Dynamic Per-Sample Processing with WebAssembly

Charles Roberts
Worcester Polytechnic Institute
charlie@charlie-roberts.com

ABSTRACT

While various audio libraries for the web have been compiled to WebAssembly from other languages, few have been written directly in WebAssembly itself. Writing DSP algorithms directly in WebAssembly enables precise control and opportunities for optimization that are perhaps difficult to achieve when a using a higher-level language coupled with a compiler; conversely, higher-level languages are often optimized for abstraction, readability, and speed of development. Despite the advantages higher-level languages provide, we hypothesized that writing a low-level signal processing library directly in WebAssembly is both appropriate to the capabilities of the language while also providing for finer control over optimization.

Accordingly, we ported a low-level library we had previously developed in JavaScript, *genish.js*, to WebAssembly. In our initial investigation we focused on entirely dynamic audio graphs with per-sample processing; our prior work also used per-sample processing but required recompilation of the audio graph after any significant changes were made. While our dynamic WebAssembly library possesses a number of notable advantages over our prior work, we ultimately decided that it is too computationally inefficient with larger audio graphs to be used for constructing higher-level libraries and tools for music creation. Despite this, we do feel it is appropriate for specific uses, such as live coding and enabling end-user signal processing without requiring compilation. To address the performance limitations of our engine, we built a WebAssembly compiler that outputs optimized representations of larger, interconnected audio graphs. Testing shows that compiling and optimizing audio graphs created using this library yields highly performant unit generators. We believe the combination of a fully dynamic graph with precompiled higher-level signal processing functions will work well for the future construction of music creation tools.

1. INTRODUCTION

Over the last four years, audio developers have rapidly adopted WebAssembly (WASM) as a target language for projects that run in the browser. In many cases WebAssembly can be more computationally efficient than JavaScript, but perhaps more importantly for audio developers is its use of ahead-of-time compilation and low-level control over memory usage.

In our prior research on developing audio synthesis libraries for the browser, we created a library, *genish.js* [11, 12], that enables developers to write synthesis algorithms using a standardized set of operations that are then compiled to optimized JavaScript functions. We hypothesized that by using WebAsssembly instead of compiling to JavaScript we could obtain performance gains while also taking advantage of ahead-of-time compilation, which would help avoid potential audio dropouts during realtime signal processing. We were also particularly interested in writing the WebAssembly ourselves (by "hand"), both to better understand the underlying language features, and to explore WebAssembly optimizations that can go missing in the compilers of popular higher-level languages like Rust or C++.

In this paper we provide a short background of audio libraries written for the browser that take advantage of WebAssembly, describe our own experiments authoring WASM bytecodes and compiled WebAssembly libraries, and then discuss further optimizations needed before using the WASM version to author high-level music creation tools.

2. BACKGROUND

WebAssembly is a binary format that has been adopted in most modern browsers with the goal of providing a portable, secure, and efficient runtime [5]. For browser-based musicians and audio developers, WebAssembly is important as it can be used inside of *AudioWorkletProcessor* nodes, which run in their own dedicated thread and are optimized for realtime performance [3]. By using WebAssembly, developers can avoid problems commonly associated with using JavaScript for realtime digital signal processing, such as non-deterministic garbage collection and the performance of browsers' just-in-time (JIT) compilation engines. In addition to the binary format used by browsers, there is also a human-readable WebAssemblyText (WAT) format that can be freely converted to binary; WASM binaries can also be easily converted into WAT.

Audio developers have been exploring the potential of WebAssembly coupled with AudioWorklets for a few years now. Csound had initially begun moving to the web by compiling to JavaScript [9] but now can also compile to WebAssembly via Emscripten [15]. Maximillian [4] is a C++ library that has since been compiled to WebAssembly

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

for use in the Sema live coding system [2]. Significant portions of SuperCollider have also been ported to WASM[1]. Glicol, a more recent computer music language, was authored in Rust with the intention of using Emscripten to compile to WASM for use in the browser [8]. Elementary is a JavaScript library that uses the JUCE C++ library to compile to WASM [13]. It features a functional, composable API that is similar in some ways to the API found in genish.

All of these languages and environments use Emscripten to compile the final WASM blob that browsers load for use. However, Battagline notes that some WASM optimizations can be performed by directly editing WebAssemblyText files that elude compilers and optimizers [1]. As stated in the introduction, part of our motivation for refactoring genish.js in WebAssemblyText was to take advantage of such opportunities for optimization.

A notable precursor to this work is the FAUST compiler, which can compile to WAT or directly to WebAssembly bytecode, in addition to a number of other programming languages via an intermediate representation. The FAUST research team has also compiled its compiler to WebAssembly, so that it can be used directly within web pages [10]. Genish.js uses a simpler model than FAUST, with no intermediate representation used. Although we have considered targeting other languages in similar fashion to FAUST, we are encouraged by a variety of WASM runtimes—such as Wasmer[2] and Wasmtime[3]—that run WebAssembly embedded in other languages. These runtimes mean that there is the potential to run genish across a variety of platforms and programming environments; while there is a performance penalty for WebAssembly compared to natively compiled applications, for many smaller functions WebAssembly can perform within 10% of native code [7]. Signal processing algorithms are an excellent use case for WebAssembly as there are typically no calls outside of the WebAssembly virtual machine while processing a block of samples; such calls are a known performance bottleneck [6].

3. DYNAMIC IMPLEMENTATION

Genish was originally authored to be a loose port of the Gen extension for Max/MSP [14]; many of the operators in genish borrow their names from similar operators found in Gen. Although operators in genish are low-level, they can be combined into fairly complex virtual analog filters, anti-aliased oscillators, and other more substantial DSP algorithms. Many such algorithms have been implemented in gibberish.js, a higher-level audio library that builds on top of genish [11]. As a quick taste of the API, Listing 1 displays code to generate a frequency-modulated sine oscillator.

```
cycle(
  add(
    440,
    mul(
      cycle(2),
      30
    )
  )
)
```

Listing 1: Basic FM in genish

[1]https://scsynth.org/t/webassembly-support/3037
[2]https://wasmer.io/
[3]https://wasmtime.dev

One problem with the previous version of genish is that it compiles JavaScript functions at runtime which must then be further optimized by the browser's JIT compiler; this additional optimization step can lead to unpredictable performance and potential audio dropouts. A motivation for porting the library to WebAssembly is to take advantage of the ahead-of-time compilation that WebAssembly provides, removing the potential for audio dropouts in realtime performance that can occur while JIT compilation is taking place. Although the backend for the new version of genish is rewritten from scratch, the API has remained virtually identical between versions.

Our new version of genish requires no compilation of end-user audio graphs; instead, all graphs are run using a custom WASM bytecode where each audio node consists of a block of memory, a pointer to a WASM function to process the memory, and a small amount of JavaScript code to make it easy for developers using the library to link nodes together and manipulate their associated memory. The library primarily consists of two files. The first is a WAT file, which contains the signal processing routines for all the various genish nodes. After compiling the WAT to WASM and optimizing, the resulting WASM file is ∼10 KB in size. The second file is a JavaScript file that performs memory allocation for newly constructed nodes; it also contains utility functions for loading audio files, creating useful wavetables, and starting up AudioWorkletProcessor nodes running genish. It is 20 KB in size, which means that, taken together, only 30 KB (uncompressed) are needed to download and begin using the new version of genish, while the original version is 485 KB in size.

The WASM and end-user JavaScript representations of signal processing nodes are linked via memory stored in a `SharedArrayBuffer`; this memory can be accessed from both the main thread and the audio thread.

3.1 WebAssemblyText

The WAT portion of the library consists of one file containing approximately 175 functions. Each function accepts the memory location of data it should process as its sole argument. This data can contain *static* parameters that are not expected to change at sample rate as well as *dynamic* parameters, which are usually the outputs of other genish functions computed on a per-sample basis. For each operator in genish, functions are defined for every combination of static or dynamic parameters that might be required. For example, the `add` operator, which requires two values to compute a result, has four underlying versions that cover every combination of adding static and dynamic numbers:

- add_static_static
- add_static_dynamic
- add_dynamic_static
- add_dynamic_dynamic

While requiring multiple versions of functions makes it cumbersome to author operators requiring a large number of potentially dynamic parameters, it saves the WASM engine from having to check to see whether or not each number used needs to be dynamically calculated on a per-sample basis; in our initial version of the library we performed these checks at

Offset	Purpose
0	Function index
4	Per-sample increment (node location or number)
8	Reset trigger (node location or number)
12	Minimum value (number)
16	Maximum value (number)
20	Phase (number)

Table 1: Memory layout for the accum operator.

runtime and found them to incur a significant computational cost.

Both static and dynamic parameters reference a single memory location. For static parameters this location contains the desired value. For dynamic parameters, a pointer to another address in memory is provided where all information needed to compute a dynamic parameter is stored, including the index of the processing function associated with the node (all exported functions in WebAssembly are stored in a table and referred to by index) and any other data required for computation. With this information dynamic parameters can then be calculated by calling WebAssembly's `call_indirect` function, which calls a function via its table index (similar to a function pointer in C).

To take one commonly-used operator as an example, Table 1 shows the memory layout for an `accum` operator. Each accumulator has an increment and a reset parameter; these can either be static numbers or the output of other genish nodes. `Accum` also uses memory to store its range and its current value; these can only be represented with 32-bit floats[4].

While the various genish operators form the bulk of the WASM used, there are two additional important functions: `render` and `renderStereo`. These accept a graph of operators (or pair of graphs for `renderStereo` and call the function at the head of the graph to fill a block of samples of a user-defined size. They also increment a global `clock` variable that is used for various purposes in genish, such as memoization.

3.2 JavaScript

As previously noted, the JavaScript required for the library is minimal. It is responsible for the following:

1. Load WASM and initialize an AudioWorkletProcessor node.
2. Initialize memory for each instantiated node in a SharedArrayBuffer that is also accessed by the WASM bytecode.
3. Provide meta-programming enabling a reasonable end-user API for manipulating nodes via JavaScript. This typically consists of using property setters to directly assign values to memory locations in the SharedArrayBuffer, but there are additional abstractions that are more complex (described below).
4. Various utility functions to load samples, generate useful wavetables etc.

[4]The `counter` ugen can accept other operators for its min/-max values, but incurs a greater overall computational cost compared to `accum`.

The meta-programming is perhaps the most interesting aspect of the JavaScript portion of the library. In order to manipulate parameters of genish nodes, we have to keep in mind that if a user changes a parameter from a static number to a dynamic value (for example, assigning a phasor to modulate the frequency of a sine oscillator) then we need to change the function we call to its dynamic version. In our current bytecode, all this requires is incrementing the function pointer associated with the node in the shared memory buffer; this is all handled behind the scenes via meta-programming. This illustrates that it is perhaps better to think of a given node as a chunk of memory instead of associating it with a particular function, because the function that is applied to a node's memory is freely changeable, potentially even at sample rate [5]. For complex nodes with multiple parameters, a bit-mask is used to keep track of which parameters are dynamic. When the mask changes, the function pointer associated with the operator is similarly updated.

The commented section of the genish WAT file in Listing 2 includes both the static and the dynamic version of `round` for comparison.

4. EVALUATION, ITERATION, AND DISCUSSION

We ran a rendering test comparing our new engine to the prior version of genish.js as well as the native nodes of the Web Audio API. In our test, we rendered 100 sine oscillators for a minute using `OfflineAudioContexts` running in Chrome, version 100. Our test computer was a 2018 Macbook Pro with a 2.6 Ghz i7 Processor, running Big Sur as its operating system. In this test, the Web Audio API took a mean time of 1614 ms over 50 runs, while the prior version of genish.js took 1678 ms. Our new WASM library took 3765 ms; the Web Audio API performs about 2.33x more efficiently.

These results are interesting and show the relative strengths and weaknesses of the various approaches. For example, it is perhaps suprising that the original version of genish.js, where all DSP was written in JavaScript, can compete in this test with the native C++ nodes that make up the Web Audio API. But it's important to note that this earlier version of genish compiles a single flat function to represent the entire audio graph; any significant changes to the graph require this function to be recompiled. The Web Audio API is using a dynamic graph that can be freely modified. Then again, genish.js is performing per-sample processing, which enables a variety of techniques the Web Audio API is not capable of using its built-in nodes; enabling the exploration of per-sample processing techniques were one of the primary motivations for genish.js.

In the new version of genish.js, we have a completely dynamic graph capable of per-sample processing, but incur a significant performance penalty by dynamically resolving function pointers at runtime. However, when compared with the prior version of genish.js, we also get the benefits of ahead-of-time compilation, which helps ensure audio dropouts will not occur during JIT compilation. Our prior

[5]While the function applied to a chunk of data can be changed at will, it will typically be important to ensure that a given function expects the same memory layout as the data it will operate on.

```
;; accepts one argument, a location in memory, and returns a Float32
(func $round_s (export "round_s") (param $loc i32) (result f32)
   local.get $loc  ;; read the location of the memory for this operator
   i32.const 4     ;; the value the operator affects has an offset of 4 bytes
   i32.add         ;; add the offset to the memory location
   f32.load        ;; load the value from the location on the stack

   f32.nearest     ;; round the value, last value is returned automatically
 )

 (func $round_d (export "round_d") (param $round_data_loc i32) (result f32)
   ;; declare variable to hold location of data assocated with input parameter
   (local $parameter_data_location i32)
   ;; load data location, which is offset by four bytes from operator location
   (i32.load (i32.add (local.get $round_data_loc) (i32.const 4) ) )
   ;; store data location in variable
   local.set $parameter_data_location
   ;; call a function accepting an integer and returning a float
   (call_indirect (type $sig-i32--f32)
     ;; pass data location
     (local.get $parameter_data_location)
     ;; this loads the function pointer to call, stored as a
     ;; 32-bit integer index
     (i32.load (local.get $parameter_data_location) )
   )

   f32.nearest  ;; round the result and return
 )
```

Listing 2: The WebAssembly Text code for the round function in genish.js

version relied heavily on JIT compilation, and while this was suitable for many applications where the graph was known ahead of time and only needed to be compiled once, it made it less suitable for applications like live coding, where the audio graph is constantly in flux. The JIT compiler was also unable to compile very large audio graphs; for example, on the same 2018 Macbook Pro used in the other tests discussed in this section, the prior version of genish.js was only capable of rendering about 200 sine oscillators concurrently. This wasn't due to computational efficiency (those 200 oscillators only took 8% of one core), but rather solely due to the size (number of lines) of the generated function, which the JIT was unable to optimize. The new version of the library can handle 1500 oscillators on the testing computer while using about 85% of a single core.

Initially, we were satisfied with these results and began making plans to use the new version of genish as the basis for gibberish.js, our higher-level library for musical synthesis. However, we discovered that as the complexity of our graphs increased, performance decreased much quicker than we expected. The problem is that the nodes in genish are very low-level; a complex instrument might be a graph containing dozens of operators. Calling any function incurs a certain amount of overhead, but calling a function using a function pointer (such as the tables used in WebAssembly) introduces an additional cost. With relatively flat graphs (such as hundreds of sine oscillators feeding a single output) these costs are hidden, but with more complex graphs the costs of the dynamism genish provides quickly become more apparent.

After considering various approaches to solving this problem, we decided on a hybrid approach. We began extending genish so that instruments with complex graphs can be compiled ahead-of-time into single WebAssembly functions; meanwhile, we plan to keep the current dynamic engine for executing user-defined audio graphs and live coding modulations. While this work is still underway, many operators have already been extended to support this, and we are excited by our preliminary results. For example, a precompiled WebAssembly function running 3000 sine oscillators creating using the newest version of genish.js takes around 40% of a single core; in effect, we can run twice as many oscillators for half the processing power as the dynamic engine. We're confident this hybrid approach will support authoring higher-level libraries for music and look forward to using genish in our future work in this way. In the meantime, the WASM version of genish.js provides the advantage of completely dynamic audio graphs and a very small download.

Genish.js is open-source software. The main repository can be found at https://github.com/charlieroberts/genish.js, with a coding playground and accompanying demos at https://charlieroberts.github.io/genish.js/playground. For those interested in exploring the dynamic WASM engine described in this paper, a separate experimental playground can be found at https://gibber.cc/genish.

5. REFERENCES

[1] R. Battagline. *The Art of WebAssembly*. No Starch Press, 2021.

[2] F. Bernardo, C. Kiefer, and T. Magnusson. An AudioWorklet-based Signal Engine for a Live Coding Language Ecosystem. In *Web Audio Conference (WAC 2019)*, pages 77–82, 2019.

[3] H. Choi. AudioWorklet: The Future of Web Audio. In *International Computer Music Conference*, 2018.

[4] M. Grierson and C. Kiefer. Maximillian: An Easy to Use, Cross Platform C++Toolkit for Interactive Audio and Synthesis Applications. In *Proceedings of the*

International Computer Music Conference. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 2011.

[5] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[6] D. Hockley and C. Williamson. Benchmarking Runtime Scripting Performance in Wasmer. In *Companion of the 2022 International Conference on Performance Engineering*, 2022.

[7] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, 2019.

[8] Q. Lan and A. R. Jensenius. Browser-based Collaborative Live Coding with Glicol: A Graph-oriented Live Coding Language Written in Rust. In *Proceedings of the Web Audio Conference*, 2021.

[9] V. Lazzarini, E. Costello, S. Yi, et al. Csound on the Web. In *Proceedings of the 2014 Linux Audio Conference*, pages 77–84. University of Bath, 2014.

[10] S. Letz, Y. Orlarey, and D. Fober. FAUST Domain Specific Audio DSP Language Compiled to WebAssembly. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, pages 701–709, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

[11] C. Roberts. Strategies for Per-Sample Processing of Audio Graphs in the Browser. In *Proceedings of the Web Audio Conference*, 2017.

[12] C. Roberts. Metaprogramming Strategies for AudioWorklets. In *Proceedings of the Web Audio Conference*, 2018.

[13] N. Thompson. Elementary Audio. http://precog.iiitd.edu.in/people/anupama. Last accessed 6.25.2022.

[14] G. Wakefield. *Real-Time Meta-Programming for Interactive Computational Arts*. PhD thesis, University of California Santa Barbara, 2012.

[15] S. Yi, V. Lazzarini, and E. Costello. WebAssembly Audioworklet Csound. In *Proceedings of the Web Audio Conference*. TU Berlin, 2018.