# AudioWorklet: The future of web audio

**Hongchan Choi**
Google Chrome
hongchan@chromium.org

## ABSTRACT

*This paper presents a newly introduced extension to the Web Audio API that enables developers to author custom audio processing on the web. AudioWorklet brings JavaScript-based audio processing that is fast, reliable and secure while addressing numerous problems found in the earlier ScriptProcessorNode. AudioWorklet's design is discussed, followed by technical aspects, usage examples, and what the new functionality offers to the computer music community.*

## 1. BACKGROUND

### 1.1 What is AudioWorklet?

Since its birth in 2010, the Web Audio API [1] has transformed the web browser into a platform for interactive applications for music and audio. Although it has been reasonably successful in accommodating various use cases with its highly dynamic design, there has been a major criticism pointing out its lack of flexibility and extensibility. Besides built-in features for basic audio processing and synthesis, the API also provided developers with a way of running user-supplied JS code through ScriptProcessorNode, a provision which failed to meet the expectations of the developer community. [2]

To address this issue, the W3C Audio Working Group [1] started working on a new functionality, named *AudioWorklet*, to support sample-accurate audio manipulation in JavaScript without compromising performance and stability. The initial design of the AudioWorklet interface was introduced in an API specification in 2014 and its first operational implementation was released to the public in early 2018 in the Chrome browser.

AudioWorklet is the most anticipated enhancement in the history of Web Audio API, however its genesis and advantages have not been fully discussed yet. The goals of this article are to 1) present a comprehensive comparison between AudioWorklet and its predecessor, ScriptProcessorNode, 2) provide usage examples, 3) discuss technical aspects and 4) highlight what this new technology brings to computer musicians.

[1] https://www.w3.org/2011/audio/

*Copyright: ©2018 Hongchan Choi et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License 3.0 Unported, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.*

### 1.2 Web Audio API processing model

To appreciate the advantages of AudioWorklet over ScriptProcessorNode, it is helpful to understand how Web Audio API operates internally. The following is a quick recap of the processing model embodied in the API specification.

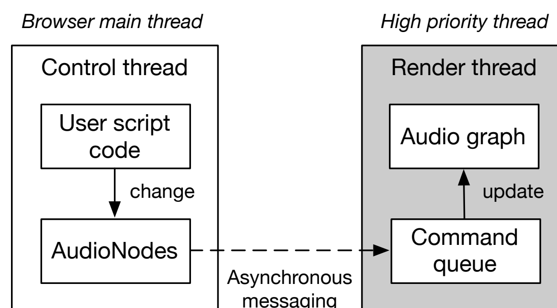


**Figure 1**. The processing model of the Web Audio API

Two crucial premises of the Web Audio API rendering mechanism are that 1) the audio rendering needs to be performed on a dedicated thread with high priority and 2) the rendering pipeline must have a fixed size *render quantum* of 128 frames. The intention behind these prerequisites is to obtain optimum audio rendering performance out of generic programming platforms like web browsers.

The Web Audio API needs two threads, the *control thread* and the *rendering thread*. The control thread is the browser's main thread where users create and manipulate Web Audio API objects. This thread is primarily reserved for generic tasks such as DOM (document object model) processing and running JavaScript code. The rendering thread is where the audio engine runs to produce the audio stream. This separation is necesssary to maintain audio rendering performance. Having a fixed render quantum ensures the rendering process can be much more efficient. Native implementations, for example, can apply different optimization techniques such as SIMD and vectorization around the fixed buffer size.

The interval of the render callback [2] is designed to be good enough for high-level user interactions like triggering musical notes. This interval is, however, too coarse for sample-accurate audio manipulation of synthesis and modulation. *AudioParam* was devised to solve this problem. One can schedule a parameter automation via the control thread and, in turn, the sample-accurate audio manipulation will be performed on the rendering thread at a later time. [3]

Despite AudioParam's ability to exert finer changes in the

[2] Approx. 2.67ms at 48KHz sample rate.

audio stream, the API still does not expose a way of running a user-defined callback function. To address this lack of flexibility, ScriptProcessorNode was added to the specification. It quickly gained popularity as a means for experimentation because the callback function can easily be written in JavaScript.

ScriptProcessorNode was the answer to the demand for extensibility from developers wishing to run user script code within Web Audio API but it was a less than ideal solution. The feature is now deprecated from the specification due to critical design flaws and has been replaced with AudioWorklet. The following section provides details on the problems caused by ScriptProcessorNode and the solutions proposed by AudioWorklet.

## 2. AUDIOWORKLET VERSUS SCRIPTPROCESSORNODE

### 2.1 Clean separation between node and processor

A native AudioNode is comprised of two components: the user-facing component (the *node*) and its internal processor component (the *processor*). The node is a valid JS object within the programmable interface (i.e. garbage-collected) and the processor takes care of the rendering functionality as a part of the audio graph. This serves two purposes: 1) to put the node and the processor in the control and the rendering threads, respectively and 2) to avoid garbage collection in the rendering thread by detaching the processor from the node.

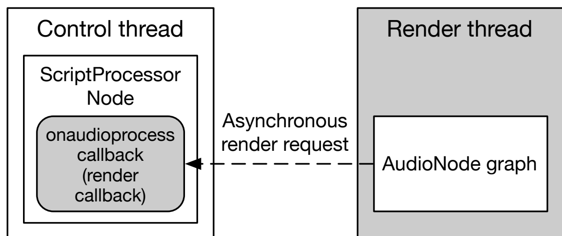


**Figure 2**. ScriptProcessorNode rendering mode

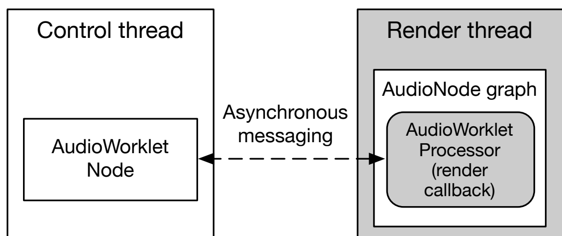


**Figure 3**. AudioWorkletNode rendering mode

Unlike all other AudioNodes, ScriptProcessorNode does not have such separation. Both the node and the processor (i.e., render callback) live on the main thread – and that has been the root cause of the difficulties. The first order of business for AudioWorklet was to clear up such a critical architectural flaw. For this reason, the new design includes an *AudioWorkletNode* (node) and an *AudioWorkletProcessor* (processor). The AudioWorkletNode lives in the main scope (i.e., window) and the corresponding AudioWorkletProcessor lives in *AudioWorkletGlobalScope*, a special scope for audio rendering purposes.

As mentioned, ScriptProcessorNode's audio process callback was computed in the main thread. This meant that when the rendering thread signals to the main thread to fill a buffer in ScriptProcessorNode, the task would be queued in the task scheduler and the callback function would fire at a later time. The problem here is the main thread task queue is usually crowded with various tasks and the `onaudioprocess` callback is highly likely to be delayed in a nondeterministic manner.

By moving the execution of the callback function to the AudioWorkletGlobalScope (which runs on the rendering thread), a significant improvement of rendering stability and performance can be achieved. The main thread also benefits, because running CPU intensive audio code every few milliseconds can bring intense pressure to the main thread and cause UI stuttering and freezing. Scope isolation is a big win for both sides.

### 2.2 Achieving Synchronous rendering

In Chromium, ScriptProcessorNode implemented the data exchange between the main thread and the rendering thread using a double buffering mechanism. However, double buffering comes with a few problems like latency, audio dropouts and duplicate. So why did we need it in Chromium's implementation? First, because the user-defined callback function (running in the main thread) is invoked asynchronously from the audio rendering thread. Second, because the buffer size of ScriptProcessorNode is not fixed like the render quantum size of Web Audio API's rendering engine. Double buffering was a relatively simple solution to reconcile these two conflicts.

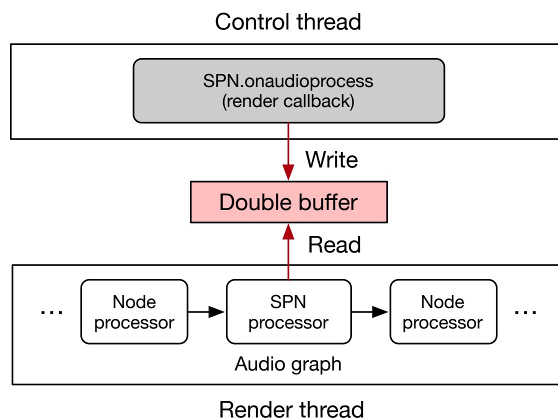


**Figure 4**. ScriptProcessorNode rendering mode

Despite being a convenient workaround, double buffering can be the source for undesirable side effects. A mutex is required to avoid race conditions in shared storage between the two different threads. Also, the timing of read and write operations are orthogonal. For instance, the main thread can be stalled while the audio thread keeps reading data out of the buffer and when the read task reaches the end of storage, the index will go back to the beginning of the buffer and read old data over again (duplicated audio). Alternatively, if the main thread is holding the buffer

and the audio thread can't read the data in time, the ScriptProcessorNode will output a buffer of silence for (a muted render quantum) heard as an audio dropout.

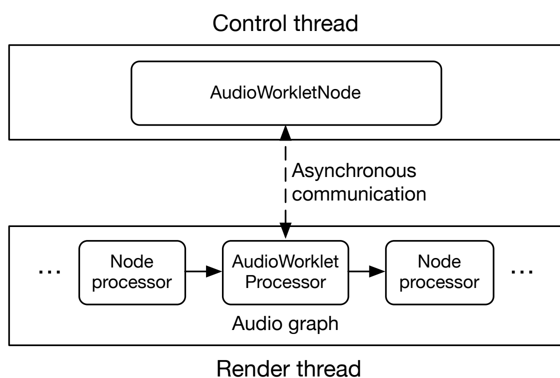


**Figure 5**. AudioWorklet rendering mode

Synchronous rendering is the one of the biggest advantages AudioWorklet offers. It eliminates the issues discussed above by placing the processing code directly in the rendering thread. There is no cross-thread function call and no need for internal buffering. To ensure the optimum performance, the buffer size of AudioWorkletProcessor is now limited to 128 frames but this rather helps AudioWorkletNode operate similar to the native AudioNode. In other words, AudioWorkletNode is computed as a first-class citizen in the world of Web Audio.

### 2.3 Extensibility: First-class Object

From the initial review of the Web Audio API by the W3C TAG (Technical Architecture Group) [3] , one criticism was the general lack of extensibility in Web Audio API. [2] Although the API was shaped to provide a set of well-defined building blocks, there are certain to be use cases that cannot be addressed by these predefined components. Before the AudioWorklet, ScriptProcessorNode was the only thing that could be fully customized, but it was far from being a first-class object in Web Audio API because of its different nature e.g., the lack of AudioParam support or a proper constructor.

Integrating support of AudioParam was required for AudioWorklet to become a first-class object in the API. In the class definition of AudioWorkletProcessor, custom AudioParam objects can be declared. The parameter values are calculated by the rendering engine and they can be directly accessed in the processor's callback function.

Extensibility is a key to building a thriving developer community. Experience has shown that the Web Audio API failed to provide developers with a proper way of extending built-in components in a scalable manner. As a result, numerous experimental projects ended up using ScriptProcessorNode and are now endangered due to the deprecation of the node. For this reason, extensibility was the most important design goal of AudioWorklet from the beginning; every aspect of its operation (i.e., class definition, instantiation and processing) can be extended in a way that is compatible with the built-in AudioNodes which, in fact, *can* be implemented with AudioWorkletNode.

[3] https://www.w3.org/2001/tag/

## 3. USAGE

The actual usage of AudioWorklet is somewhat involved compared to ScriptProcessorNode and simple drop-in replacement might be difficult. This stems from some of the differences already outlined including becoming a first-class object. Also, for programmers who are not familiar with multi-thread (e.g. WebWorker) programming, its pattern would not be straightforward. This section presents basic code examples of AudioWorklet with explanations to help build understanding. [4]

### 3.1 Registration and Instantiation

Using AudioWorklet consists of specifying two parts: an AudioWorkletNode and an AudioWorkletProcessor. This is more involved than using ScriptProcessorNode, but it is needed to give developers the low-level capability for custom audio processing. AudioWorkletNode is the counterpart of AudioWorkletProcessor and takes care of connections to and from other AudioNodes in the audio graph maintained in the main thread. It is exposed in the main global scope and functions like a regular AudioNode. AudioWorkletProcessor represents the actual audio processor written in JavaScript code, and it lives in the AudioWorkletGlobalScope.

Here's a pair of code snippets that demonstrates registration and instantiation.

```
// The code in the main global scope.
class MyWorkletNode
    extends AudioWorkletNode {
  constructor(context) {
    super(context, 'my-worklet-processor');
  }
}

const context = new AudioContext();
context.audioWorklet.addModule(
    'processors.js').then(() => {
  const node = new MyWorkletNode(context);
});
```

Creating an AudioWorkletNode requires at least two things: a BaseAudioContext object and the processor name as a string. You can subclass AudioWorkletNode to define a custom node which will handle the processor running on the rendering thread. A processor definition is loaded and registered by AudioWorklet's `addModule()` method.

Worklet APIs including AudioWorklet are only available in a secure context, thus a page using them must be served over HTTPS, although `http://localhost` is considered secure for local testing.

```
// This is "processor.js" file, evaluated in
// AudioWorkletGlobalScope upon
// audioWorklet.addModule() call in the main
// global scope.
class MyWorkletProcessor
    extends AudioWorkletProcessor {
  constructor() {
    super();
  }

  process(inputs, outputs, parameters) {
    // Your audio processing code here.
  }
```

[4] This tutorial is largely based on AudioWorklet article published at https://developers.google.com/web/updates/2017/12/audio-worklet.

```
}

registerProcessor('my-worklet-processor',
                  MyWorkletProcessor);
```

The `registerProcessor()` method in the AudioWorkletGlobalScope takes a string for the name of processor and the class definition. [4] After the completion of script code evaluation in the global scope, the promise from `AudioWorklet.addModule()` will be resolved notifying users that the registration is ready to be used in the main global scope.

### 3.2 `process()` callback in AudioWorkletProcessor

The actual audio processing happens in the `process()` callback method in the AudioWorkletProcessor and must be implemented in the class definition. The rendering engine will invoke this function in an isochronous fashion to feed inputs and parameters and fetch outputs.

```
/* AudioWorkletProcessor.process() method */
process(inputs, outputs, parameters) {
  // The processor may have multiple inputs and
  // outputs. Get the first input and output.
  const input = inputs[0];
  const output = outputs[0];

  // Each input or output may have multiple
  // channels. Get the first channel.
  const inputChannel0 = input[0];
  const outputChannel0 = output[0];

  // Get the parameter value array.
  const myParamValues = parameters.myParam;

  // Simple gain (multiplication) processing
  // over a render quantum (128 samples). This
  // processor only supports the mono channel.
  for (let i = 0; i < inputChannel0.length; ++i) {
    outputChannel0[i] =
        inputChannel0[i] * myParamValues[i];
  }

  // To keep this processor alive.
  return true;
}
```

Additionally, the return value of the `process()` method can be used to control the lifetime of AudioWorkletNode so that developers can manage the memory footprint. To keep the processor alive, the method must return true. Otherwise, the processor will be garbage collected by the system eventually after the node gets collected.

### 3.3 Custom AudioParam

One of the useful things about AudioNodes is schedulable parameter automation with AudioParams. AudioWorkletNodes can use these to get exposed parameters that can be controlled at audio rate.

User-defined AudioParams can be declared in an AudioWorkletProcessor class definition by setting up AudioParamDescriptors. The AudioWorkletGlobalScope will pick up this information upon the construction of an AudioWorkletNode, and will then create AudioParam objects for the node accordingly.

```
/* A separate script file, e.g.
 "my-worklet-processor.js" */
```

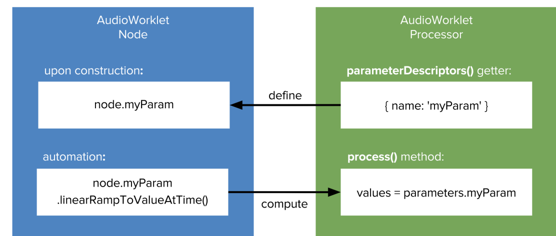


**Figure 6**. AudioParam in AudioWorklet

```
class MyWorkletProcessor
    extends AudioWorkletProcessor {
  // Static getter to define AudioParam objects
  // in this custom processor.
  static get parameterDescriptors() {
    return [{
      name: 'myParam',
      defaultValue: 0.707
    }];
  }

  constructor() { super(); }

  process(inputs, outputs, parameters) {
    // |myParamValues| is a Float32Array of
    // 128 audio samples calculated by
    // WebAudio engine from regular AudioParam
    // operations. (i.e. automation methods,
    // setter) By default this array would be
    // all values of 0.707
    const myParamValues = parameters.myParam;
  }
}
```

### 3.4 Bidirectional communication

Sometimes custom AudioWorkletNodes will want to expose controls that do not map to AudioParam. For example, a string-based type attribute could be used to control a custom filter. For this purpose and beyond, AudioWorkletNode and AudioWorkletProcessor are equipped with a *MessagePort* for bidirectional communication. Any kind of custom data can be exchanged through this channel. [5]

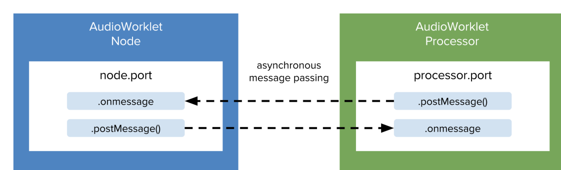


**Figure 7**. MessagePort in AudioWorklet

MessagePort can be accessed via `.port` attribute on both the node and the processor. The node's `port.postMessage()` method sends a message to the associated processor's `port.onmessage` handler and vice versa.

```
/* The code in the main global scope. */
context.audioWorklet.addModule(
    'processors.js').then(() => {
  let node =
      new AudioWorkletNode(context,
                           'port-processor');
  node.port.onmessage = (event) => {
    // Handling data from the processor.
    console.log(event.data.message);
```

```
  };

  node.port.postMessage({ message: 'Hello!'});
});

/* "processor.js" file. */
class PortProcessor
    extends AudioWorkletProcessor {
  constructor() {
    super();
    this.port.onmessage = (event) => {
      // Handling data from the node.
      console.log(event.data.message);
    };

    this.port.postMessage({ message.:'Hi!' });
  }

  process(inputs, outputs, parameters) {
    // Do nothing, producing silent output.
    return true;
  }
}

registerProcessor('port-processor',
                  PortProcessor);
```

Also note that MessagePort supports *Transferable*, which allows you to transfer data storage or a WebAssembly module over the thread boundary. [6] This opens up countless possibilities on how the AudioWorklet system can be utilized.

## 4. IN-DEPTH TECHNICAL DISCUSSIONS

### 4.1 Garbage collection and glitch

In an ideal setting, the real-time audio rendering thread must not be blocked. Any task that synchronously blocks the thread execution is a potential threat to the desired glitch-free audio system. Integrating a JS engine with the WebAudio's rendering engine was a substantial technical challenge since a JS engine is not really suitable for a real-time computing system. Even though recent advancements in the performance of the JS engine have made it possible to run script code blazingly fast, JavaScript is still a garbage-collected language and the garbage collection is a thread-blocking operation.

Garbage collection can stall the rendering task nondeterministically. This is because garbage collection in the JS engine must not be observable and its timing must not be deterministic from user's point of view. For this and other reasons, memory allocation operations in AudioWorkletGlobalScope should be kept to the minimum. Developers need to keep in mind that declaring arrays and objects, or receiving cross-thread messages via MessagePort results in the memory allocation. Any large amount of memory allocation should be done upfront so to avoid blocking the audio rendering callback. All in all, it is helpful to generate less garbage by minimizing the memory allocation and reusing allocated memory as much as possible.

Despite these inherent difficulties, the AudioWorklet project successfully harnesses the power of the JS engine for audio processing purposes. Developers have to be careful to avoid any thread-blocking task in their script code, but this new paradigm opens up unprecedented possibilities for web-based audio applications. Improvements which the AudioWorklet system will leverage include next generation garbage collection techniques that are under the development in modern web browsers (such as generational, incremental or parallel garbage collection).

### 4.2 Security: thread priority and SharedArrayBuffer

Being a part of the web platform endows a great deal of thrilling perks, but it also comes with many constraints that the computer music has rarely encountered. Among those handicaps, security is by far the most critical one. A security breach on the web can have a significant impact on billions of users thanks to its ubiquitous nature. Any new feature on the web must undergo rigorous security review process before it can reach the real-world audience. Needless to say, the Web Audio API cannot be an exception for such validation.

The audio rendering code typically runs on a thread with real-time priority (although the threading model can vary across operating systems and certain platforms require admin privilege to use a real-time thread). Unlike native applications, taking advantage of a real-time thread in the browser must be dealt with care. The real-time thread should not execute user-supplied script code. Given the highest priority, the malicious code can preempt the processors (CPUs) and deschedule other tasks on lower priority threads. This exposes a weakness for certain exploitation techniques, so software architects tend to be very conservative on this matter.

In Chromium, the audio rendering thread maintains its real-time priority unless AudioWorklet system gets activated. When a user explicitly calls `AudioContext.addModule()`, the rendering thread is replaced with a thread with *display* priority. Note that the browser's main thread runs with the same display priority, which is the second-tier priority in the browser. Chromium engineers believed it is reasonable for AudioWorklet to use a higher priority thread because it helps glitch-free audio rendering and AudioWorklet is only available within *SecureContext* [5]. By comparison, regular priority is given to general worker threads and this is for a security reason. WebWorker can run arbitrary user code from a non-secure domain.

In 2017, *SharedArrayBuffer* was proposed as a part of ECMAScript. It enables the parallel processing on the web in conjunction with WebWorker and MessagePort. Although the asynchronous message passing can be good enough for generic tasks between threads, it is not suitable for real-time audio applications due to the overhead of (de)serialization and latency from task scheduling.

SharedArrayBuffer is designed to support the high speed data transfer between two threads, and it is a practical path for existing audio applications to migrate to the web platform with the minimum effort; the idea is to port native source code (C/C++) to WebAssembly and run it in the worker thread. Then the worker generates the audio data and the AudioWorkletProcessor takes the data to send it to the audio device. The data transfer between the worker and the AudioWorkletProcessor is done via SharedArrayBuffer that behaves like a FIFO or a ring buffer. (See figure 8.)

[5] https://www.w3.org/TR/secure-contexts/

Although very promising, SharedArrayBuffer is currently disabled by a majority of browser vendors as of February 2018 due to recent security vulnerabilities dubbed Meltdown and Spectre. [7] [8]

## 5. THE FUTURE OF WEB AUDIO

At the time of writing, W3C Audio Working Group is finalizing the working draft of Web Audio API to reach the Candidate Recommendation phase. Not only was the AudioWorklet the last missing piece of Web Audio API V1, but it will be an apt transition to the second iteration of the API. It is pure liberation from the developer's point of view as it unleashes sample-accurate audio manipulation with JavaScript and enables porting of much legacy audio software to the world of the web. This section discusses what the near future will look like with the introduction of AudioWorklet.

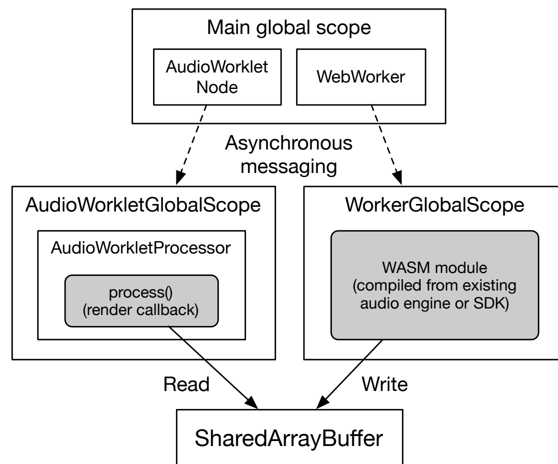


**Figure 8**. AudioWorklet, SharedArrayBuffer and WebWorker

### 5.1 WebAssembly

The rise of WebAssembly [6] in 2017 was a pivotal breakthrough in the web platform. The idea of high-performance JavaScript as a compile target was attractive enough to draw attention of several pioneers in the web audio community who took it to the next level.

As soon as AudioWorklet API surfaced on the experimental Chromium build, the Faust team started the transition to the AudioWorklet for their web-based compiler service. The tool chain compiles a Faust source code into WebAssembly with a few clicks and produces a fully interactive synth or effect on a web page. [9] Another example is Web Audio Module project [10] which demonstrated the idea of WebAssembly synthesizers by porting existing source codes. Lastly, the PedalBoard project presented a playground for effects and synths powered by AudioWorklet and WebAssembly by bringing all the pieces from Faust and Web Audio Module project into one place. [11]

### 5.2 Influx of legacy projects

SharedArrayBuffer, though currently disabled by all major browsers, is expected to be available soon once a proper security mitigation is in place. When that happens, we will naturally see more influx of legacy audio software or framework into the world of web audio.

Reusing code that has been tested and deployed over years is sensible in most cases, and translating existing audio engines for games (e.g., FMOD, Wwise) or cross-platform audio SDKs (e.g., Juce, STK) into WebAssembly can be done with minimal effort compared to a complete rewrite with JavaScript and Web Audio API. The cost of incompatible workflows and programming paradigms has been an obstacle for translating some of the existing projects, but SharedArrayBuffer and WebWorker can be the answer to the problem.

### 5.3 Ever-evolving web platform

The greatest benefit of being a part of the web platform is the seamless integration with numerous HTML5 APIs. The platform has been expanding its horizon to bleeding edge technologies like WebXR [7] , ServiceWorker [8] and Web MIDI API [9] while it is perfecting the foundations such as the network stack, the real-time communication, and the robust support for multimedia files. Along with such evolution, the gradual improvement in the browser core and the JS engine makes the web browser a viable destination for the music software.

The ecosystem around the web platform also shows a constant growth in its quality and quantity. A dependency management system like npm [10] , and build tools like Webpack [11] , Gulp [12] and Grunt [13] for streamlining workflow have a strong track record supporting software production on a massive scale. Also, projects like the headless Chrome and Puppeteer [14] make automated testing and continuous integration for web-based music applications easy. [12] Considering that the majority of audio SDK's are isolated and incompatible with each other, having a diverse and mature ecosystem will be a new-found luxury for audio programmers.

## 6. CONCLUSION

Web-based music software is not free from its fair share of skepticism. The web platform is still fighting an uphill battle to compete with native alternatives and in a large way we are pouring considerable effort into solving problems that are already solved: lower latency, less glitches, multi-threaded rendering, and so on. We should be willing to accept the fact that the web platform was not designed for the computer music but be optimistic in our belief that we can change the landscape through gradual progress.

What is the meaning of the AudioWorklet to the computer music community? First of all, it is a new technology for musical computation which has been meticulously specified and rigorously reviewed based on the W3C's process document. [13] Thanks to W3C Audio Working Group's

[6] http://webassembly.org/

[7] https://immersive-web.github.io/webxr/
[8] https://www.w3.org/TR/service-workers/
[9] http://webaudio.github.io/web-midi-api/
[10] https://www.npmjs.com/
[11] https://webpack.js.org/
[12] https://gulpjs.com/
[13] https://gruntjs.com/
[14] https://github.com/GoogleChrome/puppeteer

diligence over two years, AudioWorklet is now available to billions of users. Secondly, it forms a key ingredient that bridges between legacy music software and the web platform. The computer music can now take advantage of the rich and diverse ecosystem of the web, while the web platform will be more suited for embracing full-blown audio applications. To conclude, the AudioWorklet is an open invitation to the computer music community coming from the very wide world of the web platform.

**Acknowledgments**

Special thanks to Chris Wilson, Joshua Bell, Raymond Toy and Prof. Chris Chafe for their insightful feedback.

## 7. REFERENCES

[1] W3C Audio Working Group, "Web Audio API," https://webaudio.github.io/web-audio-api, 2018, [Online; accessed 26-February-2018].

[2] W3C Technical Architecture Group, "Web Audio API Design Review," https://github.com/w3ctag/design-reviews/blob/master/2013/07/WebAudio.md, 2014, [Online; accessed 26-February-2018].

[3] C. Wilson, "A Tale of Two Clocks - Scheduling Web Audio with Precision," https://www.html5rocks.com/en/tutorials/audio/scheduling/, 2013, [Online; accessed 26-February-2018].

[4] TC39, "ECMAScript 2019 Language Specification," https://tc39.github.io/ecma262, 2018, [Online; accessed 26-February-2018].

[5] WHATWG, "HTML Standard: Channel messaging," https://html.spec.whatwg.org/multipage/web-messaging.html#channel-messaging, 2018, [Online; accessed 26-February-2018].

[6] Mozilla, "WebAssembly - MDN," 2018, [Online; accessed 15-June-2018].

[7] Surma, "Meltdown/Spectre," https://developers.google.com/web/updates/2018/02/meltdown-spectre, 2018, [Online; accessed 26-February-2018].

[8] Graz University of Technology, "Meltdown and Spectre," https://meltdownattack.com, 2017, [Online; accessed 26-February-2018].

[9] S. Letz and Y. Orlarey, "WebAudio wasm benchmark pages and tools," http://faust.grame.fr/news/2017/12/12/benchmark-tools.html, 2017, [Online; accessed 26-February-2018].

[10] J. Kleimola and O. Larkin, "Web audio modules," *Proceedings of the Sound and Music Computing*, vol. 2015, 2015.

[11] M. Buffa, M. Demetrio, and N. Azria, "Guitar pedal board using WebAudio," *Web Audio Conference*, vol. 2016, 2016.

[12] E. Bidelman, "Automated testing with Headless Chrome," https://developers.google.com/web/updates/2017/06/headless-karma-mocha-chai, 2017, [Online; accessed 26-February-2018].

[13] W3C, "World Wide Web Consortium Process Document," https://www.w3.org/2018/Process-20180201/, 2018, [Online; accessed 26-February-2018].