Proceedings of the 2nd International Faust Conference (IFC-20), Maison des Sciences de l'Homme Paris Nord, France, December 1-2, 2020

HYBRIDIZING FAUST AND SOUL

Stéphane Letz,[a], Yann Orlarey,[a], Romain Michon,[a,b] and Dominique Fober[a]

[a]GRAME – Centre National de Création Musicale, Lyon, France
[b]Center for Computer Research in Music and Acoustics, Stanford University, USA
{letz,orlarey,michon,fober}@grame.fr

ABSTRACT

SOUL is a new audio Domain Specific Language and a runtime platform developed by ROLI, which aims at modernizing and optimizing the way high-performance and low-latency audio code is written and executed. FAUST is a functional Domain Specific Language specifically designed for real-time audio signal processing. Both approaches share common ideas: sample-level DSP computation, fixed memory and CPU footprints, dynamic JIT compilation, CPU efficiency, multi-targets deployment (native and embedded platforms, up to the Web). This paper presents a work in progress done around the idea of hybridizing FAUST with SOUL.

1. INTRODUCTION

There is now a long history of using FAUST [1] inside existing audio languages or environments. The FAUST compiler typically generates DSP objects as a C++ class to be wrapped by so-called "architecture files" and connected to the external world [2]. The first versions were developed to generate static executables, to be used as standalone applications or loaded as plugins in existing audio environments. This has been done with (to name a few):

- for Max/MSP with `faust2max6`, [1]
- for SuperCollider with `faust2supercollider`
- for Csound with `faust2csound`

The compiled DSP code then appears as an audio node processing audio streams, inserted in the host application audio pipeline, and must be controlled with a graphical user interface, MIDI, OSC, or any kind of control interface.

A second category of integrations appeared with the development of `libfaust`, the library version of the FAUST compiler, embedding its LLVM backend and able to dynamically compile and execute programs [3]. The technology has been used in several environments (non-exhaustive list):

- in `faustgen`, an external for the Max/MSP and PureData environments
- in `faustlive`, a standalone QT application [4]
- in `pure-faust` allowing us to run Faust-generated signal processing modules in Pure[2]
- in Antescofo, a language developed at IRCAM whose main goal is to synchronize electronic systems and musicians in the context of interactive music composition [5]
- in Chuck with the FAUCK project [6] which combines the FAUST and Chuck languages

Depending on the context, the FAUST code can be directly edited in the host environment using an integrated editor, or alternatively with an external editor. Having the FAUST compiler directly integrated in the host environment means that all the libraries [7, 8] and examples can be directly used, and very efficient code compiled for the native CPU [3] can be generated on-the-fly. Ubiquitous music ecosystems can then be considered and developed using this approach.

This paper presents a work in progress carried out around the idea of hybridizing FAUST and SOUL. Several tools recently developed will be demonstrated to compile FAUST DSP programs to SOUL (using the new SOUL backend added in FAUST compiler) and to combine code produced by both languages. A proposal for hybridization at source code level will be exposed.

2. THE SOUL LANGUAGE

SOUL is an audio Domain Specific Language and a runtime platform first announced at the Audio Developer Conference 2018 by ROLI. [4] It aims to provide developers a secure by design language to be safely executed on Domain Specific Architectures (DSAs) like remote devices, drivers, bare-metal or realtime kernels. [5] It follows a processors and graph model where stream processing is described in processors connected in larger graphs.

As a language, it follows a classical procedural imperative language with various types, data, and control structures. Some syntactical choices simplify writing common stream access operations and processor connections. It is also a dynamic JIT compiled language (using LLVM), producing memory bounded and hard real-time-ready programs. Since the language is compiled, various compile-time optimizations can be achieved, such as producing code for a fixed sample rate or fixed parameters for filters, etc. As a platform, it aims at becoming a runtime distributed with specialized hardware, and possibly become a standard.

2.1. Processor and Graph

Processors are the basic blocks of DSP processing. They declare a set of audio and control input/output streams and implement processing in a mandatory special `run()` function, usually containing an infinite loop, inside which inputs are read and outputs are written, doing whatever operations are needed on the processed samples.

A special `advance()` method has to be called to move all the streams forward by a fixed time interval, depending of the pro-

[1] `https://github.com/grame-cncm/faust/tree/master-dev/architecture/max-msp`

[2] `https://github.com/agraef/pure-lang/wiki/Faust`

[3] On modern SIMD supporting CPUs, compiling for the machine "native" CPU instead of a "generic" one, can give a significant speedup.

[4] `https://www.youtube.com/watch?v=-GhleKNaPdk`

[5] `https://github.com/soul-lang/SOUL/blob/master/docs/SOUL_Overview.md`

cessor rate. This coroutine model allows to return control to the graph computation, in order to process the other processors in the graph topology.

Control streams can be synchronous, with values possibly received at each time-stamp, asynchronous, with values received at any time, or event-based with an associated sample-accurate callback.

Processors are then connected together to form graphs of arbitrary size. Graphs will themselves declare input/output control and audio streams. Processors or graphs within a graph can run at a faster or slower rate than the graph containing them. The oversampling factor is specified as a compile-time integer constant with different interpolation and filtering strategies.

2.2. Polyphony Support

Compared to FAUST, since it is a more generic programming language, SOUL allows the description of DSP algorithms, as well as the mechanisms to use them in larger programs. Polyphonic instruments can be directly coded in the language as arrays of DSP voices, allocated with a voice allocator triggered by incoming MIDI events.

2.3. Running and Embedding SOUL Code

SOUL programs are described in a high-level audio-plugin-like format as SOUL patches, appropriate for enabling SOUL support in DAWs and other plugin. At the time of writing, the SOUL compiler can be accessed and tested with:

- The `soul` command, which contains both the compiler and the JIT chain embedded in a simple MIDI, audio, and GUI runtime built using the JUCE framework. The command can also generate the `heart` Internal Representation (IR) format or C++ code. [6]
- A loadable library named `SOUL_PatchLoader` usable with a C++ API to dynamically compile and run SOUL patches to be rendered by an additional audio, MIDI, and control runtime.
- The SOUL Playground, [7] a Web platform compiling SOUL code to WebAssembly, and where code and examples can be tested with MIDI and audio samples support.

For developers, a preliminary SDK with a set of C++ API header and helper files can be used to access the compiler as a library. SOUL patches can be loaded and compiled.

A `PlayerInstance` object is then created and has to be fed with audio buffers and MIDI events. External resources like audio samples can be declared in the source code with a special syntax, and will be automatically loaded by the library.

3. HYBRIDIZING FAUST AND SOUL

Several tools have been developed to experiment hybridizing the two languages. The initial idea was to act at the level of the source code, and see how the FAUST compilation chain could be adapted to generate SOUL code.

[6] At the time of writing, the exact API exposed by the C++ code is still not stable.

[7] `https://soul.dev/playground/`

3.1. The SOUL Backend

A SOUL backend has been added to the FAUST compiler. It translates the intermediate Faust Imperative Representation (FIR) into SOUL source code. The equivalent of the C++ `dsp` class standard API (with a set of access and initialisation methods) is generated as a SOUL `processor`. The sample computation code in emitted in the special mandatory `run()` function, in an endless loop, which calls the special `advance()` function at its end to give control back to the DSP graph computation.

Controllers are translated in the SOUL concept of input event streams, where all received values trigger sample-accurate callbacks. The actual controller values are declared as fields in the SOUL processor, to be written by the event callback with the received values. When external events are received, a global `fUpdated` flag is set to true (in each callback) to indicate that the control state has to be recomputed at the next audio cycle. A `control()` function, called at the beginning of the `run()` function will check this flag and do the computation if needed.

Sub-classes (typically needed when tables are used) are compiled as SOUL `struct` definitions with a set of associated functions (initialisation, filling the structure, etc). Waveform primitives are compiled as static arrays. FAUST code can be generated in `float` or `double` mode, which naturally translates to the SOUL `float32` and `float64` types. Here is an example of FAUST source code:

```
random  = +(12345)~*(1103515245);
noise   = random/2147483647.0;

vol1 =  hslider("vol1", 0.5, 0, 1, 0.01);
vol2 =  hslider("vol2", 0.5, 0, 1, 0.01);

process = noise * vol1, noise * vol2;
```

and (part of) its translation in SOUL:

```
processor noise {
    input event float32 eventfHslider0
    [[ name: "vol1",
      group: "/v:Noise/vol1",
        min: 0.0f,
        max: 1.0f,
       init: 0.5f,
       step: 0.00999999978f ]];

    input event float32 eventfHslider1
    [[ name: "vol2",
      group: "/v:Noise/vol2",
        min: 0.0f,
        max: 1.0f,
       init: 0.5f,
       step: 0.00999999978f ]];

    output stream float32 output0;
    output stream float32 output1;

    float32 fHslider0;
    int32[2] iRec0;
    float32 fHslider1;
    int32 fSampleRate;
    bool fUpdated;
    float32[2] fControl;
```

```
    event eventfHslider0 (float32 val) {
        fHslider0 = val; fUpdated = true;
    }

    event eventfHslider1 (float32 val) {
        fHslider1 = val; fUpdated = true;
    }

    void control() {
        fControl[0] = (4.65661287e-10f *
            float32 (fHslider0));
        fControl[1] = (4.65661287e-10f *
            float32 (fHslider1));
    }

    void run() {
        // DSP loop running forever...
        loop {
            // Updates control only if
                needed
            if (fUpdated) {
                fUpdated = false;
                control();
            }
            // Computes one sample
            iRec0[0] = ((1103515245 * iRec0
                [1]) + 12345);
            float32 fTemp0 = float32 (iRec0
                [0]);
            output0 << float32 ((fControl
                [0]*fTemp0));
            output1 << float32 ((fControl
                [1]*fTemp0));
            iRec0[1] = iRec0[0];

            // Moves all streams forward
            // by one 'tick'
            advance();
        }
    }
}
```

Here, two controllers are declared as `eventfHslider0` and `eventfHslider1`, and are modified in the associated callbacks. Two audio output streams `output0` and `output1` are declared. The `control()` function is called in `run()` if necessary (when `fUpdated` is true). Finally `run()` computes the output samples and calls `advance()` when finished.

3.2. Architectures

Using the `SOUL_PatchLoader` library, a `PlayerInstance` object can be created from a SOUL patch. A `soulpatch-dsp` adapter class, subclass of the `dsp` base class, has been written to use any SOUL patch with the existing FAUST architecture files. Several tools have also been developed to ease the interaction between FAUST and SOUL programs.

3.3. The faust2soul Tool

The `faust2soul` tool compiles a FAUST DSP program in a folder containing the generated SOUL source code and SOUL patch. With the adapted wrapper architecture file (`minimal.soul` for the monophonic case and `poly-dsp.soul` for the polyphonic one, containing some generically written SOUL polyphonic handling code), the result can be a monophonic DSP, or a MIDI controllable polyphonic one (when the DSP describes an instrument, following the `freq`, `gain`, `gate` parameter naming convention). The resulting SOUL patch can be played using the `soul` runtime, by using the code as a sub part of a more complex program, or by compiling it in the SOUL playground.

3.4. The soul-faust-player Tool

Since signal processors designed in the SOUL language are similar to the notion of DSP produced by the FAUST compiler, it is quite natural to consider the possible hybridization between the two languages, by having them cohabit in the same program. The two compilation chains then have to be used together to produce a single resulting executable program.

The idea has been tested by extending the SOUL language syntax, with blocks named `faust {....}` containing arbitrary FAUST programs. Those blocks are then extracted from the original file, compiled into SOUL processor blocks using the SOUL backend, and the resulting SOUL code is then inserted back into the original file, then compiled by the SOUL compilation chain. This feature can be tested in a tool name `soul-faust-player`.

An example of hybrid file is shown below: an `addSynth` synthesizer is defined first, then a SOUL block, then a second FAUST effect `stereoEcho`. The `sequence` SOUL graph does the actual connections between the three processors.

Since there is currently no way to ask processors and graphs about their audio and control inputs/outputs in a generic way, the names of the controller and audio inputs/outputs of the FAUST generated blocks have to be known in advance to write the connection block code in the `graph sequence` section. This is obviously a rather serious limitation that makes the whole approach a bit experimental for now.

```
faust addSynth {
    import("stdfaust.lib");
    vol = hslider("volume
        [unit:dB]",-20,-96,0,0.1) :
        ba.db2linear : si.smoo;
    freq = hslider("freq
        [unit:Hz]",500,100,2000,1);
    process = vgroup("addSynth",(
        os.osc(freq) +
        0.5*os.osc(2.*freq) +
        0.25*os.osc(3.*freq))*vol);
}

processor ClassicRingtone {
    output stream float out;
    void run() {
        int[] pitch =
            (76,74,66 68,73,71,62,64);
        int[] durations =
            (1,1,2,2,1,1,2,2);
```

```
        let spQuarterNote =
            int(processor.frequency/7);
        float swPhase;

        loop {
            for (int note = 0; note < pitch
                .size; ++note) {
                let nF = soul::
                    noteNumberToFrequency(
                    pitch.at(noteIndex));
                let noteLength =
                    spQuarterNote*durations.
                    at(noteIndex);
                let phaseInc = float(nF*
                    twoPi*processor.period);

                loop(noteLength) {
                    out << 0.9f * sin(
                        swPhase);
                    swPhase = addModulo2Pi(
                        swPhase, phaseInc);
                    advance();
                }
            }
        }
    }
}

faust stereoEcho {
    import("stdfaust.lib");
    gain = hslider("gain",0.5,0,1,0.01);
    feedback =
        hslider("feedback",0.8,0,1,0.01);
    echo(del_sec, fb, g) = +~de.delay
        (50000,del_samples)*fb*g
    with {
        del_samples = del_sec*ma.SR;
    };
    process =
        echo(1.6,0.6,0.7),
        echo(0.7,feedback,gain);
}

graph sequence [[main]] {
    // Events to control Faust Synth
    input event float32 eventfHslider1
    [[ name: "freq",
     path: "/addSynth/freq",
     min: 100.0f,
     max: 2000.0f,
     init: 500.0f,
     step: 0.01f,
     unit: "Hz" ]];
    input event float32 eventfHslider0
    [[ name: "volume",
     path: "/addSynth/volume",
     min: -96.0f,
     max: 0.0f,
     init: -20.0f,
     step: 0.100000001f,
     unit: "dB" ]];

    // Events to control Faust Echo
    input event float32 eventfHslider2
    [[ name: "gain",
     path: "/stereoEcho/gain",
     min: 0.0f,
     max: 1.0f,
     init: 0.5,
     step: 0.01f ]];
    input event float32 eventfHslider3
    [[ name: "feedback",
     path: "/stereoEcho/feedback",
     min: 0.0f,
     max: 1.0f,
     init: 0.8,
     step: 0.01f ]];

    output stream float audioOut0;
    output stream float audioOut1;

    connection {
        // Connect to Faust addSynth
        eventfHslider0 -> addSynth.
            eventfHslider0;
        eventfHslider1 -> addSynth.
            eventfHslider1;

        // Connect to Faust stereoEcho
        eventfHslider3 -> stereoEcho.
            eventfHslider0;
        eventfHslider2 -> stereoEcho.
            eventfHslider1;

        addSynth.output0 -> stereoEcho.
            input0;
        ClassicRingtone.out -> stereoEcho.
            input1;

        stereoEcho.output0 -> audioOut0;
        stereoEcho.output1 -> audioOut1;
    }
}
```

3.5. The soul-faust-tester Tool

The `soul-faust-tester` tool allows for the testing of DSP CPU usage of FAUST and SOUL programs, dynamically compiling and running them, using the `soulpatch-dsp` adapter class in the case of SOUL. It measures the DSP CPU usage as MBytes/sec and as a percentage of the audio bandwidth at 44.1 kHz.

4. DISCUSSION

4.1. SOUL As a Language

Giving the programmer the ability to embed piece of FAUST code directly in a SOUL program is quite natural since each FAUST program finally appears as a `processor {...}` block in the generated code, that can be used like a native SOUL processor

block. Each language has advantages and drawbacks. Depending on the needs, some programmers will prefer the imperative SOUL approach, others prefer the more declarative FAUST mathematical specification. Moreover, with the presented tools, SOUL developers can immediately take profit of all libraries and examples of the FAUST ecosystem.

The SOUL language and platform is still at its early stages. Since SOUL is developed and promoted by a commercial company, the future status of the project is still not clear. Only part of the source code is currently available as open-source, basically everything from the frontend up to the generation of the `heart` IR format.[8] It has yet to be seen if the complete compilation chain will be fully opened, possibly allowing deeper integration with external tools or languages as demonstrated with the hybrid FAUST/SOUL prototype.

4.2. SOUL As a Platform

The promise of the SOUL platform is to bring a runtime that can be deployed on a whole set of architectures, from a classical computer to embedded architectures, up to the Web. [9] The designers of SOUL hope for instance to convince Apple to allow the deployment of the SOUL JIT compiler on the iOS platform. This would allow any audio language generating SOUL code to access this platform. A similar idea could be considered in the Web browser domain. If successful, this approach will allow Faust DSP code to be easily deployed on new platforms.

5. BENCHMARK

Since FAUST and SOUL both generate sample-level DSP code, we can compare FAUST code generated using the LLVM backend and JIT compiled to native, versus FAUST code generated to SOUL and executed using the `SOUL_PatchLoader` library. Here is the result for a set of DSPs, with an additional C++ version (where the code has been compiled for a generic CPU), the LLVM version for the native CPU, and the same for SOUL, since by default SOUL always generates code for the native CPU (see Figure 1). As expected, CPU native compiled code usually runs faster than the generic one, and we can see that some optimisations are probably still lacking in the SOUL LLVM IR generated code.

6. CONCLUSION

The presented work shows how the purely functional FAUST language can be hybridized with the new imperative SOUL language. Several tools have been developed to ease interactions between the two languages.

It turns out that the current limitations of SOUL regarding generic access to processor and graphs internal characteristics limits what can be currently done at the source code level.

One other possibility would be to access the SOUL compilation chain at a more internal level (like directly generating the SOUL `heart` IR format for instance) so that to better mix the two languages. Since the SOUL implementation is not fully open-source, we cannot evaluate if this kind of smoother integration is technically possible yet.

[8] `https://github.com/soul-lang/SOUL`

[9] `https://github.com/soul-lang/SOUL/blob/master/docs/SOUL_Overview.md`

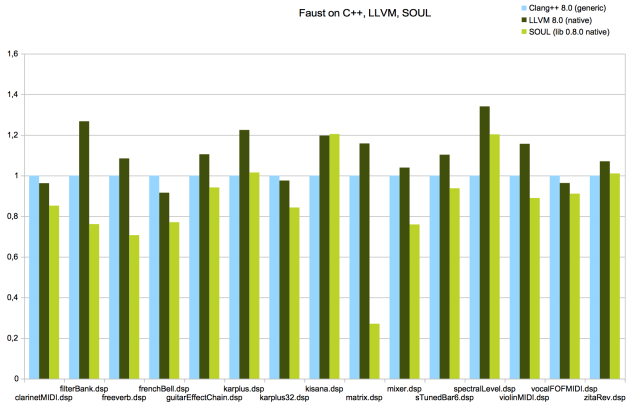

Figure 1: *Faust DSPs compiled for C++, LLVM and SOUL, all using LLVM 8.0 version. Values of 1 are given by the C++ reference for each test, other values are relative to this reference.*

7. ACKNOWLEDGMENTS

We want to thank SOUL designers and developers Cesare Ferrari and Julian Storer for fruitful discussions on the SOUL language and platform.

8. REFERENCES

[1] Yann Orlarey, Stéphane Letz, and Dominique Fober, *New Computational Paradigms for Computer Music*, chapter "Faust: an Efficient Functional Approach to DSP Programming", Delatour, Paris, France, 2009.

[2] Stéphane Letz, Yann Orlarey, Dominique Fober, and Romain Michon, "Polyphony, sample-accurate control and MIDI support for FAUST DSP using combinable architecture files," in *Proceedings of the Linux Audio Conference (LAC-17)*, Saint-Étienne, France, 2017.

[3] Stéphane Letz, Dominique Fober, and Yann Orlarey, "Comment Embarquer Le Compilateur Faust Dans Vos Applications ?," in *Proceedings of the Journées d'Informatique Musicale*, Paris, France, 2013.

[4] Sarah Denoux, Stéphane Letz, Yann Orlarey, and Dominique Fober, "FAUSTLIVE, Just-In-Time Faust Compiler... and much more," in *Proceedings of the Linux Audio Conference*, Karlsruhe, Germany, 2014.

[5] Nicolas Schmidt Gubbins, Arshia Cont, and Jean-Louis Giavitto, "First steps toward embedding real-time audio computing in Antescofo," *Journal de Investigacion de Pregado (Investigacion, Interdisciplina, Innovacion)*, vol. 6, 2016.

[6] Ge Wang and Romain Michon, "Fauck!! hybridizing the faust and chuck audio programming languges," in *Proceedings of Sound and Music Conference (SMC-16)*, Hamburg, Germany, Septmber 2016.

[7] Julius O. Smith, "Signal processing libraries for Faust," in *Proceedings of Linux Audio Conference (LAC-12)*, Stanford, USA, May 2012.

[8] Romain Michon, Julius Smith, and Yann Orlarey, "New signal processing libraries for Faust," in *Proceedings of the Linux Audio Conference (LAC-17)*, Saint-Etienne, France, 2017.