from The WebAssembly Binary Toolkit[23]. `wasm2c` converts Wasm binaries to portable C code. The downside of `wasm2c` is that AA modules are converted to C and then compiled to ARM Cortex-M machine code offline, which goes against one of our key design goals of compile once, run anywhere. At this time we don't have an alternative path that offers as good performance, along with the portability and simplicity of Wasm3. To address this we have begun development of a Cotex-M7 JIT compiler for AA Wasm modules.

In the case of both implementations a major factor was how to manage Wasm's linear memory. Wasm allocates memory in 64k chunks, and for AA modules memory is populated with DSP state, along with input and output audio buffers, but additionally the Wasm runtimes require memory to allocate a modules function tables and other internal resources. In general, we found that Daisy's on-chip memory was not enough. Luckily, the Daisy comes with 64MB of external SDRAM, which while intended for audio samples, is an excellent resource for managing Wasm's linear memory, plus the additional resources needed by the runtime. The cost of a full memory allocator was determined to be overkill for AA modules that, in general, are assumed to allocate memory statically, thus, avoiding dynamic memory allocation in the audio task. As such a simple and very compact Slab[24] based allocator was implemented, that supports allocating pages of 64k, with independent pages allocated to the AA Wasm runtime and Wasm's linear memory. In particular, pages allocated to Wasm's linear memory are guaranteed to be accessed directly by AA Wasm module code or by memory that is "mapped" into the hosting applications address space.

5. EVALUATION

In this section we present some early performance results for Audio Anywhere. It is not the intention here to provide a deep dive into the performance characteristics of AA modules and applications, rather some preliminary results are presented, providing some early evidence that AA's use of Wasm as a compile once, run anywhere target is practical.

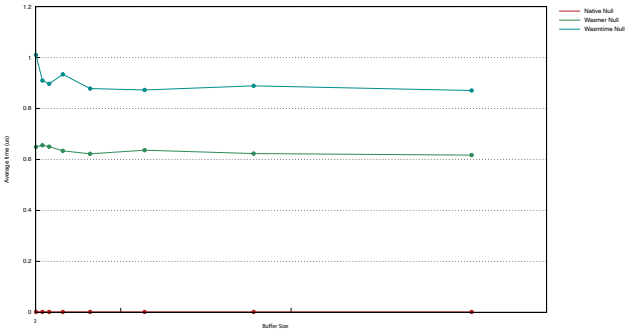

Figure 15: Benchmark for null function.

The benchmark results presented in this section were performed on a 2018 Mac Book Pro, with 2.9 GHz 6-Core Intel Core i9, 32 GB 2400 MHz DDR4, running macOS Catalina, version 10.15.6. C++ was compiled with Apple clang version 11.0.3 (clang-1103.0.32.62), compiled with -O3, and for Rust source rustc 1.47.0-nightly (6c8927b0c 2020-07-26, compiled with –release.

On the whole the remainder of this section focuses on Rust compiled to Wasm compared against the same Rust code compiled to native x86-64. However, we first briefly consider the performance of our modified Faust to Rust compiler. The following table compares a simple DSP algorithm that copies a single channel input to a single channel output:

C++	Rust	Rust Optimized
84860.747 MB/sec	3379.915 MB/sec	88353.927 MB/sec

The left hand column shows Faust code compiled via C++, while the middle is the unmodified Faust to Rust compiler, and finally the right hand column is the modified Faust to Rust compiler, which supports the optimizations described in Section 4. The original Faust to Rust compiler is more than 20x slower than both the C++ and the modified Faust to Rust compiler, with the now optimized Rust code slightly faster than the C++ path.

For the Rust and AA Wasm benchmarks each benchmark was run and results gathered with Criterion.rs[25], a statistics-driven Microbenchmarking framework for Rust, with block sizes of 1, 16, 32, 64, 128, 256, 512, and 1024.

Figure 15, compares the performance of calling an empty function, i.e the cost of a function call. For the native code this is very small and while more the cost of calling into Wasm for both the Wasmtime and Wasmer runtimes is still low, although considerable slower in comparison.

Figure 16, compares a sine wave oscillator. Here again we have included both Wasmtime and Wasmer runtimes, however, due to a bug in Wasmer it failed to load the WebAssembly with 128-SIMD generated code. As can be seen as the number of generated audio samples increases per frame the Wasm code running under Wasmer performance degrades considerably. Given that this is not the case for the Wasmtime version, it seems likely that the lack of 128-SIMD is having a large impact on the performance of the Was-

[21] `http://openocd.org/`.
[22] `https://github.com/wasm3/wasm3`.
[23] `https://github.com/WebAssembly/wabt`.
[24] `https://en.wikipedia.org/wiki/Slab_allocation`.
[25] `https://github.com/bheisler/criterion.rs`.

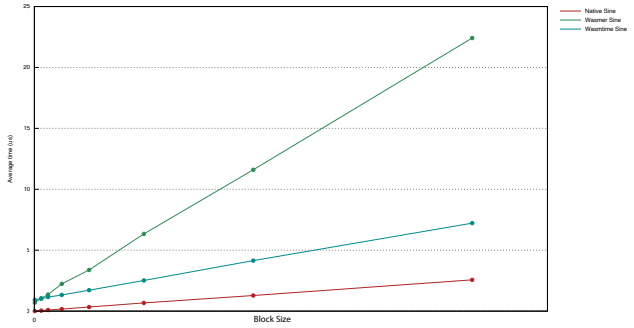

Figure 16: Benchmark for sine function.

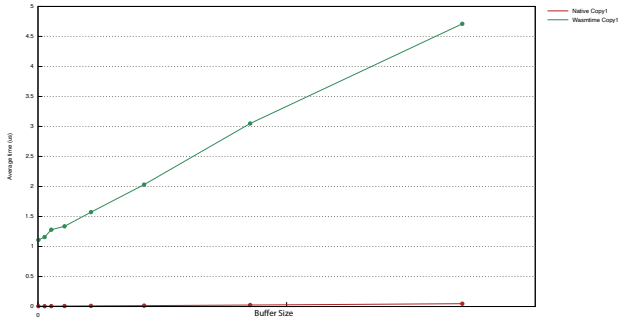

Figure 17: Benchmark for copy1 function.

mer version. The performance of the AA Wasm module running under the Wasmtime runtime is very promising and is within a 4x slowdown. As the number of audio samples computed increases the performance gap is fairly consistent.

Finally, Figure 17 reproduces the copy in and out benchmark presented above in the context of C++ and Rust, but this time for Rust and AA Wasm. The results do not include Wasmer as again without 128-SIMD support it did not compare strongly. Wasmtime again fares well compared to the native code, however, its performance seems to slow down linearly in comparison. It likely that again the overhead of calling into Wasm is playing a role, but it does not seem to explain all of the performance differences. This needs further investigation and remains an important area of future work.

6. CONCLUSION

We are developing a general framework for audio modules that can be compiled once and run anywhere. The project itself it part of a larger ideal, that maybe we could dream of compiling large portions of software just once and run close to native speeds, one day surpassing Python and other high-level languages. This early work shows the promise that maybe we are closer to that goal today than ever before.

A key design goal of Audio Anywhere is for its API to follow a C ABI. Unlike Faust this was deemed necessary as we required a simple binary format specified with WebAssembly. While we believe this comes with some benefits (it can be run almost anywhere) it does also come with some limitations and drawbacks. In particular, this meant at times it was necessary to step outside common approaches taken by Faust for portability. In particular, our approach to handling polyphony is different to that commonly used in Faust when combined with architecture files. Moreover, GUIs are not directly derived from the Faust source code, and instead Faust's UI elements are utilised to provide mechanisms to connect an externally developed UI with Faust's internally generated parameter indexes. We do not propose this as an alternative to Faust's, but rather note that it can be interesting to consider different approaches when your target is limited in certain ways, such as is the case with WebAssembly.

A key advantage of our approach is the need to only compile once, that's it! In this case you compile the Faust DSP code to Rust, using our modified Faust compiler, and then compile the Rust code into a WebAssembly library. Additionally, if required, a GUI can be developed using HTML5 and distributed alongside the WebAssembly audio unit. This works well for the desktop and we have found that given a 64-bit hosting OS, required for the most current Wasm Virtual runtimes, both the portability and performance are very usable. There is still work to be done for embedded, the virtual machines are interpreters at the moment and there is a real trade off between space and performance, particularly in the case of including a real-time JIT compiler for Wasm. For the Daisy we found that while interpreters work, the best trade off between space and performance was to utilize `wasm2c`, compiling to a very neutral C and providing a library that provides the Wasm runtime necessary to load the module. However, this goes against the compile once, run anywhere philosophy and to address this we have begun development of a tiny JIT that will load and run an AA Wasm module, it is being designed specifically for AA modules, targeting ARM Cortex family.

Early performance results show that the approach proposed by Audio Anywhere is promising, but there is a lot more work to be done to validate it more robustly. For one we plan to develop a more extensive benchmark suite, running on a variety of different platforms. Secondly, the cost of calling a Wasm function from the host seems expensive and this needs to be better understood. Finally, more micro and macro tests need to be developed to demonstrate how the performance scales to different scenarios.

7. ACKNOWLEDGMENTS

We would like to acknowledge the consistent support and insight from colleagues in the Computer Science Research Centre and Creative Technology Lab. Special thanks to Harri Renney and Nathan Renney. This work was supported in part by a UWE internship grant.

8. REFERENCES

[1] Paul Hudak, "Building domain-specific embedded languages," *ACM Comput. Surv.*, vol. 28, no. 4es, pp. 196–es, Dec. 1996.

[2] Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen, *OpenGL Shading Language*, Addison-Wesley Professional, 3rd edition, 2009.

[3] Yann Orlarey, Stéphane Letz, and Dominique Fober, *New Computational Paradigms for Computer Music*, chapter

"Faust: an Efficient Functional Approach to DSP Programming", Delatour, Paris, France, 2009.

[4] Mike Bailey, "Introduction to the vulkan graphics api," in *ACM SIGGRAPH 2018 Courses*, New York, NY, USA, 2018, SIGGRAPH '18, Association for Computing Machinery.

[5] John Kessenich, Boaz Ouriel, and Raun Krisch, "SPIR-V Specification.," 2020 (accessed August 9, 2020). `https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html`.

[6] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017, pp. 185–200, Association for Computing Machinery.

[7] Alon Zakai, "Emscripten: An llvm-to-javascript compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, New York, NY, USA, 2011, OOPSLA '11, p. 301–312, Association for Computing Machinery.

[8] Hongchan Choi and Jonathan Berger, "WAAX: Web Audio API eXtension," in *NIME'13*, p. 4.

[9] Charles Roberts, Graham Wakefield, Matthew Wright, and JoAnn Kuchera-Morin, "Designing Musical Instruments for the Browser," vol. 39, no. 1, pp. 27–40.

[10] Jari Kleimola and Owen Campbell, "Native Web Audio API Plugins," in *Proceedings of the 4th Web Audio Conference (WAC-2018)*.

[11] Stéphane Letz, Yann Orlarey, and Dominique Fober, "FAUST Domain Specific Audio DSP Language Compiled to WebAssembly," in *Companion Proceedings of the The Web Conference 2018*. WWW '18, pp. 701–709, International World Wide Web Conferences Steering Committee.

[12] Stéphane Letz, Yann Orlarey, Romain Michon, and Dominique Fober, "Polyphony, sample-accurate control and MIDI support for FAUST DSP using combinable architecture files," May 18 – 21, 2017, p. 7.

[13] Subgroup of the WebAssembly CG., *WASI: The WebAssembly System Interface*, 2020 (accessed August 8, 2020). `https://wasi.dev/`.

[14] Electrosmity, *libDaisy*, 2020 (accessed August 14, 2020) `https://github.com/electro-smith/libDaisy/blob/master/doc/libdaisy_reference.pdf`.

[15] Electrosmity, *DaisySP*, 2020 (accessed August 14, 2020). `https://github.com/electro-smith/libDaisy/blob/master/doc/libdaisy_reference.pdf`.