Composing a Web of Audio Applications

Sarah Denoux
GRAME
11 cours de verdun (gensoul)
69002 Lyon, France
sdenoux@grame.fr

Stephane Letz
GRAME
letz@grame.fr

Yann Orlarey
GRAME
orlarey@grame.fr

Dominique Fober
GRAME
fober@grame.fr

ABSTRACT

The Web offers a great opportunity to share, deploy and use programs without installation difficulties. In this article we explore the idea of freely combining/composing real-time audio applications deployed on the Web using FAUST audio DSP language.

Keywords

FAUST, Composability, Web, DSP programming

1. INTRODUCTION

In his famous 1990 article "*Why Functional Programming Matters*", John Hughes[4] argued that "*modularity is the key to successful programming*" and that "*our ability to decompose a problem into parts depends directly on our ability to glue solutions together*". This paper is about gluing, composing, audio applications together, at the Web scale.

The concept of *composition* is familiar in music as in many human activities. Composition is usually defined as the action of putting together parts in order to form a whole. Composition implies some compatibility between parts. For example in mathematics two functions f and g can be composed $(f \circ g)$ if and only if the image of g is a subset of the domain of f. A highly composable system offers compatible components that share a common interface, and can be assembled in great variety of combinations. Lego bricks, Unix scripts combined with pipes, are good examples of composable designs. In the music domain, composable designs, like modular synthesizers and unit-generator based programming languages, favor expressiveness and creativity.

The idea of composability is also essential to the Web. The reflexion about how to build bridges between pages and compose data have brought hyperlinks, iframe, rss streams, etc. Recently, Web Components have made their apparition, trying to provide a higher level of composability for Web pages.

WAC '15 IRCAM Paris France

The conjunction of asm.js and the WebAudio API offers an unprecedented situation for realtime audio applications. Using a reverb, a compressor or a synthesizer could be as simple as opening the corresponding Web pages. In such a world, how can we go beyond data composition and do real-time behavior composition ? How can we turn the Web into a gigantic reservoir of modular audio components that can be indefinitely recombined ?

In this article we propose a first solution to the problem based on FAUST, asm.js and the WebAudio API. It will first describe the necessary principles of FAUST and its compiler (Section 2). Then the Web audio tools will be exposed in Section 3. Two approaches of composition will be explored in Section 4, to finally present the provided interface for online composition of FAUST programs (Section 5) and its performances (Section 6).

2. FAUST - A COMPOSABLE LANGUAGE

FAUST [Functional Audio Stream] [8] [7] is a functional, synchronous, domain-specific programming language specifically designed for real-time signal processing and synthesis. It aims to provide an adequate notation to describe *signal processors*: mathematical functions on signals. For example `+` is a function (of type $\mathbb{S}^2 \to \mathbb{S}^1$) that takes two input signals $x(t)$ and $y(t)$ and produces one output signal $z(t)$ such that $\forall t \geq 0, z(t) = x(t) + y(t)$.

Coding in FAUST is essentially composing *signal-processors* together to form new ones. For that purpose FAUST relies on an *algebra* of five composition operations. For example `+:abs` is the sequential composition of `+` and `abs`. The output of `+` is connected to the input of `abs`. It denotes a function that takes two input signals $x(t)$ and $y(t)$ and produces one output signal $z(t)$ such that $\forall t \geq 0, z(t) = |x(t) + y(t)|$.

A unique feature of FAUST is that programs are fully compiled and can be deployed in several environments (Max/MSP, VST, ...), programming languages (C++, C, Java, JS, LLVM), and platforms (OSX, Android, Linux, iOS, Web,...). The compiler provides advanced optimization techniques allowing the generated code to compete with hand written programs in terms of efficiency.

2.1 FAUST composing algebra

Since this article is about composing FAUST programs together, a focus on the composability aspects of the language is given. Everything in FAUST is a signal processor and programming in FAUST is essentially composing signal proces-

sors together using an algebra of five binary composition operations (see table 1).

`(A,B)`	parallel composition
`(A:B)`	sequential composition
`(A<:B)`	split composition
`(A:>B)`	merge composition
`(A~B)`	recursive composition

Table 1: The five binary composition operations

One can think of each of these composition operations as a particular way to patch two block diagrams. The algebra is complete and any topology of patch can be expressed. Nevertheless there are some constraints. Depending of the number of inputs and outputs of two components not all compositions are legal. For example the sequential composition `(A:B)` requires that the number of outputs of `A` is equal to the number of inputs of `B`. Therefore `(+:abs)` is a legal composition while `(abs:+)` will trig a type error because `abs` provides only one output while `+` requires two inputs.

Valid compositions of two components `A` and `B` according to their respective numbers of inputs and outputs are defined table 2.

`(A,B)`	$(\mathbb{S}^n \to \mathbb{S}^m) \to (\mathbb{S}^{n'} \to \mathbb{S}^{m'}) \to (\mathbb{S}^{n+n'} \to \mathbb{S}^{m+m'})$
`(A:B)`	$(\mathbb{S}^n \to \mathbb{S}^m) \to (\mathbb{S}^m \to \mathbb{S}^p) \to (\mathbb{S}^n \to \mathbb{S}^p)$
`(A<:B)`	$(\mathbb{S}^n \to \mathbb{S}^m) \to (\mathbb{S}^{k.m} \to \mathbb{S}^p) \to (\mathbb{S}^n \to \mathbb{S}^p)$
`(A:>B)`	$(\mathbb{S}^n \to \mathbb{S}^{k.m}) \to (\mathbb{S}^m \to \mathbb{S}^p) \to (\mathbb{S}^n \to \mathbb{S}^p)$
`(A~B)`	$(\mathbb{S}^{n+n'} \to \mathbb{S}^{m+m'}) \to (\mathbb{S}^{m'} \to \mathbb{S}^{n'}) \to (\mathbb{S}^n \to \mathbb{S}^{m+m'})$

Table 2: Valid compositions in Faust

Arbitrarily complex expressions can be composed with these five operations. But an additional level of modularity can be achieved with the `component("...")` primitive. It allows a full FAUST program, referred by a filename or an URL, to be used as a simple primitive inside any FAUST expression. For example:

```
component("http://faust.grame.fr/wahwah.dsp")
```

can be used to refer to a stereo Wah-Wah effect published on FAUST web site. All definitions, file imports, etc. of `wahwah.dsp` are kept in a separate lexical environment in order to avoid conflicts with the surrounding expressions.

By using `component()`, it is very simple to create a program that combines two existing programs published on the Web :

```
process=component("url1"):component("url2");
```

In the following sections we will see how to use this feature in conjunction with JavaScript tools to combine Web audio applications.

2.2 Faust Compiler

2.2.1 *Language deployment*

By being a specification language the FAUST code says nothing about the audio drivers or the GUI toolkit to be used. It is the role of the *architecture file* to describe how to relate the DSP code to the external world. This additional generic code is added to connect the DSP computation itself with audio inputs/outputs, and with control parameters, which could be buttons, sliders, num entries etc. in a standard user interface, or any kind of control using a remote protocol like OSC or HTTP.

This approach allows a single FAUST program to be easily deployed to a large variety of audio standards (Max-MSP externals, PD externals, VST plugins, CoreAudio applications, JACK applications, etc.).

2.2.2 *Static compilation chain*

The current version of the FAUST compiler (*faust1*) produces the resulting DSP code as a C++ class, to be inserted in the architecture file. The C++ file is finally compiled with a regular C++ compiler to obtain the final executable program or plug-in (Figure 1).

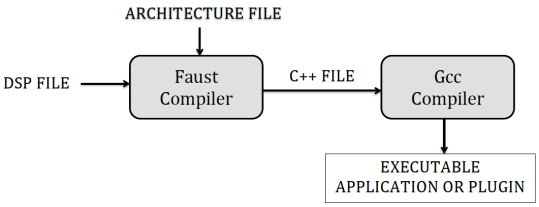

Figure 1: Steps of Faust compilation chain

2.2.3 *Multiple backends*

Faust2 development branch uses an intermediate FIR representation (FAUST Imperative Representation), which can be translated into several output languages.

The FIR language describes the computation performed on the samples in a generic manner. From this representation, various backends have been developed to produce C, C++, Java, JavaScript, asm.js, and LLVM IR (Figure 2).

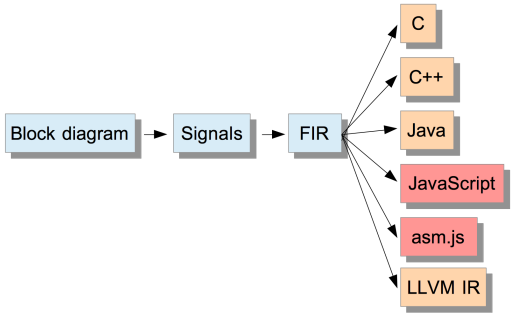

Figure 2: Faust2 compilation chain

2.2.4 *Faust Compiler as a library*

In the *faust2* development branch, the FAUST compiler has been packaged as an embeddable library called *libfaust*, published with an associated API. The *createDSPFactory(...)* function allows to create a DSP factory for a given DSP source code, then the *createDSPInstance(...)* function is used to create a DSP instance for a given factory.

This instance has to be wrapped by the audio and user interface architecture logic to connect the DSP computation with audio inputs/outputs, and the DSP parameters with the interface.

3. WEB AUDIO TOOLS

3.1 Interacting with the Web Audio API

The Web Audio API [9] specification describes a high-level JavaScript API for processing and synthesizing audio in Web applications. The conception model is based on an audio routing graph, where a number of AudioNode objects are connected together to define the overall audio rendering.

The actual processing is usually executed in the underlying implementation (typically optimized Assembly/C++ code), but direct JavaScript processing and synthesis is also supported.

3.2 Native nodes

The initial idea of the specification is to give the developers a list of highly optimized *native* nodes, implementing the commonly needed functions : playing buffers, filtering, panning, convolution etc. The nodes are connected to create an audio graph, to be processed by the underlying audio real-time rendering layer.

3.3 JavaScript ScriptProcessorNode

The *ScriptProcessorNode* interface allows the synthesis, processing, or analysis of audio using JavaScript. This AudioNode subtype module is linked to two buffers, one containing the input audio data, one containing the processed output audio data.

Each time the input buffer contains new data, an "AudioProcessingEvent" is sent to the scriptProcessorNode. The event handler terminates when the output buffer is filled with data.

This is the hook given to developers to add new low level DSP processing capabilities in the system.

3.3.1 Compiling to ASM JavaScript

Started in 2011 to facilitate the port of large C/C++ code base in JavaScript, Mozilla developers have started the *Emscripten* compiler project[11], based on LLVM technology, that generates JavaScript from C/C++ code.

Later on, they designed *asm.js*, a completely typed subset of JavaScript, statically compilable, garbage-collection free, that can be highly optimized by the compilation chain embedded in recent Web browsers. It is then possible to reach performances similar to pure native code [1].

Mainly designed to manipulate simple types like floating point or integer numbers, the *asm.js* language is particularly of interest for audio code.

3.3.2 Other works over the Web Audio API

Various JavaScript DSP libraries or musical languages, have been developed over the years ([2], [5], [1]) to extend, abstract and empower the capabilities of the official API. They offer users a richer set of audio DSP algorithms and sound models to be directly used in JavaScript code.

[1]asm.js code is said to be only 2 or 3 times slower than pure native code.

Using this approach means that developments have to be restarted from scratch, or reuse already written code (often in more real-time friendly languages like C/C++) to be rewritten and adapted in JavaScript.

An alternative interesting approach has recently been developed by the Csound team [6], using the C/C++ to JavaScript emscripten compiler, the complete C written Csound runtime and DSP language (so including a large number of sound opcodes and DSP algorithms) is now available in the context of the Web Audio API. Using an automatic C/C++ to JavaScript compilation chain opens interesting possibilities to ease the deployment of well-known and mature code base on the Web.

3.3.3 Developing a direct asm.js backend

In the FAUST project, a pure asm.js backend has been added in the *faust2* branch. It produces the asm.js module[2] as well as some additional helper JavaScript functions, to be wrapped by generic JavaScript to become a completely usable Web Audio node. Heap memory access and connection with helper functions defined in the asm.js module is managed by the wrapping code.

A new DSP instance is created using the following code, taking the Web audio context and a given buffer_size as parameters:

```
var dsp
    = faust.karplus(context, buffer_size);
```

The user interface can be retrieved as a JSON description:

```
var json = dsp.json();
```

The instance can be used with the following code:

```
dsp.start();
dsp.connect(context.destination);
dsp.update(path_to_control, val);
```

3.4 Programming Web Audio nodes with FAUST

Self contained ready to use Web audio nodes can be produced using the *faust2asmjs* script, using the static compilation chain previously described (c.f 2.2.2). The script basically calls the FAUST compiler targeting the asm.js backend, then wraps the produced code with generic JavaScript to be usable in the Web Audio API context.

3.4.1 Deploying Faust DSP examples in the Web

Since *faust2asmjs* produces a usable audio node, an HTML wrapper is the only missing thing to deploy a FAUST DSP as a self-contained Web page.

A script called *faust2webaudioasm* executes every step of this compilation to go from the DSP specification to the resulting HTML page : the FAUST compiler targeting the asm.js backend is called, then a more complex HTML code template is added to the DSP node, and the final HTML page is obtained. Thus it becomes simpler to publish DSP algorithms, helping wider the adoption of the FAUST DSL approach.

Adding the *-links* parameter to the script makes the HTML page also contain links to the original DSP textual file, as well as the block-diagram SVG representation.

[2]a scope consisting of a list of functions definitions and their exported prototypes, to be used in regular JavaScript code

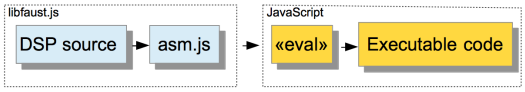

Figure 3: libfaust.js + asm.js dynamic compilation chain

3.4.2 *Producing the libfaust.js library*

Since the Emscripten compiler aims to help deploying any C++ code in the Web, it becomes possible to compile the FAUST compiler itself in pure JavaScript[3]. This has been done by going from the C++ *libfaust* library into the JavaScript *libfaust.js* library.

A unique *createAsmCDSPFactoryFromString(...)* entry point has been defined, allowing to create a DSP factory (as a result string) from DSP program given as a source string.

3.4.3 *Using the libfaust.js library*

Following the pattern of the C++ version of libfaust, two additional entry points have been built on top of the internal library entry point to create a DSP and execute it :

- *createDSPFactory(...)* : call the internal *createAsmCDSPFactoryFromString(...)* function to produce the asm.js module as a result string. Then calling JavaScript "eval" *compiles* this module in the browser, as well as several additional pure JavaScript methods that are part of the produced code (Figure 3).

```
var factory = faust.createDSPFactory(code);
```

- *createDSPInstance* : creates a fully working DSP instance as a Web audio Node.

```
var dsp = faust.createDSPInstance(factory,
                  context,
                  buffer_size);
```

The instance can then run with the following code:

```
dsp.start();
dsp.connect(context.destination);
```

To create a user interface, the JSON description can be retrieved :

```
var json = dsp.json();
```

Once controllers have been identified, they can be accessed through :

```
dsp.update(path_to_control, val);
```

4. TWO APPROACHES TO THE COMPOSITION OF FAUST PROGRAMS

With the arrival of Web tools for FAUST, the compilation and execution of FAUST programs within a Web page is now possible. The goal of the following sections is to explore their uses to compose audio applications deployed in the Web.

First, two approaches of composition will be examined.

[3]More specifically asm.js subset.

4.1 Graph composition

The notion of audio graph is to the numerical domain, what patching audio modules with cables is to the analog world. This concept of creating nodes and connecting one another is shared by the audio community (JACK, Max/MSP, SuperCollider, the Web Audio API, etc). The communication between nodes of such a graph is possible since the flowing data types are compatible.

FAUST programs can therefore be deployed in JACK, Max/MSP or SuperCollider and be composed "externally" in their respective graph structures. In the Web, FAUST programs will be Web audio nodes patched together to create a graph.

4.2 Equivalent Faust Composition

Another approach for composing FAUST programs is to calculate an equivalent FAUST program using the FAUST syntax (c.f. 2.1) to compose them. An interface for this type of composition could be designed in many ways. The first implementation imagined is available in FaustLive[3], where a table permits to create a component graphically : rows and columns respectively represent sequential and parallel composition. In each unit, a DSP can be dropped in the form of a string, a file or a Web URL. Once the visual patch is assembled, the FAUST equivalent is calculated, resulting in a standard FAUST program running in FaustLive.

The Web interface is the guideline of the next section. It combines both the Web audio graph composition with the equivalent FAUST approach, taking their respective advantages.

5. FAUST ON THE WEBAUDIO PLAYGROUND

5.1 Use Case

John, a FAUST user, prototyped a physical model of guitar and wants to share it. So, using faust2webaudioasm (cf. 3.4.1), he deploys his DSP as an HTML page. By adding it to his website, anybody can play with his guitar. Jane, another FAUST user plublished a disto on her website.

On the other side of the Web, Jeremy, comes on John guitar and Jane disto and wants to test them together. A simple test would be to compose them in sequence. So, he opens the component-creator page on FAUST website[4], drops the guitar and disto URLs that are compiled and executed in the page. Then he can adjust the connections as he wants and play with the parameters. At any point, an equivalent FAUST DSP can be calculated with the option to deploy it as a new native HTML page or as any other kind of application/plugin supported by the FAUST project[5].

5.2 Component creator interface

Following the described use case (cf. 5.1), the idea of the component creator is to have a tool to compose FAUST programs as a Web audio graph and also have the possibility to create the FAUST equivalent composition.

The interface is an extension of Chris Wilson page : "The Web audio playground", [10]. Added to this page is the possibility to create FAUST modules by dropping FAUST code

[4]a page containing libfaust.js

[5]This feature uses the remote compilation service, FaustWeb, available on http://faustservice.grame.fr

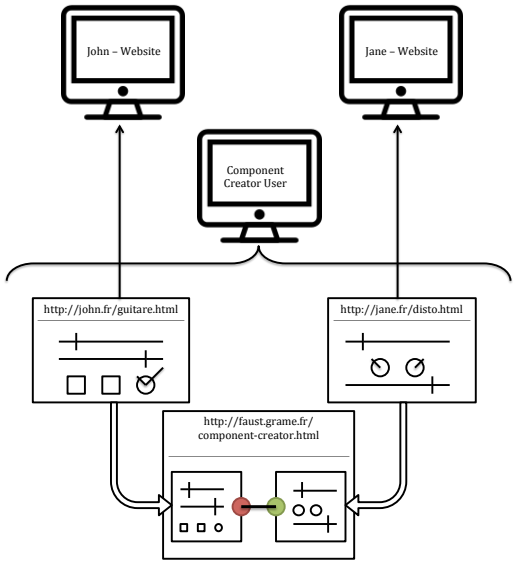

Figure 4: Use case description

into the page (Figure 5). The code can be dropped in the form of a string, a file or a Web URL. The DSP is compiled through libfaust.js to become a functional Web audio node that can be connected to others. At any time, the source code of a node can be edited and recompiled. Once a patch is assembled, tested and satisfying, the equivalent FAUST node can be calculated, following the algorithm described in section 5.3.

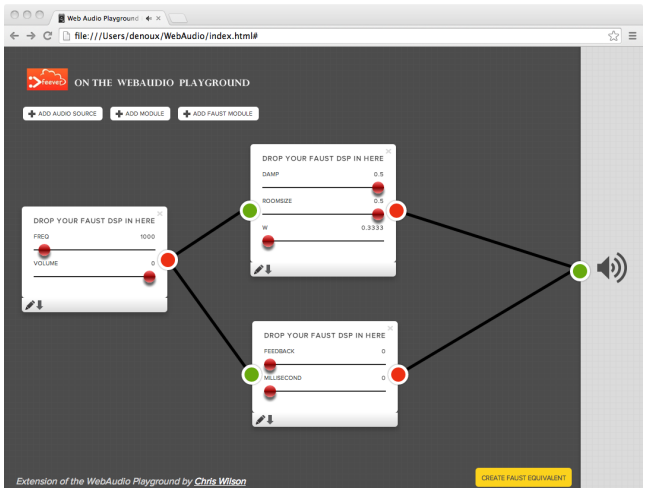

Figure 5: Screenshot of the component creator Web interface

5.3 Faust equivalent algorithm

To create the FAUST equivalent expression, three aspects were taken into account :

- create a simple expression
- stay coherent with the connections as they are made in the Web audio graph, so that the users would not have a totally different behavior once they compute the FAUST equivalent.
- follow FAUST semantics constraint that the number of channels have to match in order to compose two DSPs (c.f. 2.1).

Following these conditions, a tree is calculated from the graph of modules where the root node is the output of the graph and each node takes its entries as subnodes.

Once the tree is created, it can be computed to retrieve the FAUST equivalent expression. The function "compute" can be declared as such:

```
C(Graph) = "process = C(Output);"
```

The "compute" of a node is the "compute" of its sub-nodes, composed in parallel and merged into the node's own FAUST code.

```
C (N) = P(I0, I1, ..., In) :> Sc[N]
    with n : number of input nodes of N

P(I0, I1, ..., In) = C[I0], C[I1], ..., C[In]
```

In order to follow the constraints of channel matching, it was chosen to wrap the FAUST code of each node in the "stereoize" function to create stereo DSPs independently from their original characteristics.

```
Sc[N] = stereoize(faustcode(N));
```

```
  stereoize(p) = S(inputs(p),outputs(p));

with :
//degenerated processor
    S(n,0) = !,! : 0,0;

//processors with no inputs
    S(0,1) = p <: _,_;
    S(0,2) = p;
    S(0,n) = p,p :> _,_;

//processors with one input
    S(1,1) = p,p;
    S(1,n) = p,p :> _,_;

//processors with two inputs
    S(2,1) = p <: _,_;
    S(2,2) = p;

//other cases
    S(n,m) = _,_ <: p,p :> _,_;
```

This algorithm to create a single FAUST equivalent node from a graph of nodes is not optimized. As an example, the patch on Figure 6 is considered.

When creating the FAUST equivalent node, the algorithm will create the tree on Figure 7 to be computed by the FAUST compiler. In the example, the node A is repeated several times. Fortunately, the characteristic of the FAUST model is that this unoptimized code will be compiled and optimized within the FAUST compiler, so that the computed tree will be equivalent to the original one (Figure 7).

Thanks to this optimization, the algorithm can be kept simple and does not have to worry about the redundancies.

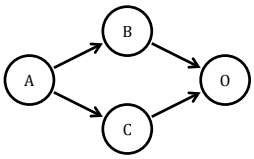

Figure 6: Patch of Faust modules

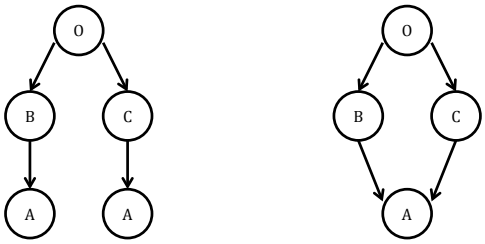

Figure 7: Computed patch and its equivalent

6. COMPARISON OF PERFORMANCES

The gathering of the Web audio graph approach with the FAUST equivalent combines their respective advantages.

On the one hand, the graph approach brings a very short and ergonomic test/edit/connect cycle but it can be difficult to handle a large number of modules (ergonomically and computationally).

On the other hand, the FAUST equivalent approach makes it more complicated to edit the code/the connections betweens modules but it brings a more compact vision of the patch along with a computational optimization. Moreover, the resulting program can be downloaded as any type of application/plugin supported by the FAUST project (as can be each module independently).

A benchmark has been carried out to test the performances of a patch of FAUST modules in comparison with the FAUST equivalent module. It is based on an algorithm describing the rebound of a ball, built with multiple delay lines composed together. The Figure 8 displays the results of the benchmark for different number of rebounds[6].

Focusing on the FAUST approaches (patch and equivalent), the difference of performance can be appreciated. To explain this gap, two aspects can be inferred :

- First, a graph with many nodes can be heavy since the audio rendering layer has to evaluate a complex graph, basically calling each node "process" function and transferring audio data between nodes etc. Reducing the number of nodes reduces this cost.
- Moreover, the FAUST compiler has optimization steps that allow to reduce the CPU usage of the single FAUST equivalent node. In this particular case, FAUST applies several rules to simplify and normalize output signal computation to share the same delay line between a unique writer and several readers.

[6]Executed on OSX10.6.8 - Chrome 39.0.2171.95 - Values retrieved from Chrome developer tools

Number of Delay Lines	Patch of Native Nodes	Patch of Faust Nodes	Single Faust Equivalent Node
5	9,5%	7%	4%
10	11,00%	11%	4,50%
15	13,0%	14,2%	5%
20	13,5%	16,8%	5,0%
25	14,0%	17%	5,5%
30	16%	19,00%	6,00%
35	16,30%	20%	6,0%
40	17,00%	22,40%	6,3%
45	17,50%	24%	6,4%
50	18%	25%	6,70%

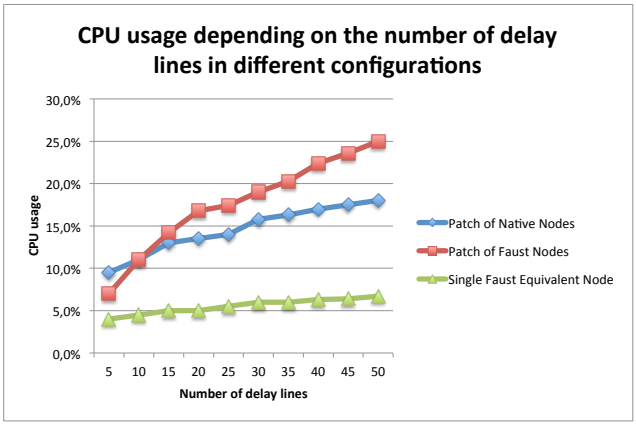

Figure 8: Comparison of performances for a rebond algorithm in different configurations

Differentiating what can be granted to the FAUST compiler and what comes from the Javascript cost is not trivial.

Another test has been carried out with multiple biquad filters in parallel (Figure 9). The same trend can be observed.

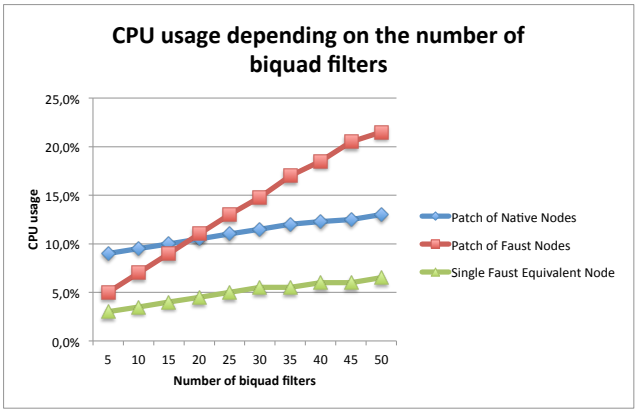

Figure 9: Comparison of performances for multiple biquads

As informative comparison, a patch of native nodes has been benchmarked, supposing that the implementation of the delay and biquad nodes are more or less similar to the FAUST algorithms. It is possible to observe that the patches of FAUST modules are not so costlier than the native patches (from what we expected). Having said that, the tests that were made are not conclusive in knowing what part of the slopes could be imputed to the processing and which one to the graph evaluation. This benchmark also has the restriction of the browser it was tested on.

Nevertheless, the results are encouraging for the FAUST model. And the creation of a FAUST equivalent node from a graph of modules highly gains in performance.

7. CONCLUSION

The FAUST audio DSP language can be used to easily develop new audio nodes in the Web Audio model, and use them in an audio graph. Complete HTML pages with a working user interface can also be generated.

Moreover, having the dynamic compilation chain, through *libfaust.js*, directly available in the browser made it accessible to develop a composition tool for FAUST programs. More complete benchmarks with other examples will be performed, along with a comparison of performances on different browsers.

Combining the composability of the FAUST algebra, with the Web audio tools makes composing audio applications as simple as dropping URLs in a Web page.

Acknowledgments

This work has been implemented under the FEEVER project [ANR-13-BS02-0008] supported by the "Agence Nationale pour la Recherche".

8. REFERENCES

[1] H. Choi and J. Berger. Waax: Web audio api extension. New Interfaces for Musical Expression Conference, 2013.

[2] C. Clark and A. Tindale. Flocking: a framework for declarative music-making on the web. 2014.

[3] S. Denoux, S. Letz, Y. Orlarey, and D. Fober. Faustlive just-in-time faust compiler... and much more. Linux Audio Conference, 2014.

[4] J. Hughes. Why functional programming matters. 1990.

[5] Kalliokoski J. audiolib.js, a powerful toolkit for audio written in js. https://github.com/jussi-kalliokoski/audiolib.js, 2014.

[6] V. Lazzarini, E. Costello, S. Yi, and J. Fitch. Csound on the web. Linux Audio Conference, 2014.

[7] Y. Orlarey, S. Letz, and D. Fober. Syntactical and semantical aspects of faust. 2004.

[8] Y. Orlarey, S. Letz, and D. Fober. Faust: an efficient functional approach to dsp programming. 2009.

[9] Web audio api. http://webaudio.github.io/web-audio-api.

[10] The web audio playground. http://webaudioplayground.appspot.com.

[11] A. Zakai. Emscripten: an llvm to javascript compiler. ACM international conference companion on Object oriented programming systems languages and applications, 2011.