An AudioWorklet-based Signal Engine for a Live Coding Language Ecosystem

Francisco Bernardo
Emute Lab, Dept. Music
University of Sussex
F.Bernardo@sussex.ac.uk

Chris Kiefer
Emute Lab, Dept. Music
University of Sussex
C.Kiefer@sussex.ac.uk

Thor Magnusson
Emute Lab, Dept. Music
University of Sussex
T.Magnusson@sussex.ac.uk

ABSTRACT

This paper reports on early advances in the design of an ecosystem for creating new live coding languages, optimal for audio synthesis, machine learning and machine listening. We present the design rationale and challenges when applying the Web Audio API, and in particular, an *AudioWorklet*-based solution to refactoring our digital signal processing library Maximilian.js for our high-performance signal synthesis engine. Furthermore, we contribute with a new system implementation, engineered for modern web applications, and for the live coding community to design their own idiosyncratic languages and interfaces applying our signal engine. The evaluation shows that the system runs with high reliability, efficiency and low latency.

1. INTRODUCTION

Since its introduction in 2011, the Web Audio API (WAAPI) [18] has powered a substantial number of libraries and applications for interactive sound and music. Among WAAPI-powered libraries, there have been varied approaches and features provided: from high-level control and abstractions over WAAPI graphs and native nodes (e.g. Tone.js [12]), to abstractions with low-level digital signal processing (DSP) using optimised JS (e.g. Genish.js and Gibberish.js [14]), asm.js (e.g. Maximilian [6]) and WebAssembly (e.g. Faust [10], CSound [9]). Some of these libraries which initially implemented custom nodes using *ScriptProcessorNode*, with its inherent limitations (e.g. latency, main thread interrupts), have been shifting towards utilising *AudioWorklet* [3] (e.g. Faust, CSound, Genish.js). Such improvements have been used previously in advancing the design of live coding environments for the Web that enable custom low-level DSP and high-level control of musical processes [14], and that support end-user live coding language design [19].

Client-side Web-based machine learning middleware have shown recent and meaningful advances (e.g. Tensorflow.js [16], Magenta.js [13], ml5.js [15], RapidLib.js [7]). These libraries enable end-to-end client-side machine learning workflows with many benefits, including simplified deployment with "zero install," instant access across platforms, data privacy and, in the cases of Tensorflow.js -based libraries, hardware acceleration. Common to these libraries has been a strong consideration for developer-friendly experience design, particularly in relation to API abstraction level and learning resources, documentation and examples. There are also significant challenges involved, concerning varying client-side hardware and the progress of the web browser ecosystem.

We believe that the improvements in these Web technologies can afford the evolution of an ecosystem of real-time, user-defined, live coding languages that combine machine learning, machine listening and audio threads. Previous attempts to integrate real-time interactive audio and machine learning in a scalable and accessible environment such as the Web, have been fraught with technical limitations of the single-threaded JavaScript (JS) environment and the Web Audio API *ScriptProcessorNode* (SPN) [7]. We contribute a novel system design for live coding with a high-performance signal engine leveraging the *Maximilian* DSP library transpiled for *AudioWorklet*[1].

The paper is structured as follows: we first present the requirements and a proposal for a middleware architecture that integrates signal synthesis and processing, machine learning and live coding language design for client-side Web applications. Section 3 presents the elements of our design strategy and rationale for refactoring Maximilian [6] for an *AudioWorklet*-based signal engine. Section 4 describes the implementation and first use of an early signal engine prototype for a live coding performance. Section 5 discusses some of the main limitations for the audio worklet implementation and section 6 concludes with the main takeaways and future work.

2. INTEGRATED ARCHITECTURE OF SIGNAL SYNTHESIS AND MACHINE LEARNING, FOR A WEB-BASED LIVE CODING ECOSYSTEM

In 2018, we conducted a survey on language design with communities of practitioners of live coding, machine listening, machine learning and deep learning [8]. We asked participants which features they envisioned for future live coding environments and languages that could integrate machine listening and machine learning. The findings indicated a wide space of possibilities. Respondents suggested features

[1]https://github.com/mimic-sussex/Maximilian

Licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). **Attribution**: owner/author(s).

Web Audio Conference WAC-2019, December 4–6, 2019, Trondheim, Norway.

© 2019 Copyright held by the owner/author(s).

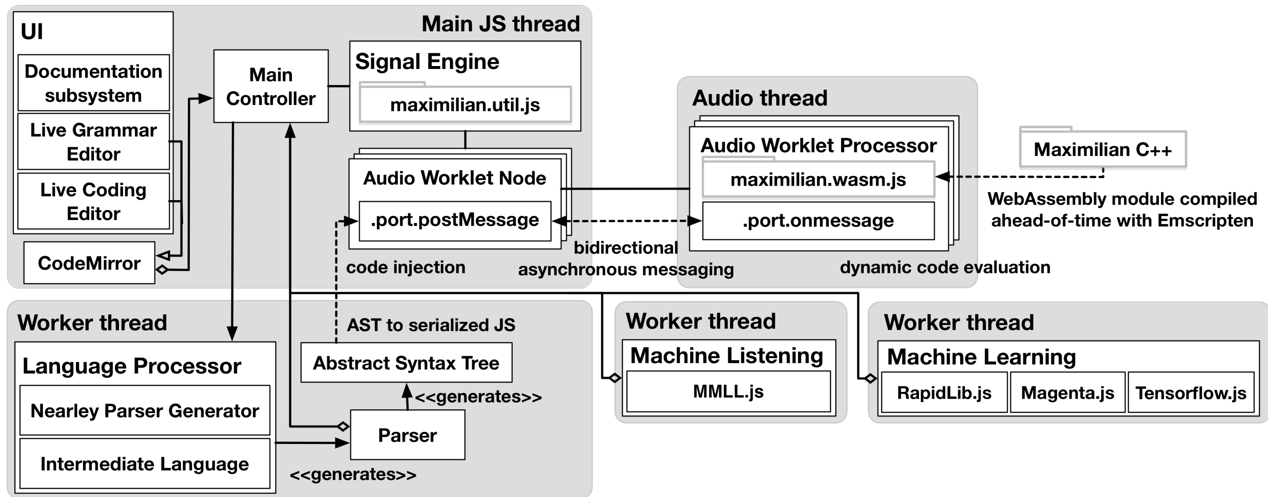

Figure 1: Diagram of the architecture of *Sema*, our live coding language design system.

such as support for hybrid approaches and multi-paradigm languages (i.e. object-oriented and functional), flexible, expressive and extensible languages and prototyping environments, good quality documentation and examples, as well as a clear and informative error report system. In a nutshell: brevity, simplicity, expressivity, flexibility, adaptability and plurality.

These findings prompted us to build *Sema*, a modern Web-based system (Figure 1) to support rapid prototyping of multiple mini-languages for live coding, enabling users to satisfy their idiosyncratic expressive preferences. It seemed wiser to make a system where the user creates or refines their favourite language, rather than trying to satisfy everyone with one general design. Considering also the history and tradition of live coders building their own systems [11], this decision would contribute to a plurality of systems in a field that is already brimming with inventive solutions. Our design exploration will help us find and refine the constraints, meta constraints and features for our system to support live coding language design.

2.1 Design requirements for a live coding system's signal engine

In order to support a broad spectrum of live coding languages that ranges from abstract high-level languages to more powerful low-level languages, we strive for an adequate compromise between simplicity and flexibility. The following are the design principles that guided the implementation of our signal engine:

Integrated signal engine There is no conceptual split between the intermediate language and signal engine. Everything is a signal.

Single-sample signal processing Per-sample sound processing affords a broad set of sound control and synthesis processes, including more sophisticated and expressive techniques that use feedback loops, such as physical modelling, reverberation and infinite impulse response filtering [17].

Sample rate transduction It is simpler to do signal processing with one principal sample rate, i.e. the audio rate. Different sample rate requirements of dependent objects can be resolved by upsampling and downsampling, using a 'transducer'. The 'transducer' analogy enables us to accommodate a variety of processes with varying sample rates (e.g. video, spectral rate, sensors, ML model inference) within a single engine.

Minimal abstractions There are no high-level abstractions such as buses, synths, nodes, servers, or any language scaffolding in our signal engine. Such abstractions sit within the end-user mini-language design space.

We prioritise usability and learnability in the design of our signal engine and intermediate language. As such, some of these design principles prioritise usability over efficiency (e.g. single-sample processing [20]). Nevertheless, we are balancing out any performance trade-offs with a highly efficient implementation (as demonstrated in section 4).

2.2 ML and MML workers

We are designing machine learning (ML) and musical machine listening[2] (MML) as first-class citizens and core components of our system. Typically, both ML and MML processes have computationally intensive stages that are better suited for execution on a dedicated thread, or eventually, in an external process. Even with interactive machine learning workflows [2], where end-users build custom ML models from small, lightweight, user-created data sets, having the main JS thread freeze while an ML algorithm trains a model, causes critical usability issues and undermines the user experience of any application [1]. As such, ML and MML processes in our system follow a loosely coupled architecture based on JS workers[3]: threads which leverage multi-core CPUs and communicate using asynchronous message exchange and event streams with the main thread. These processes also adhere to our transducer concept, in that the sample rates from the event streams they generate are converted to and from the sample rate of the audio context.

[2]https://mimicproject.com/guides/mmll
[3]https://w3c.github.io/workers/

2.3 Live code and grammar editors

After some experimentation integrating editor components such as Ace,[4] Monaco[5] and CodeMirror,[6] we opted for the latter. Our choice considered multiple criteria, such as component architecture, community adoption, maintenance and support, and ease of integration with webpack[7]. CodeMirror powers two editor instances in our web-based live coding environment, which consist of the main user-facing components—the live coding and live grammar editors. A responsive live coding editor provides the user with both manual and continuous evaluation. The grammar editor enables users to specify, inspect and customise different mini-language grammar specification. Both editors will integrate with an interactive documentation subsystem comprising conceptual knowledge guides, tutorials and examples.

2.4 Language processor and intermediate language

In order to parse, validate and evaluate instances of a live coding language from textual input in the editors, we need to be able to define its formal grammar. We opted for Nearley.js, a toolkit and library that implements the Earley [5] algorithm to generate JS parsers from formal grammar specification in the Backus-Naur Form (BNF). Nearley allows a broader set of grammars than parsing expression grammars, a formalism which has been previously used in live coding language design (e.g. PEG.js [19] and Ohm-js[8]), including ambiguous grammars with left-side recursion.

The trade-off for the versatility and flexibility of Nearley is performance. Even with the integration of a tokeniser for higher parsing performance, benchmarks measured across browsers on a large JavaScript Object Notation (JSON) file,[9] show that Nearley scores below domain specific language (DSL) and hand-written parser implementations, parsing libraries, and other parser generators. Nevertheless, preliminary tests that we present later in this paper show that its performance is reasonable for the latency requirements of live coding in which, typically, small files and code snippets are evaluated.

3. DESIGN STRATEGY: REFACTORING MAXIMILIAN FOR AUDIO WORKLET

At the core of our signal engine is Maximilian [7]. It is a free, open source and MIT-licensed audio synthesis and signal processing library implemented in C++. Maximilian was designed as a cross-platform and easy-to-use library for artists and creative computing students, and for rapid prototyping and commercial software development of interactive audio applications. The library provides a high-level interface to a comprehensive set of DSP primitives, including oscillators with standard waveforms, envelopes, filters with resonance, sample playback, delay lines, Fast Fourier Transform (FFT), granular synthesis, feature extraction, and multi-channel support.

Previously, Maximilian was transpiled to an asm.js library using Emscripten [21].[10] The output consisted of single file with asm.js appended with custom JS—e.g. a pre-assembled WAAPI graph with a user-definable closure injected at the audio callback of a SPN, JS TypedArray conversion to emscripten native types and asynchronous sample loading. This library provided a simple framework and abstraction over the WAAPI graphs and nodes. Most usefully, the closure bound to the scope of the SPN audio callback provided users with a sandbox for accessing the audio buffer directly and implementing custom signal processing for interactive audio applications with Maximilian DSP classes.

While this solution revealed useful to end-users, particularly for teaching students about DSP and audio with the accessibility and convenience of a Web environment [7], it suffered from the well-known limitations of SPN [3]. In earlier studies [1], we assessed users integrating Maximilian with machine learning and different data sources for rapid prototyping interactive audio applications. The main thread interrupts at the ML-training stage resulted in audio glitches and freezes that frustrated users. These usability issues were considered critical and would clearly undermine the adoption of these tools.

In order to overcome the SPN issues, and to arrive at a more scalable, efficient and usable design for a signal engine, we decided to advance towards an AudioWorklet-based implementation, and to refactor Maximilian.js accordingly.

3.1 WebAssembly generation with Emscripten

Refactoring Maximilian.js for *AudioWorklet* implied changing the build process to generate Wasm instead of asm.js. Compiling Maximilian to Wasm caused a breaking change that rendered the Maximilian DSP types unavailable in the SPN callback at load time, due to the asynchronous Wasm module instantiation. To ensure retro-compatibility to Maximilian.js SPN users, we restructured the build process to provide conditional builds of the library. The SPN version is modularised, exports with library name, and provides embindings with smart-pointer constructors for automated memory management and garbage collection of the DSP types. The Wasm version of the module was trimmed down and now only contains the embindings for loading the DSP classes in the AudioWorkletProcessor (AWP) scope with normal constructors for finer control over object disposal. Listing 1 shows an excerpt of the embinding for *maxiOsc*, Maximilian's native type for standard wavetable oscillators.

Listing 1: Excerpt of Maximilian maxiOsc embinding

```
class_<maxiOsc>("maxiOsc")
#ifdef SPN
 .smart_ptr_constructor("shared_ptr<
    maxiOsc>", &std::make_shared<maxiOsc
    >)
#else
 .constructor<>()
#endif
 .function("sinewave", &maxiOsc::sinewave)
 .function("saw", &maxiOsc::saw)
```

The design pattern[11] for *AudioWorklet* with WebAssembly provides a restricted set of Emscripten compilation flags.

[4]Ace Editor, https://ace.c9.io/
[5]Monaco Editor, https://microsoft.github.io/monaco-editor
[6]CodeMirror, https://codemirror.net
[7]Webpack, https://webpack.js.org/
[8]Ohm-js, https://github.com/harc/ohm
[9]https://sap.github.io/chevrotain/performance/
[10]https://gitlab.doc.gold.ac.uk/mick/maxi-js-emscripten
[11]https://developers.google.com/web/updates/2018/06/audio-worklet-design-pattern

We found that some of the previously used compilation flags, such as `MODULARIZE` and `EXPORT_NAME` prevented the WASM module to load successfully and were removed.

```
CFLAGS=--bind -O1 -s WASM=1 -s SINGLE_FILE=1
-s BINARYEN_ASYNC_COMPILATION=0 \
-s ABORTING_MALLOC=0 -s ALLOW_MEMORY_GROWTH=1 \
-s TOTAL_MEMORY=128M
```

To support the strategy of dynamic code evaluation, we found that it was necessary to add a compilation option to control the Wasm heap memory expansion. We will discuss this further in Section 5.

3.2 Audio worklet code injection

Upon the introduction of *AudioWorklet*, a few community examples emerged that use dynamic code injection (e.g. dsp.audio's Worklet Editor[12] and A. Carabott's Live Coding Playground[13]). These examples de-serialise worklet processor templates in vanilla JS using Blob, manipulate them, and hot-swap them from memory into the worklet as interchangeable modules. Our first attempts explored this pattern using code generation and the additional constraint of dynamically loading the Maximilian Wasm module into the *AWP* scope. This constraint introduced a series of difficulties revealed by the "*Error on loading worklet: DOMException*" error message. The ambiguity of the error message and the constraints of the *AWP* scope caused a challenging debugging process.

To solve this problem, we considered tactics such as dynamic imports, fetch, and *wasm-through-'message port'*, as suggested in WAAPI discussion forum. We found out that the AWP scope excluded some of the APIs that enable the first alternatives. We also found out through StackOverflow[14] that the opaque origin of the worklet processor requires absolute URIs to load the module asynchronously. This unveiled another problem related to the unsuccessful parsing of the Wasm module when it was imported by processor code that was generated and loaded dynamically from a Blob. Eventually, we turned to a more creative solution.

3.3 Parsing, evaluation and execution of user-defined code

The broader design of our livecoding system will be discussed in a forthcoming paper. Here we present the solution that we have arrived at for supporting the evaluation of user-defined DSP code. Fundamentally, it consists in utilising the worklet processor as a template that imports the Maximilian Wasm module and loads its DSP classes into the AWP scope. The AWP is where the DSP code is injected and dynamically evaluated.

In our system, when users evaluate an expression in the live coding editor—i.e. by pressing Cmd-Enter after selecting an expression or placing the cursor on a given line in the editor—they are triggering a specific workflow in our system (Figure 1). First, the user-evaluated expression is parsed by the Nearley-generated parser. If the expression is valid according to the language formally defined by the BNF grammar specification, the parser outputs an Abstract Syntax Tree (AST), a tree-like data structure that breaks down the user expression. The AST is converted into serialised JS expressions that specify which Maximilian DSP objects are used and how they are assembled into DSP functions that will run on the main audio loop `setupFunction` and `loopFunction`, respectively. These JS expressions are packed into a JS object which is posted through the processor *AudioWorkletNode* messaging port (Listing 2).

Listing 2: Code injection into the AWP scope

```
this.audioWorkletNode.port.postMessage({
  eval: 1,
  setup: userInterpretedFunction.setup,
  loop: userInterpretedFunction.loop
});
```

Once the message containing a payload of serialised DSP JS resolves asynchronously in AWP scope, the code is evaluated just-in-time (JIT) using the `eval()` function. The user-evaluated expression that was converted to a JS DSP specification with Maximilian objects is compiled and set ready to run in the AWP loop (Listing 3).

Listing 3: Dynamic code evaluation in the AWP scope

```
this.port.onmessage = event => {
  if ("eval" in event.data) {
    try {
      let setupFunction =
        eval(event.data["setup"]);
      let loopFunction =
        eval(event.date["loop"]);
      ...
    } catch (err) {
}
```

For instance, considering the following expression for FM synthesis (Listing 4), that is valid for of a live coding language that a user has defined through a given grammar:

Listing 4: Example of a test user-evaluated DSP expression

```
osc sin(osc sin 100 + osc tri 40)
```

The system interprets this expression and generates two anonymous functions. The *setupFunction* (Listing 5) declares and instantiates the required Maximilian Wasm objects for the custom DSP expression in the AWP scope.

Listing 5: The *setupFunction* generated from the AST

```
() => {
  let q=[];
  q.osc13 = new Module.maxiOsc();
  q.osc14 = new Module.maxiOsc();
  q.osc15 = new Module.maxiOsc();
  return q;
}
```

The original user-expression is converted to a compiled JS function, the *loopFunction* (Listing 6), which composes the Maximilian objects previously declared in *setupFunction* and runs in the AWP *process* loop (the main audio loop). The audio engine crossfades between the previous *loopFunction* and the newly generated one.

Listing 6: The *loopFunction* generated from the AST

```
(q) => {
  return q.osc13.sinewave((
    q.osc14.sinewave(100)
    + q.osc15.triangle(40))
  );
}
```

[12] Worklet editor, http://dsp.audio/editor
[13] https://acarabott.github.io/audio-worklet-live-coding
[14] http://bit.ly/2F4vZKM

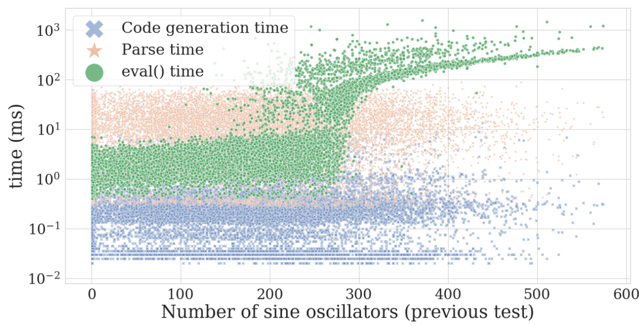

Figure 2: Load testing results

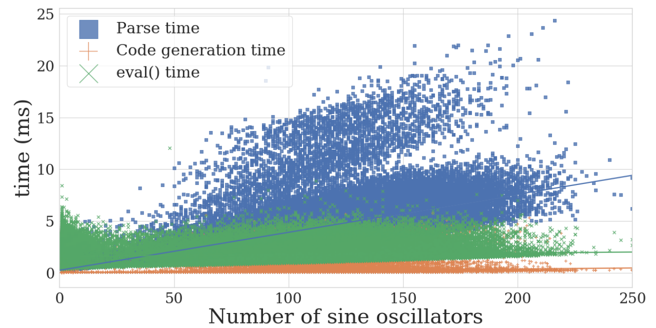

Figure 3: Latency test results: individual processes

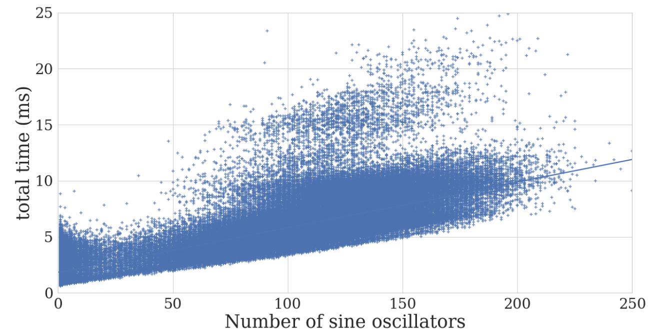

Figure 4: Latency test results: combined processes

4. PERFORMANCE AND RELIABILITY EVALUATION

Good performance and reliability are central measures of the success of a live coding system, which must be able to meet the demands of live musical performance. We measured the reliability, latency and signal processing capacity of our system. To achieve this, we designed a simple testing language that was capable of creating nested trees of oscillators. An algorithm was designed to probabilistically generate code for oscillator trees with variable width and depth. An example of generated code is shown below.

```
osc sin (osc sin 960 + osc sin 961 + osc
   sin (osc sin 657 + osc sin 348 + osc
   sin 405) + osc sin (osc sin 1050 + osc
    sin 122 + osc sin (osc sin 1052 + osc
    sin 1090 + osc sin (osc sin 1088 +
   osc sin (osc sin 258) + osc sin (osc
   sin (osc sin 820))) + osc sin 221)))
```

This code generator could be hand tuned to change the probable size of its output. The test procedure worked as follows: (1) generate some code, (2) record the number of oscillator objects in the code, (3) run the code in the audio engine, recording times for parsing, JS code generation from the parse tree, and for compiling the code using `eval()`, (4) wait for 200ms and repeat.

All tests run in Chrome 75.0, on a MacBook Pro mid-2015, with 16Gb RAM and a 2.5GHz i7 CPU. We recorded time using `window.performance.now()`, with sub-millisecond resolution. These results figures measure the capacity of a single audio worklet, rather than the capacity of the browser's JS virtual machine, which is capable of running multiple audio worklets.

4.1 Signal Processing Capacity

Our first investigation tested the signal processing limits of the system, in a test designed to mimic real-world scenarios. The code generator was configured to create code that might overload the AWP signal processing loop. When overloading happens, this has an effect on the `eval()` time of the subsequent test, because the `eval()` command is carried out in the AWP thread and slows down if it is overloaded. We ran the test for one hour, over 14000 iterations. Figure 2 shows a scatter plot comparing the number of sine wave oscillators running in the previous test compared to the time taken for `eval()` to run in the current test. The `eval()` time is stable until around 200 oscillators, when variance starts to increase, and then above 250 oscillators we start to see an increase in latency. Parsing and code generation times remain stable throughout, as they are unaffected, running in a separate worker thread. From these results, we can estimate that the system will run with good stability with approximately 200-250 sine wave oscillators.

4.2 Reliability and Latency

We tested the latency of the system within the stable limits discovered in the previous test. This is the time between when a user evaluates a line of code, and when the compiled code is ready to run in the next DSP cycle. The test ran for ten hours and 11 minutes, with 173422 iterations. Extreme outliers (with z-score over 20) were removed, accounting for 0.007% of datapoints. The remaining 99.993% of tests show latency of <25ms, indicating high reliability. Figure 3 shows individual latencies compared to the number of sine wave oscillators in the test code. The largest and most variable cause of latency is the Nearley parser. The combined latency times (Figure 4) demonstrate that the system is likely to run with sub-perceptual (<20ms) latencies when running within stable limits. The blue lines in both figures show a linear regression fit.

5. DISCUSSION AND DESIGN PATTERNS

The JS community often considers the use of the `eval()` function a bad coding practice [4]. However, it has been used in the Webkit console, JSBin, and the widely adopted library lodash[15]. The problems of using `eval()` have been associated with malicious attacks through code injection and cross-site scripting, hard maintenance and debugging, slow performance due to the cost of compilation, and wastefulness of resources (e.g. creating execution context collecting garbage, exporting variables). While we are

[15]Angus Croll: Break all the rules, JSConf 2012, https://youtu.be/MFtijdklZDo

still exploring this solution within the constraints of our scenario—e.g. client-side, immediate execution in the AWP scope—our evaluation metrics suggest that the advantages of `eval()` supersede the potential problems. One problem that we found with using `eval()` and Wasm was in the heap memory expansion. Although Emscripten provides an ALLOW_MEMORY_GROWTH compilation option, we found it led to inconsistent behaviours. We resolved this issue by setting a maximum memory growth threshold.

In the exploration and gradual development of our solution, we dealt with difficulties arising from the integration of technologies with different degrees of maturity. Some difficulties inherent to *AudioWorklet* were, for instance, error messaging, the availability of APIs in the AWP scope, and its impact in the Wasm module loading (e.g. for standard modularisation and module naming). Other difficulties emerged from loading these components within a webpack development environment which revealed somewhat cumbersome (e.g. adding imports for emitting Wasm without compilation, server MIME type definitions).

Our solution can provide a design pattern for people interested in re-using our stack or developing a similar approach. The first use of the signal engine was in a live coding performance at AlgoMech 2019[16]. The signal engine integrated with the live coding artist's web page and supported its use as the main live coding instrument. Live coding was performed directly in Chrome Dev tools console.

6. CONCLUSION AND FUTURE WORK

We present a high-performance implementation of a signal engine for a new Web-based system for the live coding community. Evaluation shows that the system runs with efficiency and high reliability for 200-250 oscillators with subperceptual latency. Future work will focus on the presentation layer, user language specification support, and further development of the ML and MML workers. These developments include moving to an optimised parser implementation with WebAssembly JIT compilation, and supporting *SharedArrayBuffer* for inter-thread communication.

7. ACKNOWLEDGMENTS

This work was supported by the AHRC-funded MIMIC project, ref: AH/R002657/1 (https://gtr.ukri.org/projects?ref=AH/R002657/1).

8. REFERENCES

[1] F. Bernardo, M. Grierson, and R. Fiebrink. User-Centred Design Actions for Lightweight Evaluation of an Interactive Machine Learning Toolkit. *Journal of Science and Technology of the Arts*, 10(2):2, 2018.

[2] F. Bernardo, M. Zbyszyński, R. Fiebrink, and M. Grierson. Interactive Machine Learning for End-User Innovation. In *Proc. of the Association for Advancement of ArtificiaI Intelligence Symposium Series: Designing the User Experience of Machine Learning Systems*, pages 369–375, 2017.

[3] H. Choi. AudioWorklet: The future of web audio. In *Web Audio Conference*, 2018.

[4] D. Crockford. *JavaScript: The Good Parts.* O'Reilly Media, first edit edition, 2008.

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] M. Grierson and C. Kiefer. Maximilian: An Easy to Use, Cross Platform C++ Toolkit for Interactive Audio and Synthesis Applications. In *Proc. of The International Computer Music Conference*, number August, pages 276–279, 2011.

[7] M. Grierson, M. Yee-king, L. McCallum, C. Kiefer, and M. Zbyszyński. Contemporary Machine Learning for Audio and Music Generation on the Web: Current Challenges and Potential Solutions. *ICMC*, 2018.

[8] C. Kiefer and T. Magnusson. Live Coding Machine Learning and Machine Listening: A Survey on the Design of Languages and Environments for Live Coding. In *Proc. of the International Conference on Live Coding.*, Madrid, 2019.

[9] V. Lazzarini, E. Costello, and S. Yi. Csound on the web. In *Linux Audio Developers Conference*, 2014.

[10] S. Letz, S. Denoux, Y. Orlarey, and D. Fober. Faust audio DSP language in the Web. *Linux Audio Conference*, pages 29–36, 2015.

[11] T. Magnusson. Herding Cats: Observing Live Coding in the Wild. *Computer Music Journal*, 38(1):91–101, 2014.

[12] Y. Mann. Interactive Music with Tone.js. *Proc. of the 1st annual Web Audio Conference*, 2015.

[13] A. Roberts, C. Hawthorne, and I. Simon. Magenta.js: A JavaScript API for Augmenting Creativity with Deep Learning. In *Proc. of the 35th International Conference on Machine Learning*, pages 2–4, 2018.

[14] C. Roberts. Metaprogramming Strategies for AudioWorklets. In *Web Audio Conference*, 2018.

[15] D. Shiffman, J. Ashe, K. Compton, H. Davis, D. Kazemi, G. Kogan, K. McDonald, and Yining Shi. ml5: Friendly Open Source Machine Learning Library for the Web. *ADJACENT*, (3), 2018.

[16] D. Smilkov, N. Thorat, Y. Assogba, A. Yuan, N. Kreeger, P. Yu, K. Zhang, S. Cai, E. Nielsen, D. Soergel, S. Bileschi, M. Terry, C. Nicholson, S. N. Gupta, S. Sirajuddin, D. Sculley, R. Monga, G. Corrado, F. B. Viégas, and M. Wattenberg. TensorFlow.js: Machine Learning for the Web and Beyond. In *Proc. of the 2nd SysML Conference*, 2019.

[17] K. Steiglitz. *A Digital Signal Processing Primer: With Applications to Digital Audio and Computer Music.* Addison-Wesley, 1996.

[18] W3C Audio Working Group. Web Audio API. https://webaudio.github.io/web-audio-api, 2019.

[19] G. Wakefield and C. Roberts. A Virtual Machine for Live Coding Language Design. *Proc. of New Interfaces for Musical Expression 2017*, pages 275–278, 2017.

[20] G. Wang, P. R. Cook, and S. Salazar. ChucK: A Strongly Timed Computer Music Language. *Computer Music Journal*, 39(4):91–101, 2015.

[21] A. Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *Proc. of the ACM international conference companion on Object-Oriented Programming, Systems, Languages, and Applications.* ACM, 2011.

[16] Algo/Mesh, https://algomech.com/2019/events/mesh/