Csound on the Web

Victor LAZZARINI and **Edward COSTELLO** and **Steven YI** and **John FITCH**
Department of Music
National University of Ireland
Maynooth,
Ireland,
{victor.lazzarini@nuim.ie, edwardcostello@gmail.com, stevenyi@gmail.com, jpff@codemist.co.uk }

Abstract

This paper reports on two approaches to provide a general-purpose audio programming support for web applications based on Csound. It reviews the current state of web audio development, and discusses some previous attempts at this. We then introduce a Javascript version of Csound that has been created using the Emscripten compiler, and discuss its features and limitations. In complement to this, we look at a Native Client implementation of Csound, which is a fully-functional version of Csound running in Chrome and Chromium browsers.

Keywords

Music Programming Languages; Web Applications;

1 Introduction

The web browser has become an increasingly viable platform for the creation and distribution of various types of media computing applications[Wyse and Subramanian, 2013]. It is no surprise that audio is an important part of these developments. For a good while now we have been interested in the possibilities of deployment of client-side Csound-based applications, in addition to the already existing server-side capabilities of the system. Such scenarios would be ideal for various uses of Csound. For instance, in Education, we could see the easy deployment of Computer Music training software for all levels, from secondary schools to third-level institutions. For the researcher, web applications can provide an easy means of creating prototypes and demonstrations. Composers and media artists can also benefit from the wide reach of the internet to create portable works of art. In summary, given the right conditions, Csound can provide a solid and robust general-purpose audio development environment for a variety of uses. In this paper, we report on the progress towards supporting these conditions.

2 Audio Technologies for the Web

The current state of audio systems for world-wide web applications is primarily based upon three technologies: Java[1], Adobe Flash[2], and HTML5 Web Audio[3]. Of the three, Java is the oldest. Applications using Java are deployed via the web either as Applets[4] or via Java Web Start[5]. Java as a platform for web applications has lost popularity since its introduction, primarily due to historically sluggish start-up times as well as concerns over security breaches. Also of concern is that major browser vendors have either completely disabled Applet loading or disabled them by default, and that NPAPI plugin support, with which the Java plugin for browsers is implemented, is planned to be dropped in future browser versions[6]. While Java sees strong support on the server-side and desktop, its future as a web-deployed application is tenuous at best and difficult to recommend for future audio system development.

Adobe Flash as a platform has seen large-scale support across platforms and across browsers. Numerous large-scale applications have been developed such as AudioTool[7], Patchwork[8], and Noteflight[9]. Flash developers can choose to deploy to the web using the Flash plugin, as well as use Adobe Air[10] to deploy to desktop and mobile devices. While these applications demonstrate what can be developed for the web

[1] http://java.oracle.com
[2] http://www.adobe.com/products/flashruntimes.html
[3] http://www.w3.org/TR/webaudio/
[4] http://docs.oracle.com/javase/tutorial/deployment/applet/index.html
[5] http://docs.oracle.com/javase/tutorial/deployment/webstart/index.html
[6] http://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html
[7] http://www.audiotool.com/
[8] http://www.patchwork-synth.com
[9] http://www.noteflight.com
[10] http://www.adobe.com/products/air.html

using Flash, the Flash platform itself has a number of drawbacks. The primary tools for Flash development are closed-source, commercial applications that are unavailable on Linux, though open source Flash compilers and IDEs do exist[11]. There has been a backlash against Flash in browsers, most famously by Steve Jobs and Apple[12], and the technology stack as a whole has seen limited development with the growing popularity of HTML5. At this time, Flash may be a viable platform for building audio applications, but the uncertain future makes it difficult to recommend.

Finally, HTML5 Web Audio is the most recent of technologies for web audio applications. Examples include the "Recreating the sounds of the BBC Radiophonic Workshop using the Web Audio API" site[13], Gibberish[14], and WebPd[15]. Unlike Java or Flash, which are implemented as browser plug-ins, the Web Audio API is a W3C proposed standard that is implemented by the browser itself.[16] Having built-in support for Audio removes the security issues and concerns over the future of plug-ins that affect Java and Flash. However, the Web Audio API has limitations that will be explored further below in the section on Emscripten.

3 Csound-based Web Application Design

Csound is a music synthesis system that has roots in the very earliest history of computer music. Csound use in Desktop and Mobile applications has been discussed previously in [Lazzarini et al., 2012b], [Yi and Lazzarini, 2012], and [Lazzarini et al., 2012a].

Prior to the technologies presented this paper, Csound-based web applications have employed Csound mostly on the server-side. For example, NetCsound [17] allows sending a CSD file to the server, where it would render the project to disk and email the user a link to the rendered file when complete. Another use of Csound on the server is Oeyvind Brandtsegg's VLBI Music [18], where Csound is running on the server and publishes its audio output to an audio stream that end users can listen to. A similar architecture is found in [Johannes and Toshihiro, 2013]. Since version 6.02, Csound also includes a built-in server, that can be activated through an option on start up. The server is able to receive code directly through UDP connections and compile them on the fly.

Using Csound server-side has both positives and negatives that should be evaluated for a project's requirements. It can be appropriate to use if the project's design calls for a single audio stream/Csound instance that is shared by all listeners. In this case, users might interact with the audio system over the web, at the expense of network latency. Using multiple realtime Csound instances, as would be the case if there was one per user, would certainly be taxing for a single server and would require careful resource limiting. For multiple non-realtime Csound instances, as in the case of NetCsound, multiple jobs may be scheduled and batch processed with less problems than with realtime systems, though resource management is still a concern.

An early project to employ client-side audio computation by Csound was described in [Casey and Smaragdis, 1996], where a sound and music description system was proposed for the rendering of network-supplied data streams. A possibly more flexible way to use Csound in client-side applications, however, is to use the web browser as a platform. Two attempts at this have been made in the past. The first was the now-defunct ActiveX Csound (also known as AXCsound)[19], which allowed embedding Csound into a webpage as an ActiveX Object. This technology is no longer maintained and was only available for use on Windows with Internet Explorer. A second attempt was made in the Mobile Csound Project[Lazzarini et al., 2012b], where a proof-of-concept Csound-based application was developed with Java and deployed using Java Web Start, achieving client-side Csound use via the browser. However, the technology required special permissions to run on the client side and required Java to be installed. Due to those issues and the unsure future of Java over the web,

[11] `http://www.flashdevelop.org/`

[12] `http://www.apple.com/hotnews/thoughts-on-flash/`

[13] `http://webaudio.prototyping.bbc.co.uk/`

[14] Available at `https://github.com/charlieroberts/Gibberish`, discussed in [Roberts et al., 2013]

[15] `https://github.com/sebpiq/WebPd`

[16] `http://caniuse.com/audio-api` lists current browsers that support the Web Audio API

[17] Available at `http://dream.cs.bath.ac.uk/netcsound/`, discussed in [ffitch et al., 2007]

[18] `http://www.researchcatalogue.net/view/55360/55361`

[19] We were unable to find a copy of this online, but one is available from the CD-ROM included with [Boulanger, 2000]

the solution was not further explored.

The two systems described in this paper are browser-based solutions that run on the client-side. The both share the following benefits:

- Csound has a large array of signal processing opcodes made immediately available to web-based projects.
- They are compiled using the same source code as is used for the desktop and mobile version of Csound. They only require recompiling to keep them in sync with the latest Csound features and bug fixes.
- Csound code that can be run with these browser solutions can be used on other platforms. Audio systems developed using Csound code is then cross-platform across the web, desktop, mobile, and embedded systems (i.e. Raspberry Pi, Beaglebone; discussed in [Batchelor and Wignall, 2013]). Developers can reuse their audio code from their web-based projects elsewhere, and vice versa.

4 Emscripten

Emscripten is a a project created by Alon Zakai at the Mozilla Foundation that compiles the assembly language used by the LLVM compiler into Javascript [Zakai, 2011]. When used in combination with LLVM's Clang frontend, Emscripten allows applications written in C/C++ or languages that use C/C++ runtimes to be run directly in web browsers. This eliminates the need for browser plugins and takes full advantage of web standards that are already in common use.

In order to generate Javascript from C/C++ sourcecode the codebase is first compiled into LLVM assembly language using LLVM's Clang frontend. Emscripten translates the resulting LLVM assembly language into Javascript, specifically an optimised subset of Javascript entitled asm.js. The asm.js subset of Javascript is intended as a low-level target language for compilers and allows a number of optimisations which are not possible with standard Javascript[20]. Code semantics which differ between Javascript and LLVM assembly can be emulated when accurate code is required. Emscripten has built-in methods to check for arithmetic overflow, signing issues and rounding errors. If emulation is not required, code can be translated without semantic emulation in order to achieve the best execution performance [Zakai, 2011].

Implementations of the C and C++ runtime libraries have been created for applications compiled with Emscripten. These allow programs written in C/C++ to transparently perform common tasks such as using the file system, allocating memory and printing to the console. Emscripten allows a virtual filesystem to be created using its FS library, which is used by Emscripten's libc and libcxx for file I/O[21]. Files can be added or removed from the virtual filesystem using Javascript helper functions. It is also possible to directly call C functions from Javascript using Emscripten[22]. These functions must first be named at compile time so they are not optimised out of the resulting compiled Javascript code. The required functions are then wrapped using Emscripten's *cwrap* function, and assigned to a Javascript function name. The *cwrap* function allows many Javascript variables to be used transparently as arguments to C functions, such as passing Javascript strings to functions which require the C languages *const char* array type.

Although Emscripten can successfully compile a large section of C/C++ code there are still a number of limitations to this approach due to limitations within the Javascript language and runtime. As Javascript doesn't support threading, Emscripten is unable to compile codebases that make use of threads. Some concurrency is possible using web workers, but they do not share state. It is also not possible to directly implement 64-bit integers in Javascript as all numbers are represented using 64-bit doubles. This results in a risk of rounding errors being introduced to the compiled Javascript when performing arithmetic operations with 64-bit integers [Zakai, 2011].

4.1 CsoundEmscripten

CsoundEmscripten is an implementation of the Csound language in Javascript using the Emscripten compiler. A working example of CsoundEmscripten can be found at `http://eddyc.github.io/CsoundEmscripten/`. The compiled Csound library and *CsoundObj* Javascript class can be found at `https://github.com/eddyc/CsoundEmscripten/`. CsoundEmscripten con-

[20] `http://asmjs.org/spec/latest/`

[21] `https://github.com/kripken/emscripten/wiki/Filesystem-API`

[22] `https://github.com/kripken/emscripten/wiki/Interacting-with-code`

sists of three main modules:

- The Csound library compiled to Javascript using Emscripten.
- A structure and associated functions written in C named *CsoundObj* implemented on top of the Csound library that is compiled to Javascript using Emscripten.
- A handwritten Javascript class also named *CsoundObj* that contains the public interface to CsoundEmscripten. The Javascript class both wraps the compiled *CsoundObj* structure and associated functions, and connects the Csound library to the Web Audio API.

4.1.1 Wrapping the Csound C API for use with Javascript

In order to simplify the interface between the Csound C API and the Javascript class containing the CsoundEmscripten public interface, a structure named *CsoundObj* and a number of functions which use this structure were created. The structure contains a reference to the current instance of Csound, a reference to Csound's input and output buffer, and Csound's 0dBFS value. Some of the functions that use this structure are:

- `CsoundObj_new()` - This function allocates and returns an instance of the *CsoundObj* structure. It also initialises an instance of Csound and disables Csound's default handling of sound I/O, allowing Csound's input and output buffers to be used directly.
- `CsoundObj_compileCSD(self, filePath, samplerate, controlrate, buffersize)` - This function is used to compile CSD files, it takes as its arguments: a pointer to the *CsoundObj* structure *self*, the address of a CSD file given by *filePath*, a specified sample rate given by *samplerate*, a specified control rate given by *controlrate* and a buffer size given by *buffersize*. The CSD file at the given address is compiled using these arguments.
- `CsoundObj_process(self, inNumberFrames, inputBuffer, outputBuffer)` - This function copies audio samples to Csound's input buffer and copies samples from Csound's output buffer. It takes as its arguments: a pointer to the *CsoundObj* structure *self*, an integer *inNumberFrames* specifying the number of samples to be copied, a pointer to a buffer containing the input samples named *inputBuffer* and a pointer to a destination buffer to copy the output samples named *outputBuffer*.

Each of the other functions that use the *CsoundObj* structure simply wrap existing functions present in the Csound C API. The relevant functions are:

- `csoundGetKsmps(csound)` - This function takes as its argument a pointer to an instance of Csound and returns the number of specified audio frames per control sample.
- `csoundGetNchnls(csound)` - This function takes as its argument a pointer to an instance of Csound and returns the number of specified audio output channels.
- `csoundGetNchnlsInput(csound)` - This function takes as its argument a pointer to an instance of Csound and returns the number of specified audio input channels.
- `csoundStop(csound)` - This function takes as its argument a pointer to an instance of Csound stops the current performance pass.
- `csoundReset(csound)` - This function takes as its argument a pointer to an instance of Csound and resets its internal memory and state in preparation for a new performance.
- `csoundSetControlChannel(csound, name, val)` - This function takes as its arguments: a pointer to an instance of Csound, a string given by *name*, and number given by *val*, it sets the numerical value of a Csound control channel specified by the string *name*.

The *CsoundObj* structure and associated functions are compiled to Javascript using Emscripten and added to the compiled Csound Javascript library. Although this is not necessary, keeping the compiled *CsoundObj* structure and functions in the same file as the Csound library makes it more convenient when including CsoundEmscripten within web pages.

4.1.2 The CsoundEmscripten Javascript interface

The last component of CsoundEmscripten is the *CsoundObj* Javascript class. This class provides the public interface for interacting with the compiled Csound library. As well as allocating an instance of Csound this class provides methods for controlling performance and setting the values of Csound's control channels. Additionally, this class interfaces with the Web Audio API, providing Csound with samples from the audio input bus and copying samples from Csound to the audio output bus. Audio I/O and the Csound process are performed in Javascript using the Web Audio API's *ScriptProcessorNode*. This node allows direct access to input and output samples in Javascript allowing audio processing and synthesis using the Csound library.

Csound can be used in any webpage by creating an instance of *CsoundObj* and calling the available public methods in Javascript. The methods available in the *CsoundObj* class are:

- `compileCSD(fileName)` This method takes as its argument the address of a CSD file *fileName* and compiles it for performance. The CSD file must be present in Emscripten's virtual filesystem. This method calls the compiled C function *CsoundObj_compileCSD*. It also creates a *ScriptProcessorNode* instance for Audio I/O.
- `enableAudioInput()` This method enables audio input to the web browser. When called, it triggers a permissions dialogue in the host web browser requesting permission to allow audio input. If permission is granted, audio input is available for the running Csound instance.
- `startAudioCallback()` This method connects the *ScriptProcessorNode* to the audio output and, if required, the audio input. The *ScriptProcessorNodes* audio processing callback is also started. During each callback, if required, audio samples from the *ScriptProcessorNodes* input are copied into Csound's input buffer and any new values for Csound's software channels are set. Csound's *csoundPerformKsmps()* function is called and any output samples are copied into the *ScriptProcessorNodes* output buffer.
- `stopAudioCallback()` This method disconnects the current running *ScriptProcessorNode* and stops the audio process callback. If required this method also disconnects any audio inputs.
- `addControlChannel(name, initialValue)` This method adds an object to a Javascript array that is used to update Csound's named channel values. Each object contains a string value given by *name*, a float value given by *initialValue* and additionally a boolean value indicating whether the float value has been updated.
- `setControlChannelValue(name, value)` This method sets a named control channel given by the string *name* to the specified number given by the *value* argument.
- `getControlChannelValue(name)` This method returns the current value of a named control channel given by the string *name*.

4.1.3 Limitations

Using CsoundEmscripten, it is possible to add Csound's audio processing and synthesis capabilities to any web browser that supports the Web Audio API. Unfortunately this approach of bringing Csound to the web comes with a number of drawbacks.

Although Javascript engines are constantly improving in speed and efficiency, running Csound entirely in Javascript is a processor intensive task on modern systems. This is especially troublesome when trying to run even moderately complex CSD files on mobile computing devices.

Another limitation is due to the design of the *ScriptProcessorNode* part of the Web Audio API. Unfortunately, the *ScriptProcessorNode* runs on the main thread. This can result in audio glitching when another process on the main thread—such as the UI—causes a delay in audio processing. As part of the W3Cs Web Audio Spec review it has been suggested that the *ScriptProcessorNode* be moved off of the main thread[23]. There has also been a resolution by the Web Audio API developers that they will make it possible to use the *ScriptProcessorNode* with web workers[24]. Hopefully in a future version of the Web Audio API the *ScriptProcessorNode* will be more capable of running the

[23] `https://github.com/w3ctag/spec-reviews/blob/master/2013/07/WebAudio.md#issue-scriptprocessornode-is-unfit-for-purpose-section-1`

[24] `https://www.w3.org/Bugs/Public/show_bug.cgi?id=17415#c94`

kind complex audio processing and synthesis capabilities allowed by the Csound library.

This version of Csound also does not support plugins, making some opcodes unavailable. Additionally, MIDI I/O is not currently supported. This is not due to the technical limitations of Emscripten, rather it was not implemented due to the current lack of support for the WebMIDI standard in Mozilla Firefox[25] and in the Webkit library[26].

5 Beyond Web Audio: Creating Audio Applications with PNaCl

As an alternative to the development of audio applications for web deployment in pure Javascript, it is possible to take advantage of the Native Clients (NaCl) platform[27]. This allows the use of C and C++ code to create components that are accessible to client-side Javascript, and run natively inside the browser. NaCl is described as a sandboxing technology, as it provides a safe environment for code to be executed, in an OS-independent manner [Yee et al., 2009] [Sehr et al., 2010]. This is not completely unlike the use of Java with the Java Webstart Technology (JAWS), which has been discussed elsewhere in relation to Csound [Lazzarini et al., 2012b].

There are two basic toolchains in NaCl: native/gcc and PNaCl [Donovan et al., 2010]. While the former produces architecture-dependent code (arm, x86, etc.), the latter is completely independent of any existing architecture. NaCl is currently only supported by the Chrome and Chromium browsers. Since version 31, Chrome enables PNaCl by default, allowing applications created with that technology to work completely out-of-the-box. While PNaCl modules can be served from anywhere in the open web, native-toolchain NaCl applications and extensions can only be installed from Google's Chrome Web Store.

5.1 The Pepper Plugin API

An integral part of NaCl is the Pepper Plugin API (PPAPI, or just Pepper). It offers various services, of which interfacing with Javascript and accessing the audio device is particularly relevant to our ends. All of the toolchains also include support for parts of the standard C library (eg. stdio), and very importantly for Csound, the pthread library. However, absent from the PNaCl toolchain are dlopen() and friends, which means no dynamic loading is available there.

Javascript client-side code is responsible for requesting the loading of a NaCl module. Once the module is loaded, execution is controlled through Javascript event listeners and message passing. A postMessage() method is used by Pepper to allow communication from Javascript to PNaCl module, triggering a message handler in the C/C++ side. In the opposite direction, a *message* event is issued when C/C++ code calls the equivalent PostMessage() function.

Audio output is well supported in Pepper with a mid-latency callback mechanism (ca. 10-11ms, 512 frames at 44.1 or 48 KHz sampling rate). Its performance appears to be very uniform across the various platforms. The Audio API design is very straightforward, although the library is a little rigid in terms of parameters. It supports only stereo at one of the two sampling rates mentioned above). Audio input is not yet available in the production release, but support can already be seen in the development repository.

The most complex part of NaCl is access to the local files. In short, there is no open access to the client disk, only to sandboxed filesystems. It is possible to mount a server filesystem (through httpfs), a memory filesystem (memfs), as well as local temporary or permanent filesystems (html5fs). For those to be useful, they can only be mounted and accessed through the NaCl module, which means that any copying of data from the user disk into these partitions has to be mediated by code written in the NaCl module. For instance, it is possible to take advantage of the file HTML5 tag and to get data from NaCl into a Javascript blob so that it can be saved into the user's disk. It is also possible to copy a file from disk into the sandbox using the URLReader service supplied by Pepper.

5.2 PNaCl

The PNaCl toolchain compiles code down to a portable bitcode executable (called a *pexe*). When this is delivered to the browser, an ahead-of-time compiler is used to translate the code into native form. A web application using PNaCl will contain three basic components: the pexe binary, a manifest file describing it, and a client-side script in JS, which loads and allows interaction with the module via the Pepper messaging

[25] `https://bugzilla.mozilla.org/show_bug.cgi?id=836897`

[26] `https://bugs.webkit.org/show_bug.cgi?id=107250`

[27] `https://developers.google.com/native-client`

system.

5.3 Csound for PNaCl

A fully functional implementation of Csound for Portable Native Clients is available from `http://vlazzarini.github.io`. The package is composed of three elements: the Javascript module (csound.js), the manifest file (csound.nmf), and the pexe binary (csound.pexe). The source for the PNaCl component is also available from that site (csound.cpp). It depends on the Csound and Libsndfile libraries compiled for PNaCl and the NaCL sdk. A Makefile for PNaCl exists in the Csound 6 sources.

5.3.1 The Javascript interface

Users of Csound for PNaCl will only interact with the services offered by the Javascript module. Typically an application written in HTML5 will require the following elements to use it:

- the csound.js script
- a reference to the module using a div tag with id="engine"
- a script containing the code to control Csound.

The script will contain calls to methods in csound.js, such as:

- `csound.Play()` - starts performance
- `csound.PlayCsd(s)` - starts performance from a CSD file *s*, which can be in ./http/ (ORIGIN server) or ./local/ (local sandbox).
- `csound.RenderCsd(s)` - renders a CSD file *s*, which can be in ./http/ (ORIGIN server) or ./local/ (local sandbox), with no RT audio output. The "finished render" message is issued on completion.
- `csound.Pause()` - pauses performance
- `csound.CompileOrc(s)` - compiles the Csound code in the string *s*
- `csound.ReadScore(s)` - reads the score in the string *s* (with preprocessing support)
- `csound.Event(s)` - sends in the line events contained in the string *s* (no preprocessing)
- `csound.SetChannel(name, value)` - sends the control channel *name* the value *value*, both arguments being strings.

As it starts, the PNaCl module will call a `moduleDidLoad()` function, if it exists. This can be defined in the application script. Also the following callbacks are also definable:

- `function handleMessage(message)`: called when there are messages from Csound (pnacl module). The string *message.data* contains the message.
- `function attachListeners()`: this is called when listeners for different events are to be attached.

In addition to Csound-specific controls, the module also includes a number of filesystem facilities, to allow the manipulation of resources in the server and in the sandbox:

- `csound.CopyToLocal(src, dest)` - copies the file *src* in the ORIGIN directory to the local file *dest*, which can be accessed at ./local/*dest*. The "Complete" message is issued on completion.
- `csound.CopyUrlToLocal(url,dest)` - copies the url *url* to the local file *dest*, which can be accessed at ./local/*dest*. Currently only ORIGIN and CORS urls are allowed remotely, but local files can also be passed if encoded as urls with the webkitURL.createObjectURL() javascript method. The "Complete" message is issued on completion.
- `csound.RequestFileFromLocal(src)` - requests the data from the local file *src*. The "Complete" message is issued on completion.
- `csound.GetFileData()` - returns the most recently requested file data as an ArrayObject.

A series of examples demonstrating this API is provided in github. In particular, an introductory example is found on `http://vlazzarini.github.io/minimal.html`.

5.3.2 Limitations

The following limitations apply to the current release of Csound for PNaCl:

- no realtime audio input (not supported yet in Pepper/NaCl)
- no MIDI in the NaCl module. However, it might be possible to implement MIDI in

JavaScript (through WebMIDI), and using the csound.js functions, send control data to Csound, and respond to the various channel messages.

- no plugins, as pNaCl does not support dlopen() and friends. This means some Csound opcodes are not available as they reside in plugin libraries. It might be possible to add some of these opcodes statically to the Csound pNaCl library in the future.

6 Conclusions

In this paper we reviewed the current state of support for the development of web-based audio and music applications. As part of this, we explored two approaches in deploying Csound as an engine for general-purpose media software. The first consisted of a Javascript version created with the help of the Emscripten compiler, and the second a native C/C++ port for the Native Client platform, using the Portable Native Client toolchain. The first has the advantage of enjoying widespread support by a variety of browsers, but is not yet fully deployable. On the other hand, the second approach, while at the moment only running on Chrome and Chromium browsers, is a robust and ready-for-production version of Csound.

7 Acknowledgements

This research was partly funded by the Program of Research in Third Level Institutions (PRTLI 5) of the Higher Education Authority (HEA) of Ireland, through the Digital Arts and Humanities programme.

References

Paul Batchelor and Trev Wignall. 2013. BeaglePi: An Introductory Guide to Csound on the BeagleBone and the Raspberry Pi, as well other Linux-powered tinyware. *Csound Journal*, (18).

Richard J. Boulanger, editor. 2000. *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, February.

Michael Casey and Paris Smaragdis. 1996. Netsound. In *On the Edge*. ICMA and HKUST, August.

Alan Donovan, Robert Muth, Brad Chen, and David Sehr. 2010. PNaCl: Portable Native Client Executables. *Google White Paper*.

John ffitch, James Mitchell, and Julian Padget. 2007. Composition with sound web services and workflows. In Suvisoft Oy Ltd, editor, *Proceedings of the 2007 International Computer Music Conference*, volume I, pages 419–422. ICMA and Re:New, August. ISBN 0-9713192-5-1.

Tarmo Johannes and Kita Toshihiro. 2013. „Và, pensiero!“ - Fly, thought! Experiment for interactive internet based piece using Csound6 . `http://tarmo.uuu.ee/varia/failid/cs/pensiero-files/pensiero-presentation.pdf`. Accessed: February 2nd, 2014.

Victor Lazzarini, Steven Yi, and Joseph Timoney. 2012a. Digital audio effects on mobile platforms. In *Proceedings of DAFx 2012*.

Victor Lazzarini, Steven Yi, Joseph Timoney, Damian Keller, and Marco Pimenta. 2012b. The Mobile Csound Platform. In *Proceedings of ICMC 2012*.

Charles Roberts, Graham Wakefield, and Matthew Wright. 2013. The Web Browser As Synthesizer And Interface. *Proceedings of the International Conference on New Interfaces for Musical Expression*.

David Sehr, Robert Muth, Cliff Bife, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium*.

Lonce Wyse and Srikumar Subramanian. 2013. The Viability of the Web Browser as a Computer Music Platform. *Computer Music Journal*, 37(4):10–23.

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 IEEE Symposium on Security and Privacy*.

Steven Yi and Victor Lazzarini. 2012. Csound for Android. In *Linux Audio Conference*, volume 6.

Alon Zakai. 2011. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications*, pages 301–312. ACM.