DAW Plugins for Web Browsers

Jari Kleimola
Dept. of Media Technology
School of Science, Aalto University
Otaniementie 17, Espoo, Finland
jari.kleimola@aalto.fi

ABSTRACT

A large collection of Digital Audio Workstation (DAW) plugins is available on the Internet in open source form. This paper explores their reuse in browser environments, focusing on hosting options that do not require manual installation. Two options based on Emscripten and PNaCl are introduced, implemented, evaluated, and released as open source. We found that ported DAW effect and sound synthesizer plugins complement and integrate with the Web Audio API, and that the existing preset patch collections make the plugins readily usable in online contexts. The latency measures are higher than in native plugin implementations, but expected to reduce with the emerging AudioWorker node.

Categories and Subject Descriptors

Information systems~Browsers, *Applied computing~Sound and music computing*, *Software and its engineering~Real-time systems software*, Software and its engineering~Reusability

General Terms

Algorithms, Performance, Design, Standardization

Keywords

web browser, DAW plugin host, PNaCl, Emscripten, LADSPA, DSSI, VST

1. INTRODUCTION

HTML5 and related technologies continue to expand the browser sandbox. In aural domain, `<audio>` element provides a consistent application programming interface (API) and user experience for audio playback in web browsers. Web Audio API specifies further extensions for audio synthesis, processing, analysis and routing. The current API [1] defines 18 general purpose audio source, processing, and sink *nodes*. Each node implements a well-defined DSP algorithm in native form, and exposes its parameters via JavaScript API. Control rate parameters support automation and manual control using user interface (UI) elements, or external controller devices via the emerging Web MIDI API [7]. Nodes are often interconnected through audiorate parameter ports to form higher level digital signal processing (DSP) algorithms.

However, the set of native Web Audio API nodes is insufficient to cover all needs of highly focused DSP applications. Since the speed of contemporary JavaScript engines enable audiorate algorithm implementations in script, Web Audio API introduces a ScriptProcessor node (or its future AudioWorker reincarnation) for custom DSP development in JavaScript. Unfortunately, scripting support leads to inevitable performance penalties, which can be mitigated by using Mozilla's high performance *asm.js* [6] and *Emscripten* [12] virtual machine as an optimized runtime platform on top of JavaScript. Google's Portable Native Client (*PNaCl*) technology [4] provides another solution for accelerated custom DSP development. Instead of compiling C/C++ source code into a subset of JavaScript as in Emscripten, PNaCl employs bitcode modules that are embedded into the webpage, and translated into native executable form ahead of runtime. Emscripten/asm.js and PNaCl technologies work directly from the open web and do not require manual installation.

Since Emscripten and PNaCl are both based on C/C++ source code compilation, reuse of existing codebases in web platforms comes as an additional benefit. This is especially interesting for the web audio community: most open source DSP libraries are implemented in C/C++ and may therefore be used to extend the currently available native Web Audio API node set. Furthermore, a wealth of digital audio workstation (DAW) plugins also exists in open source. The reuse of DAW plugins would afford integration of high quality virtual effects devices and sound synthesizers in the Web Audio API node graph. Lower level DSP libraries and higher level DAW plugins would thereby promote the use of browsers as professional audio platforms.

This work focuses on enabling DAW plugins in web browsers. The plugins are hosted either as PNaCl or Emscripten modules, and integrated into the Web Audio API rendering pipeline using a ScriptProcessor node (SPN) backend. An alternative PNaCl hosting option routes audio directly into to the browser audio rendering pipeline for improved latency. Both hosting options are explored with detailed implementations, and evaluated in terms of latency and computational load. The tangible contributions of this paper are:

- a browser-based DAW plugin host. A web page may embed or load the plugins dynamically, control their parameters, and integrate them into the Web Audio API rendering pipeline.
- a collection of browser-hosted DAW plugins to i) complement the current native Web Audio API node set, and ii) to provide semantically higher level audio effect units and sound synthesizers for browser platforms.

The rest of this paper is structured as follows. Section 2 provides a look at related research. Sections 3 and 4 provide implementations and evaluations of PNaCl and Emscripten based plugin hosting options. Their implications are discussed in Section 5, and finally, Section 6 concludes. A demo page[1] and source code[2] are available in public domain.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
1st Web Audio Conference (WAC), January 26–27, 2015, Paris, France.

[1] https://mediatech.aalto.fi/publications/webservices/dawplugins

[2] https://github.com/jariseon/webdawplugins

2. RELATED WORK

2.1 Native Plugin Architectures

The first affordable digital hardware synthesizers and effects devices started to appear at the market in the early 1980's. In mid 1990's off-the-shelf desktop PCs were already powerful enough to perform real-time audio effects processing in software, and towards the end of the century, software sound synthesizers emerged as virtual re-incarnations of their hardware counterparts. The software effects units and instruments were either standalone applications or plugins inside a DAW host. The host capabilites were exposed as proprietary plugin APIs, including Apple's Audio Units (AU), Microsoft's DirectX instruments (DXi), and Steinberg's cross-platform Virtual Studio Technology (VST). In Linux, audio effects units were first interfaced with Linux Audio Developer's Simple Plugin API (LADSPA) [8], which was later extended with MIDI and Open Sound Control (OSC) support to give rise to the Disposable Soft Synth Interface (DSSI) [5]. The evolution progressed further into LADSPA v2 (LV2), which addresses the limitations of LADSPA and DSSI through an extensible API. This paper focuses on LADSPA, DSSI and VST2 plugin APIs.

2.2 Web Audio

At the start of new millennium, Java and Flash plugins enabled real-time audio processing and sound synthesis in web browsers. More recently, HTML5 `<audio>` element and the W3C Web Audio [1] and MediaStream JavaScript APIs have been adopted as standards for web audio development. Several third party frameworks have also emerged to increase the functionality of the platform. The frameworks rely either on native Web Audio API nodes (for increased performance), or use the ScriptProcessor node as a backend (for increased flexibility). The libraries and frameworks of the former category include tuna[3] and Pedalboard.js[4], which model guitar effects, while a dual oscillator analog synth[5] demonstrates a sound synthesizer built on top of TemplateSynth framework. The latter category includes Flocking[6], WAAX[7] [3] and Gibberish [11]. Flocking implements a set of JavaScript unit generators that are connected into synths and synthdefs similar to those in the SuperCollider environment. WAAX abstracts audio graphs as node-like units with condensed parameter access syntax. Gibberish provides low and high level unit generators with single sample processing model, and attempts to optimize the code with run-time code generation reminiscent of asm.js. The present paper achieves increased flexibility of SPN backends, but also gains improved performance by exploiting PNaCl and Emscripten accelerator technologies.

2.3 Accelerator Technologies

Google's native client (NaCl) provides a sandboxed environment for untrusted code execution in web browsers [4]. It defines a set of secure APIs that are accessible from JavaScript or C/C++ code. The API includes partial support for standard clib functions and other common libraries such as pthreads, and affords direct, albeit restricted low level access to the audio rendering pipeline of the browser. NaCl modules require manual installation from Chrome Web Store. In contrast, Portable native client (PNaCl) enables extension loading directly from the open web without manual installation [4]. PNaCl defines an instruction-set independent layer on top of NaCl. Its distribution format is low-level virtual machine (LLVM) bitcode, which is translated into native code at client side before execution. PNaCl thus enlarges the conventional browser sandbox with third party modules that achieve high performance while still remaining secure.

PNaCl modules are developed in C/C++. They are compiled and linked into LLVM bitcode by taking advantage of optimization strategies available in the toolchain[8]. All features of underlying NaCl API are available in PNaCl. For instance, JavaScript code can communicate with the PNaCl bitcode using asynchronous messages that carry marshalled payload. PNaCl thus allows reuse of previously written C/C++ libraries straight from the JavaScript front end code.

Mozilla's Emscripten [12] achieves a similar goal. However, instead of targeting LLVM bitcode, Emscripten toolchain takes an additional step: the intermediate LLVM assembly is optimized in compliance with contemporary JavaScript engines, and finally expressed in a high performance subset of JavaScript defined by asm.js [6]. The outcome is embedded inside a virtual machine also implemented in asm.js. Since the Emscripten target is JavaScript compliant, it can be downloaded and interpreted in any modern JavaScript engine.

Both accelerator techniques have already been used in web audio contexts. For instance, Faust defines a functional DSP language and a compiler targeting various DSP frameworks [10]. One of the targets produce vanilla C++ code with minimal boilerplate. This form has been successfully transformed into SPN compliant unit generators using Emscripten [2]. Furthermore, the Csound environment has been implemented in both Emscripten and PNaCl form [9], although the PNaCl implementation does not integrate with the Web Audio API. With minor modifications, the DAW plugin host produced in this work may also host the minimal boilerplate Faust unit generators as PNaCl modules, and enable the PNaCl Csound implementation to be integrated with the Web Audio API node graph.

3. IMPLEMENTATION

The primary goal of this work was to provide web compliant ports of LADSPA, DSSI and VST plugins, and enable their hosting in web browsers straight from the open web, i.e., without any client side installation requirements. The secondary goal was to strive towards high performance, and if possible, minimum latency by using PNaCl and Emscripten as implementation strategies.

3.1 Porting

The porting of existing DAW plugins involves (1) compiling the original plugin code into a PNaCl or Emscripten module, and (2) wrapping the module into a JavaScript class. The wrapper class integrates with Web Audio API, Web MIDI API, and HTML5 compliant UI. The wrapper also supports parameter and preset handling.

The compilation part (1) of LADSPA plugins is straightforward. The only issues requiring modifications in the original source code involved dynamic link library entry and exit points (`_init` and `_fini`), which conflicted with the PNaCl framework. These issues were resolved by simply compiling the source as C++ instead of C, or by renaming the conflicting entries. DSSI and

[3] https://github.com/Dinahmoe/tuna
[4] http://dashersw.github.io/pedalboard.js/
[5] http://webaudiodemos.appspot.com/midi-synth/
[6] https://github.com/colinbdclark/Flocking
[7] http://hoch.github.io/WAAX/

[8] https://developer.chrome.com/native-client/overview

VST plugin compilation is more involved because of increased functionality. For instance, the code for UI and preset handling needs to be ripped off and re-implemented in the JavaScript wrapper. VST also allows several entry point formats, which was resolved with preprocessor directives.

This work provides a proof-of-concept collection of DAW plugin ports. The LADSPA collection contains several effects processors to complement the Web Audio API node set. The DSSI collection includes Hexter (virtual Yamaha DX7 FM synthesizer clone), Xsynth (dual oscillator virtual analog emulation), and a hybrid WhySynth instrument. The VST2 collection includes AZR3 (Hammond organ emulation), mdaJX10 (another dual-oscillator virtual analog), and a sortiment of effects units. All synthesizers come with presets for instant use. Links to the original plugins are available in the demo page[1].

3.2 JavaScript Wrappers

To simplify the wrapping part (2), the host produced in this work provides boilerplate for audio and MIDI message routing, generic UI construction, bank and patch uploads, and plugin parameter accessors. Code Listing 1 demonstrates the proposed JavaScript API. The example sets up the plugin host (line 2), and loads a Hexter DSSI plugin asynchronously (line 9). Lines 3-8 implement a callback which is invoked once the plugin is ready. The callback first uploads a bank (line 4), selects the first patch of the bank using MIDI (line 5), and sets main volume (line 6). Line 7 inserts the plugin into the Web Audio API node graph, and finally, line 8 connects the plugin into a Web MIDI API input port. The synthesized audio buffers are routed into the node graph via an embedded ScriptProcessor node (SPN). An alternative PNaCl routing option, `plug.setParam("directOut",true)`, sends the audio buffers directly to the browser audio rendering pipeline for reduced latency. The complete API documentation is available in the source code repository[2].

Code Listing 1

```
1 var actx = new AudioContext();
2 var host = new DAWPluginHost(actx);
3 host.onpluginready = function (plug) {
4    plug.setBank(plug.factoryBank, 0);
5    plug.sendMidi(0xC0, 0, 0);
6    plug.setParam("Volume", 0.75);
7    plug.connect(actx.destination);
8    midiIn.onmidimessage = plug.onMidi.bind(plug); }
9 var synth = new Hexter(host);
```

Code Listing 2 demonstrates a simple wrapper implementation. As can be seen, the functionality of Code Listing 1 is achieved in large part by just relying on the boilerplate code, i.e., using standard prototypal inheritance from the `DAWPlugin` class (not depicted). The wrappers typically implement logic for custom preset parsing and UI implementation. In Code Listing 2, Line 2 installs an override for the `setBank` method invoked from line 4 of Code Listing 1. Hexter accepts original Yamaha DX7 sysex dumps, and uses the override to strip MIDI headers from the patch dump before passing the bank to the base class. Line 3 installs an override to extract the patch name from a custom patch structure. Finally, line 4 loads and creates the plugin, eventually triggering the `onpluginready` callback in Code Listing 1. The second parameter of `createPlugin` instructs the host to either load a PNaCl module (as indicated by the ".nmf" tail), or an Emscripten module (indicated by ".js"). The third parameter describes the index of the plugin in a LADSPA or DSSI bundle: LADSPA and DSSI modules are actually plugin libraries which may contain one or more distinct plugin implementations. The third parameter is unused in VST contexts.

Code Listing 2

```
1 Hexter = function (host) {
2   this.setBank = function (bank,i) { ... }
3   this.getPatchName = function (patch) { ... }
4   host.createPlugin(this, "hexter.nmf", 0); };
```

The complexity behind Code Listings 1 and 2 is hidden inside `DAWPluginHost` and `DAWPlugin` classes. `DAWPluginHost` is itself a thin wrapper that converges the PNaCl and Emscripten loading options into a single API (PNaCl modules are injected as `<embed>` and Emscripten modules as `<script>` tags). It also contains preset loading and parsing logic, including support for VST and AU format banks and patches.

`DAWPlugin` provides a bridge between the wrapper and the ported plugin implementation. The interface is implemented as RESTful asynchronous messaging in the PNaCl case, and as wrapped function calls in the Emscripten case. `DAWPlugin` is responsible for the creation of the plugin instance, SPN backend management, audio buffer routing between the node graph and the hosted plugin, MIDI routing, and uploading of preset banks, patches, and parameters. The generic UI support is built on top of `DAWPlugin` as well.

3.3 PNaCl Host

The block diagram of PNaCl form DAW plugin hosting is shown in Figure 1. The host resides partly on JavaScript side (see Section 3.2), and partly on PNaCl side. The PNaCl side consists of `PDAWHost` module instance, and the shared `DAWHost`, `DAWPlugin` and `DAWBundle` classes. `PDAWHost` provides the asynchronous communication end point for the JavaScript wrapper, and parses and marshalls the received messages into a format that is common to both PNaCl and Emscripten implementations. Audio buffers are created at the JavaScript side and transferred as ArrayBuffers. Audio needs to be double-buffered because of thread boundary between JavaScript wrapper and PNaCl module, and because of the resulting asynchronous messaging paradigm. The marshalled messages are handed over to the `DAWHost` class.

`DAWHost`, `DAWPlugin` and `DAWBundle` classes implement the actual hosting functionality. `DAWHost` manages the plugin instances, and routes the received messages to targeted plugins, and back to `PDAWHost` if requested. `DAWPlugin` and `DAWBundle` are abstract classes that define a common API for different plugin formats.

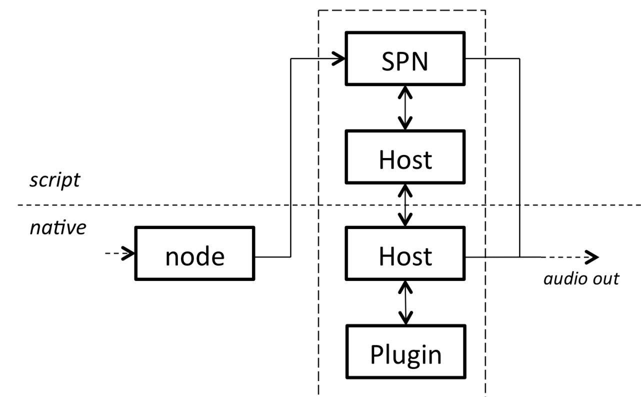

Figure 1. DAW plugins as PNaCl modules.

DAWPlugin class is inherited by LadspaPlugin, which again is a superclass of DSSIPlugin class. DAWPlugin provides a base for VSTPlugin as well. DAWPlugin initializes and activates plugin instances, provides parameter accessors, manages input and output audio buffers, and provides an entry point to the process method that contains the actual DSP code. DSSIPlugin and VSTPlugin contain additional methods for MIDI input and preset handling.

DAWBundle is inherited by LadspaBundle, which in turn is a superclass of DSSIBundle. The bundle classes initialize the plugin library, collect descriptors of each contained plugin type into a JSON string that is returned back to the JavaScript side, and dispose the library once the PNaCl instance is destroyed.

In summary, the end-to-end audio processing in PNaCl hosting works as follows. At JavaScript side, the DAWPlugin wrapper embeds an SPN backend, which periodically pulls sample buffers from the PNaCl implementation. PDAWHost receives the pull request and marshalls the passed ArrayBuffers into C float arrays. The arrays are then handed over to the DAWHost class for routing to a proper VSTPlugin, DSSIPlugin, or LadspaPlugin container. The container finally calls the hosted plugin implementation, which computes the requested buffer. Because of the thread boundary between JavaScript and PNaCl module, the SPN pull request is asynchronous. Therefore, PDAWHost needs to double buffer the samples, and for each pull request, return the buffer processed during a previous call.

PNaCl host also allows direct connection with the browser audio rendering pipeline. This link is enabled by Pepper API's Audio class, which is interfaced in PDAWHost in a similar way as the SPN backed endpoint. The difference is, however, that the call is synchronouse and that the buffer size is smaller.

3.4 Emscripten Host

The block diagram of Emscripten form DAW plugin hosting is shown in Figure 2. In this case, the host resides entirely at JavaScript side, but is still divided into a wrapper part (see Section 3.2), and an Emscripten part. The latter interfaces the virtual maching using wrapped function calls and direct memory access. As in PNaCl hosting, the Emscripten part comprises DAWHost, and abstract DAWPlugin and DAWBundle classes.

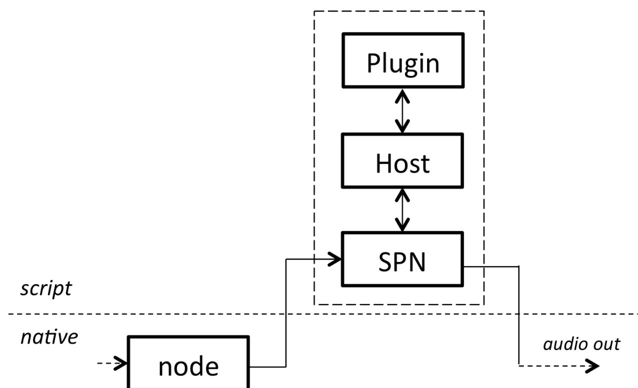

Figure 2. DAW plugins as Emscripten modules.

Messaging between the JavaScript wrapper and Emscripten part is now synchronous, which means that the audio does not require double buffering. The audio buffers are allocated inside the Emscripten part, and the JavaScript wrapper has direct access to the buffers. Plugin may not connect directly into the browser audio rendering pipeline however, since all audio data needs to be routed through the SPN backend. In addition, message payload does not need to be marshaled as in PNaCl case.

Emscripten also allows partial support for dynamic loading of libraries, and a dlopen() function was initially included in the Emscripten part. The implementation employs XmlHttpRequest in plugin library loading. Library entry and exit points can then be retrieved and invoked. Although this method proved succesfull, the amount of manual intervention was considered impractical and dynamic loading was left out of the contribution of this paper.

4. EVALUATION

PNaCl and Emscripten hosting solutions were evaluated in terms of latency and performance. The evaluation was done in MacBook Pro running OSX 10.9.5 (Mavericks), and Google Chrome v38 and Firefox v32 browsers. Since PNaCl is only available in Chrome (Pepper API v37), Firefox was only used in Emscripten (v1.25.0) based hosting solution.

The theoretical latencies are given in Table 1. According to Web Audio API spec, the minimum SPN buffer size is 256 samples. This produced audible artifacts, and therefore the buffer size was increased to 512 samples. 512 samples equals 11.6 ms at 44.1 kHz sample rate, totaling 23.2 ms when double-buffered. PNaCl hosting requires another thread hop and another double buffer. The direct connection into the browser audio rendering pipeline using Pepper Audio API enabled 128 sample buffer size, which equals 2.9 ms latency at 44.1 kHz.

Table 1. Theoretical latencies and measured polyphony.

Hosting Type	samples	ms	polyphony
PNaCl (SPN)	2048	46.4	350
Emscripten (SPN)	1024	23.2	260
PNaCl (direct)	128	2.9	435

End-to-end latencies were measured by triggering a percussive plugin timbre from an external USB-MIDI keyboard, and using embedded laptop microphone to capture mechanical key clicks and synthesized sounds. The span between the onsets was then examined in audio waveform editor. The measured end-to-end latencies were 20-30 ms higher than the theoretical values, regardless of hosting type.

The performance was evaluated as the number of polyphonic voices. A sinusoidal oscillator bank implemented as a LADSPA plugin was used in the evaluation. The number of simultaneous oscillators was increased until audible artifacts started to appear (buffer sizes of 512 and 128 samples were used for SPN and direct audio rendering, respectively). Chrome and Firefox did not have a significant difference in Emscripten performance evaluation. The maximum number of sinusoidal oscillators still producing artifact free audio is given in the rightmost column of Table 1.

5. DISCUSSION

The theoretical latency figures for direct PNaCl rendering are excellent, and the synthesis engine felt responsive in that case. The latencies of SPN-backed hosting types were noticeable, and required adaptation in playing technique. SPN-backed methods were unfortunately considered too unresponsive for precise

performative control. However, part of the perceived latency may be attributed to the control mechanism, which added a constant 20-30 ms to the theoretical values. Since the buffer sizes did not have an effect on the added latency, we argue that the controller device key action mechanics combined with MIDI message routing overhead in the browser gives raise to the increased measure. The emerging AudioWorker node with synchronous callbacks and reduced buffer sizes is expected to improve the perfomative experiences. We would also like to see direct audio routing from PNaCl modules into the AudioWorker node.

The polyphony measures look promising. As expected, PNaCl outperformed Emscripten, but nevertheless, scripted performance is still impressive. It should be noted that direct PNaCl polyphony was measured with 128 sample buffering, and polyphony is expected to increase with larger buffer sizes.

The size of a ported DAW plugin is fairly large because of embedded virtual machine and supporting library code. Pluginless host containing only DSSI scaffolding code weighs 312 kB in PNaCl, and 612 kB in Emscripten. Full Hexter DSSI plugin implementation adds only 128 kB to the scaffolding PNaCl host size. Dynamically linked libraries would be thus beneficial in reducing the download size of the plugins. Another option is to bundle several different plugins into a single LADSPA or DSSI module.

Although PNaCl and Emscripten share several concepts, they also differ in many aspects. The choice between the two targets depends on deployment and application specific preferences. Emscripten targets are compatible with all modern JavaScript engines, whereas PNaCl modules run only within Chrome. On the other hand, PNaCl supported pthreads are at the time of writing unsupported in Emscripten. In addition, smaller download size, higher performance, low latency in direct mode, and effortless JavaScript/C++ binding of pepper messaging API make PNaCl targets attractive. However, the emerging AudioWorker node will favor Emscripten targets if the PNaCl audio I/O is omitted from the AudioWorker interface.

Code Listing 1 demonstrates the asynchronous nature of PNaCl communication. This may complicate certain implementations especially when many plugins are loaded and connected into the audio graph. Promises were found extremely useful in this context. Finally, since DAW plugins are more complex than lower level DSP libraries, the produced host may be easily modified to support existing DSP libraries as well.

6. CONCLUSION

This paper explored the use of LADSPA, DSSI and VST DAW plugins in web browsers. The architecture of plugin hosting options for PNaCl and Emscripten accelerators was designed, implemented and evaluated. The evaluation revealed that PNaCl plugins contributed less latency when connected directly into the browser audio rendering pipeline, but Emscripten plugins outperformed them in terms of latency when integrated into Web Audio API rendering pipeline. On the other hand, PNaCl implementation was found to be more efficient in terms of polyphony because of native execution. The emerging AudioWorker node was expected to become beneficial for DAW plugin hosting in open web. In addition, a direct link between PNaCl modules and AudioWorker node was suggested.

We foresee that browser hosted DAW plugins will provide usable extensions to the current web audio processing stack. The plugins complement the existing Web Audio API node set, and provide semantically higher level components for musical applications and audiovisual installations. Future work includes AudioWorker integration and PNaCl plugin pipeline exploration. Improved GUI support and full MIDI integration are other future improvements. Finally, support for LV2 and JUCE plugins is planned in near future.

7. REFERENCES

[1] Adenot, P., Wilson, C., and Rogers, C. 2013. *Web Audio API*. W3C Working Draft, Oct 10, 2013. Available online at http://www.w3.org/TR/webaudio/. (editor's draft at http://webaudio.github.io/web-audio-api/).

[2] Borins, M. 2014. From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten. in *Proc. Linux Audio Developers' Conference* (LAC-2014).

[3] Choi, H. and Berger, J. 2013. "WAAX: Web Audio API eXtension." In *Proc. New Interfaces for Musical Expression* (NIME'13), pp. 499–502.

[4] Donovan, A., Muth, R., Chen, B., and Sehr, D. 2010. PNaCl: Portable Native Client Executables. White paper, Feb. 22, 2010. Website at https://developer.chrome.com/native-client/overview

[5] *DSSI* – Disposable Soft Synth Interface, homepage, http://dssi.sourceforge.net

[6] Herman, D., Wagner, L: and Zakai, A. 2014. *asm.js*. Working Draft, Aug. 18, 2014. Available online at http://asmjs.org/spec/latest/.

[7] Kalliokoski, J. and Wilson, C. 2013. *Web Midi API*. W3C Working Draft, Nov. 26, 2013. Available online at http://www.w3.org/TR/webmidi/. (editor's draft at http://webaudio.github.com/web-midi-api/).

[8] *LADSPA* – Linux Audio Developer's Simple Plugin API, homepage, http://www.ladspa.org

[9] Lazzarini, V., Costello, E., Yi, S., and Fitch, J. 2014. Csound on the Web. in Proc. Linux Audio Developers' Conference (LAC-2014).

[10] Orlarey, Y., Fober, D., and Letz, S. 2009. Faust : an efficient functional approach to DSP programming. *New Computational Paradigms for Computer Music*, pp. 65–96.

[11] Roberts, C., Wakefield, G., and Wright, M. 2013. The Web Browser as Synthesizer and Interface. In *Proc. New Interfaces for Musical Expression* (NIME'13), pp. 313–318.

[12] Zakai, A. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proc. ACM Int. Conf. companion on Object oriented programming systems languages and applications companion* (OOPSLA '11). ACM, New York, NY, USA, pp. 301-312. Website at http://kripken.github.io/emscripten-site/