Automatically generating eurorack hardware running faust programs using eurorack-blocks

Raphaël Dingé, Stéphane Letz

AUTOMATICALLY GENERATING EURORACK HARDWARE RUNNING FAUST PROGRAMS USING EURORACK-BLOCKS

Raphaël Dingé

Ohm Force, 8 le Nézert 22340 Trébrivan, France
raphael.dinge@ohmforce.com

Stéphane Letz

Univ Lyon, GRAME-CNCM, INSA Lyon, Inria, CITI, EA3720,
69621 Villeurbanne, France
letz@grame.fr

ABSTRACT

This paper describes the integration of FAUST with Eurorack-blocks, a framework capable of generating programmatically Eurorack digital modules' hardware and firmware files for manufacturing, and providing a virtual environment for early-stage design, development, testing and debugging iterations on a desktop computer.

It presents a method to statically bind the inherently nested structure of a FAUST DSP program with the flat namespace and different types of the ERBUI and ERBB languages, which are Domain Specific Languages to describe the Eurorack module UI, module DSP file and associated audio data, respectively.

An implementation is demonstrated, taking into consideration the specific memory model of the hardware embedded platform, as well as the meta-programming technique used to minimize computations done at run time by relocating them at build time.

1. INTRODUCTION

1.1. Background

Eurorack is a modular synthesizer format originally specified in 1996 by Doepfer Musikelektronik[1]. It became since 2021 the dominant hardware modular synthesizer format, with over 12,000 modules available from more than 500 different manufacturers and individuals[2][3].

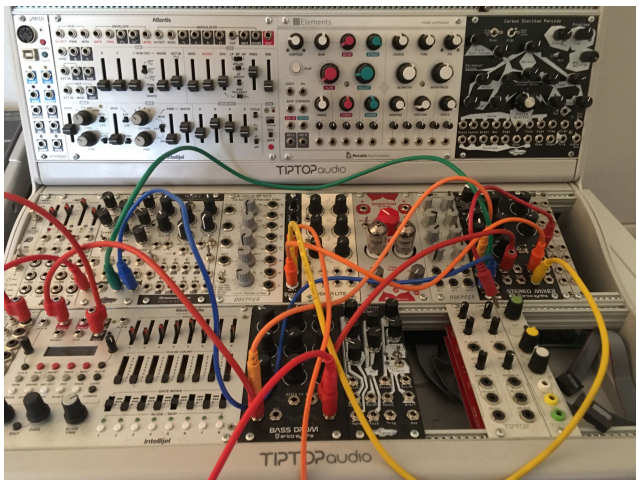

Figure 1: *A Eurorack system.*

A Eurorack system (see Figure **??**) is composed of Eurorack modules that implement electrical and mechanical specifications from Doepfer, for them to properly interact with each other.

A module has a user interface that typically includes buttons, switches, pots or LEDs the usage of which is totally up to the module developer. Conversely, modules also include 3.5mm mono jack connectors used as inputs or outputs, which allow transporting a signal from one module to another, and which shall follow Doepfer electrical specifications [4].

There are 3 main types of signals: Audio, Control Voltage (CV), or Gate signals:

- An audio signal represents a mono analog audio signal, typically ranging from -5V to 5V,
- A Control Voltage represents a continuous parameter typically used for modulation, They can represent LFOs, ADSR envelopes, or pitch information, over various electrically specified ranges of voltages,
- A Gate represents a boolean parameter typically used for triggers or tempo clocks.

Eurorack users usually credit the platform for its haptic feedback and immediacy that prevents from interrupting their creative flow. Conversely though, the rapid complexity of a patch can quickly lead to errors, which can produce unexpected pleasing sounds, which users call "happy mistakes".

While the Eurorack platform was only using analog components when first introduced, more and more manufacturers are producing today digital modules which use Analog to Digital Converters (ADC) and Digital to Analog Converters (DAC) to stay compatible with the original electrical specifications, while allowing the use of embedded digital processors to produce the desired effect.

We can roughly estimate that the proportion of digital modules is approximately less than 10% of the total number of Eurorack modules produced[5]. This can be explained by the quite complex setup needed to just start any kind of work, while most individuals can start to develop analog modules with a few low-cost electronic components. Another limiting factor is the power consumption of digital modules compared to their analog counterparts, which restricts the models of processors usable in practice, as the maximum total consumption for a Eurorack system is usually around 40W.

[1] https://doepfer.de/a100e.htm
[2] https://www.modulargrid.net/e/vendors
[3] https://www.modulargrid.net/e/modules/browser

[4] In practice manufacturers bend some rules, but in a way that came to a general consensus. For example, LFOs are using usually audio signal voltage ranges instead of the specified half the range, and Gate signals can go up to 10V instead of the specified 5V

[5] https://www.modulargrid.net/e/modules/browser?SearchFunction=12

Based on this observation, the company Electro-smith [6] started to develop the Daisy Seed board which combines on a single board, a low-consumption ARM embedded processor, SDRAM memory as well as a stereo audio codec, and they developed a software library which abstracts low-level embedded software development for this hardware board.

The project was brought to a crowd-funding platform and funding was quite a success. Electro-smith produced the Daisy Patch Eurorack module (see Figure **??**), which brings a predefined user interface and uses the Daisy Seed board while adapting signals to the Eurorack electrical specifications.

Figure 2: *The Daisy Patch combines a predefined set of 4 knobs and linked CV inputs, 4 audio inputs and outputs, 2 CV outputs, 2 Gate inputs, 1 Gate output, 1 MIDI input and output, a rotary encoder, OLED screen and SD card reader. The Daisy Seed board can be seen on the lower right of the picture.*

At the beginning of 2022, Electro-smith released a new Daisy Patch Submodule board [7] (see Figure **??**) which was designed exclusively with Eurorack in mind, by implementing Eurorack electrical specifications directly onto the board, to further ease the development of Eurorack modules for manufacturers and individuals.

Using the Daisy Patch Submodule board still requires knowledge in hardware engineering, and compliance with the mechanical specifications dictated by Doepfer.

Our open-source project, Eurorack-blocks [8], focuses on this part, since the number of engineers who have both strong hardware and software expertise is quite low.

The Eurorack-blocks allows a software developer to develop their own custom Eurorack module, with a custom user-defined interface, by allowing them to design, develop, test, debug and iterate on their desktop computer with their development platform of choice, using a simulator and a virtual Eurorack environment, before they even start to produce the hardware.

This allows for a much more Agile approach to Eurorack development, which makes Eurorack-blocks at least a good option for prototyping, as producing a prototype requires fewer team interactions in a company, as a software developer can produce their prototype in isolation, without requiring the help of a hardware engineer colleague.

[6] `https://www.electro-smith.com`
[7] `https://www.electro-smith.com/daisy/patch-sm`
[8] `https://github.com/ohmtech-rdi/eurorack-blocks`

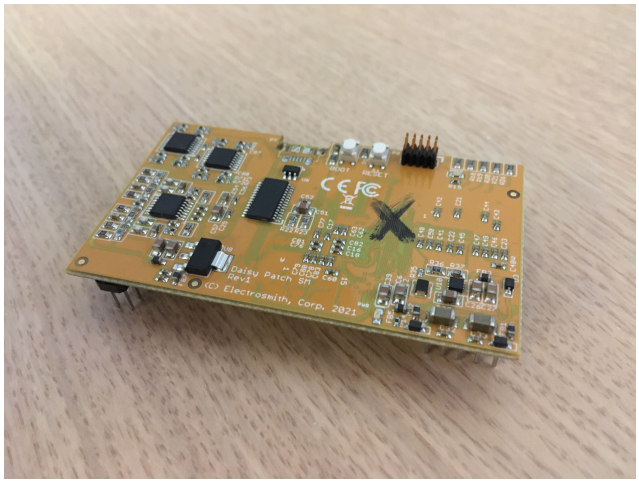

Figure 3: *The Daisy Patch Submodule board.*

The simulator is a thin abstraction layer between the user code and the target platform, and the latter can be either the virtual Eurorack environment or the Daisy Patch Submodule hardware board. It is done in a way that a program that can build for the virtual environment will also build for the real environment. The simulator can also detect problems that the firmware would have at run time, when still developing on the desktop computer.

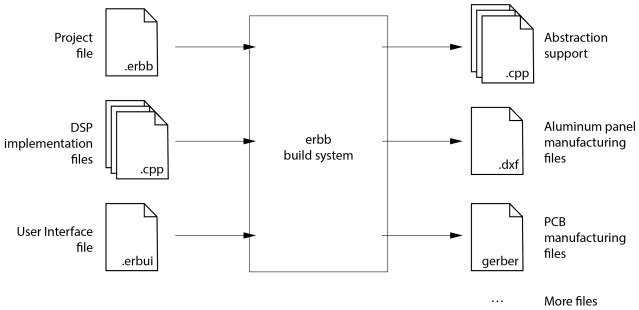

Figure 4: *The* ERBB *build system.*

On the software side, the main components of Eurorack-blocks are (see Figure **??**):

- A build-system called `erbb` and its associated ERBB language, which abstracts the target platform and produces makefiles or Integrated Development Environment (IDE) project files, to either build the software for the simulator or the firmware,
- A user interface system and its associated ERBUI language, which allows describing a custom hardware user panel,
- A set of generators that produce Printed Circuit Board (PCB) production files, instruction files for front aluminum panel milling and drilling machines, PDF files for front panel silkscreen printing from the ERBUI source code, and C++ files to provide language bindings between ERBB, ERBUI and C++.

The simulator is written in C++ while the build-system is written in Python. Developers can use C++ to write their DSP code, but also `gen~` from the Max/MSP visual environment[9], or the FAUST language.

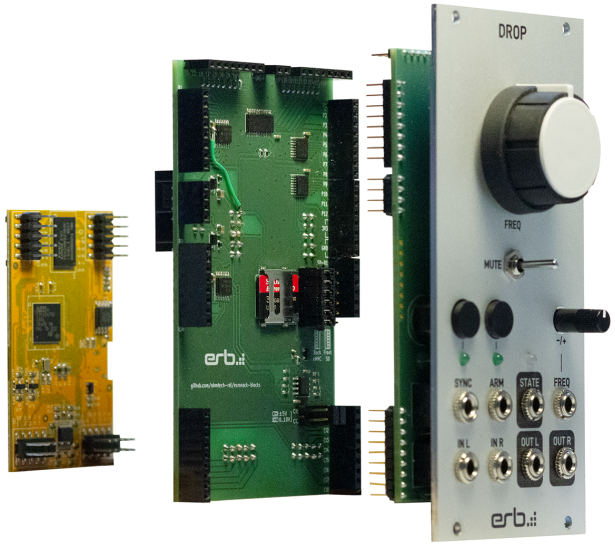

Figure 5: *From left to right, the Daisy Patch Submodule board, the Eurorack-block board "Kivu12", the front PCB tightened to the front aluminum panel with its jack connectors and pot.*

On the hardware side, a Eurorack-block project is composed of three parts (see Figure **??**):

- The Daisy Patch Submodule, which provides the computation and memory units, as well as a stereo audio codec, and all the circuitry needed to adapt to the Eurorack electrical specifications,
- A Eurorack-block board, which mediates between the Daisy Patch Submodule and the front panel user-defined user interface, by extending the capabilities of the former by using multiplexers and demultiplexers. It also provides power conditioning. It is designed for mass production,
- The front PCB and its aluminum panel, which hosts all the user-defined controls of the user interface.

1.2. Simulated Environment

Making hardware can be long for different reasons, one of them being the time it takes between the moment where a design file is ready to be produced, and the time where it can really be used. Producing all the different needed parts to assemble a module, such as the Printed Circuit Board or front aluminum panel, typically takes a bit more than 6 working days to be produced and received, after which the module still needs to be assembled.

This means that the feedback loop between a change and the effectiveness of that change is around 2 weeks. In contrast, the feedback loop to which software developers are used to usually ranges from a couple of seconds to some minutes. This only in itself explains why so many companies have moved from traditional hardware making to one in which the hardware could be changed quickly using a "firmware", which drastically reduces the time to market.

This process has been enhanced even more for platforms like iOS developed by Apple, where a software developer can test their application on a specific iPhone, without even owning it. Rather, a simulated environment is used, in which a Low-Fidelity (in terms of user interactions) prototype of the application can be developed on a desktop computer.

Eurorack-blocks uses the same method. It uses VCV Rack [10], a virtual Eurorack system that can run on a desktop computer running on macOS, Linux, or Windows. We made a small layer, called the *simulator*, which tries to emulate as much as it can the real hardware. This prototype is also Low-Fidelity but is enough for most uses. Apart from processing power, it ensures that a program that can run in the simulator will also run on the real Daisy hardware (see Figure **??**).

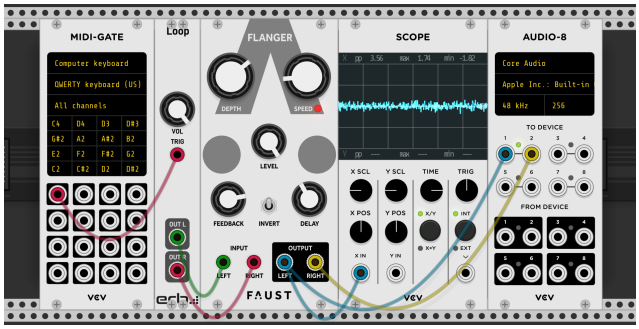

Figure 6: *The* FAUST *Flanger module running in VCV Rack. It is possible to input Audio in the Flanger, listen on the desktop connected sound interface, as well as monitor signal properties using an oscilloscope.*

Using VCV Rack is very interesting, as it offers a rich variety of modules that can be used for testing, such as LFOs, VCOs, oscilloscopes, and frequency spectrum.

The final generated module, once manufactured, can be used in a real Eurorack system (see Figure **??**).

1.3. Using the ERBB DSL

ERBB is a Eurorack-blocks Domain Specific Language, which aims to simplify building a module for both the simulator and the firmware, which have very different configurations. It is called a "meta-build-system" in the sense that it is not a build-system in itself like `make`, but generates build-system files for a variety of build systems, including `make`, but also Integrated Development Environments, for different targets (here either the simulator or the firmware).

[9] `https://docs.cycling74.com/max8/vignettes/gen_topic`

[10] `https://vcvrack.com`

Figure 7: *The finalized* FAUST *Flanger module running in a Eurorack system*

For example, the following simple file for FAUST:

```
// Flanger.erbb

use strict

module Flanger {
    sources {
        file "Flanger.erbui"
        file "Flanger.dsp"
    }
}
```

Will generate:

- A `Makefile` for `make` to build a VCV Rack module to allow testing of the module in a virtual environment,
- A `Makefile` for `make` to build the Daisy firmware that will be uploaded on the hardware,
- A Xcode project for the native macOS development environment, which builds a VCV Rack module and allows to debug it,
- A VS code workspace for Linux, macOS, and Windows, which can build a VCV Rack module, the firmware, and allows to debug the firmware on the target Daisy hardware.

Additionally, it will also generate "actions" for those build systems, which:

- Detect when an ERBUI file changed to re-generate everything needed that is used during the build process,
- Generate a specific representation of audio files that can be directly compiled *into* the program,
- Order the FAUST transpiler to generate updated C++ files when the source `dsp` file changed,
- And many other task management actions to automate common actions, such as copying the built VCV Rack module to the right place, so that developers can start to test the module as soon as the build process is finished.

1.4. Using the ERBUI DSL

ERBUI is a Eurorack-blocks Domain Specific Language, which aims to simplify the physical layout of the user interface of a module. It provides standard electronic components such as potentiometers, switches, LEDs, or the standard Eurorack-compatible jack 3.5mm mono connectors.

For example, the following simple file for FAUST:

```
module Flanger {
    width 12hp
    board kivu12
    material aluminum
    header {}
    footer {}

    image "panel.svg"

    control depth Pot {
        faust { bind {
            address "/FLANGER/0x00/Depth"
        }}
        position 13.5mm, 25.645mm
        style rogan.5ps
        pin P4
    }

    control speed Pot {
        faust { bind {
            address "/FLANGER/0x00/Speed"
        }}
        position 47.1mm, 25.645mm
        style rogan.5ps
        pin P5
    }
    ...
}
```

Will generate:

- The front aluminum panel DXF file, which gives information to milling and drilling machines to produce the front panel,
- The printed circuit board (PCB) "gerber" files, which gives information for manufacturer on how to produce the board, on which controls like potentiometers, switches LEDs, or the standard Eurorack-compatible jack 3.5mm mono connector will be soldered too,
- The bill of materials (BOM), to simplify the ordering of electronic components to global component suppliers,
- A PDF file that is used to digitally print on the front aluminum panel,
- A SVG file that is used for the VCV Rack module's silkscreen,
- The C++ code to generate the module UI for the VCV Rack SDK,
- The C++ code compatible with the firmware.

1.5. Using the Faust DSL

FAUST [**?**] as a Domain Specific Language, specifically designed for real-time signal processing and synthesis, is a good candidate to speed up and facilitate the development of the DSP part of a Eurorack-block module.

A FAUST program describes a signal processor taking a set of audio inputs and control signals (such as buttons or sliders) and generating a set of output audio signals and controls (such as bargraphs).

After the compilation step, the controls form a hierarchy of groups nested within each other as a tree, with the terminal leaves as sliders, buttons or bargraph items. The tree description is then generated in the `buildUserInterface(...)` method as a sequence of calls to construction methods (like `openHorizontalBox/closeBox` and `addButton/addVerticalSlider`) to be implemented by an external UI object.

This object can typically interpret the tree course to build a unique path (as `/root/group1/group2/.../item` kind of string) for each tree leaf [11], or create a JSON file to describe the same information.

From the DSP source program, the FAUST transpiler will be driven by the ERBB DSL to produce a C++ class, to be compiled with specific architectures files also generated by ERBB.

2. PROBLEM STATEMENT

On one hand, the DSP files of FAUST are programs that mix the presentation (e.g. primitives like `hslider`) and the computation. A program can import libraries, which contains functions, and those functions can possibly use presentation primitives as well, and in particular layout information (e.g. primitives like `hgroup`) therefore representing a directed tree. Those primitives have a label and represent a scalar value. Two labels can have the same name in a program, as long as they are in different DSP files.

The same directed tree and labels principles apply to audio samples through the use of the `soundfile` primitive, which supports only a non-interleaved audio representation at the time of writing this paper. The `soundfile` primitive supports however the concept of "parts", which conveniently join multiple distinct audio files, into one dynamic continuous view from the DSP program perspective.

On the other hand, the ERBUI file represents a flat namespace of controls, with each control having a name and a type. A type can represent a scalar value, but also for example a boolean value, a compound type (e.g. a 2-dimensional vector for a joystick control), or possibly an entire filesystem. The computation and presentation are separated and bound at build time.

The ERBB files represent the sources (e.g. DSP file) and resources (audio samples) again in a flat namespace. Interleaved and planar audio formats are supported, but the current model only supports a one-to-one mapping between one audio file and its representation in the program.

Finally, the total number of UI primitives of a program might not allow them to fit practically on a Eurorack panel, because of space considerations, or to limit the number of features to enhance the user experience. In that case, some FAUST UI primitive will be left unbounded on purpose, and in some cases, a user might need to set a default value that might not match the one originally defined in the program (specifically in the case of imported libraries).

2.1. Constraints

Eurorack-blocks uses the Daisy Patch Submodule hardware for computation. It is an embedded platform that, while being extremely powerful for embedded use, is also very slow compared to a desktop platform, and has a different memory model than a traditional desktop computer.

Three types of different memory on the embedded platform can be used for computation, with different sizes and speeds, the larger meaning being also the slower.

Because memory and computation are very precious resources, we shall avoid computing at run time what could be computed at build time.

Finally, while the approaches exposed in the problem statement are radically different, the implementation must unify them, without requiring the user to make any modification to their past programs or the standard library of DSP functions delivered with FAUST.

2.2. Previous Work

Embedded platforms for audio processing (as microcontrollers or specialized Linux systems running on more standard CPUs like ARM) have appeared in recent years. When they run a complete OS, they can usually be programmed using C/C++ and possibly by deploying standard audio environments like PureData, Csound and SuperCollider. Some of them only contain a minimal OS, and the audio processing code has to be directly developed in C/C++ and deployed on various kinds of *bare-metal* approaches (like having the audio code directly running in the audio interrupt for instance).

Most of those platforms can directly be used with FAUST when the appropriate `faust2xx` script exists, using a set of adapted architecture files. Here are some examples of recently developed tools:

- `faust2esp32` [**?**], a tool to generate digital signal processing engines for the ESP32 microcontroller family. It can target both the C++ and the Arduino ESP32 programming environment and it supports a wide range of audio codecs, making it compatible with most ESP32-based prototyping boards
- `faust2teensy` [**?**] is a command-line application that can be used both to generate new objects for the Teensy Audio Library and standalone Teensy programs.
- `faust2bela` [12] allows to build and run a Bela project, possibly using polyphonic mode and a remote GUI.

A `faust2daisy` [13] implementation is already targeting the Daisy platform. However, it is meant for existing ready-to-use pieces of hardware, such as the Daisy Pod, while we intend to provide a development platform for a piece of hardware that doesn't already exist.

Furthermore, the virtual environment mentioned earlier is VCV Rack, a virtual Eurorack system for desktop platforms. We built an abstraction that ensures that the same code will both run on the simulator (a thin abstraction layer to the VCV Rack module SDK in charge of emulating specific behaviors of the embedded hardware) and on the embedded hardware.

[11] This is typically used for OSC control

[12] `https://learn.bela.io/using-bela/languages/faust-experimental/`

[13] `https://github.com/grame-cncm/faust/tree/master-dev/architecture/daisy`

2.3. Proposed Solution

A hierarchical namespace can be easily turned into a flat namespace using *fully-qualified names*. This concept exists already in FAUST and is called an *address*, and is used for example in the OSC implementation. Similarly, the same concept can be used for ERBUI compound types. This can be done by introducing new keywords in the ERBUI language to map a control to a fully-qualified name in the FAUST program. The same approach will be used for the ERBB language when binding audio sample data.

The FAUST compiler can transpile its DSP programs into a JSON static representation. This representation, and the use of meta-programming, allows us to generate C++ files with already parts of the computation done, in a way very similar to constant folding in modern compilers. This method is used to bind FAUST program values to Eurorack-blocks controls and similarly for audio sample data.

3. METHOD

3.1. Architecture Files

Making a port to a new language or framework for FAUST relies on "architecture files" as a way to inject code into the generated FAUST C++ code. Architecture files are generic by nature: they are meant to be written once and then dynamically generate at run time whatever is needed by the underlying target.

Typically the FAUST generated function to build the user interface might look like this:

```
virtual void buildUserInterface(UI* ui) {
   ui->openVerticalBox("FLANGER");
   ui->openHorizontalBox("0x00");
   ui->addCheckButton("Bypass", &fCheckbox0);
   ui->addCheckButton("Invert Flange Sum",
↪   &fCheckbox1);
   ui->addHorizontalBargraph("Flange LFO",
↪   &fHbargraph0, -1.5f, 1.5f);
   ui->closeBox();
   ui->openHorizontalBox("0x00");
   ui->addHorizontalSlider("Speed", &fHslider1,
↪   0.5f, 0.0f, 10.0f, 0.00999999978f);
   ui->addHorizontalSlider("Depth", &fHslider5,
↪   1.0f, 0.0f, 1.0f, 0.00100000005f);
   ...
}
```

And a GTK+ architecture file would create a Checkbox button on the call to `addCheckButton` and analogously for the rest of the user interface.

Memory allocation is achieved similarly, as the FAUST generated code enumerates what allocations to perform. In particular, the latest version of the FAUST compiler gives additional information on the use of those memory blocks, by instructing how much time they are read or written during a DSP compute step.

3.2. Embedded Systems Constraints

While the previous approach is perfectly valid and simple most of the time for programs running on a desktop computer, it is more problematic on embedded processors.

The embedded system we use has 128K of stack memory, 512K of fast heap memory, and 64MB of slower heap memory, and has a single-core processor running at 480MHz. Therefore the usage of computation resources and memory is very limited compared to a desktop computer.

In the example program above, to bind a `hslider` value to its ERBUI counterpart, we would have to make a small algorithm similar to a simple parser, which is more or less fine but still could use resources that would be hard to reclaim afterward.

This is more problematic for the memory manager, as every time we receive an `allocate` call we need to decide whether to put it in the fast SRAM region or the slower SDRAM region. At the time the `allocate` function is called, and without knowing what will be the future `allocate` calls, we can't make an optimal placement of blocks into memory regions.

For example, consider the following FAUST generated calls:

```
// a delay line accessed only once
memory_manager->allocate (64 * 1024);

 // another delay line accessed only once
memory_manager->allocate (64 * 1024);

// some filter memory accessed multiple times
memory_manager->allocate (12);
```

With a naive heuristic, the two first allocations would fill the entire fast SRAM region, when the last memory block which would benefit from fast SRAM is allocated to the slow SDRAM region.

The latest version of the FAUST compiler generates code that indicates how blocks are accessed, to enable better decision-making, and works in a two-phase manner:

- It first enumerates all memory blocks with their access recurrence and size,
- It then asks the DSP memory manager to allocate those blocks.

This decision problem is very well known in operational research as the *bin packing* optimization problem. Computationally, this problem is NP-hard, but optimal solutions to large problem instances can be produced with sophisticated algorithms, and many approximation algorithms exist as well.

3.3. Moving Computation at Build Time

Solving this problem is however not really adapted to our small embedded processor, and we can observe too that this problem does not need to be solved dynamically either: since FAUST can produce this information when transpiling the DSP file, it is possible to solve this problem at build time rather than run time. This approach makes a huge difference as building the program is done on a powerful desktop computer.

We therefore propose this novel approach in FAUST, to relocate to build-time language bindings, parameter value mappings, and memory management pre-computations, that do not need to execute at run time, to save crucial memory and computation resources on the target embedded platform.

In practice, our system generates C++ code that matches exactly the module that is currently produced, from the ERBUI and ERBB files. We use the `-json` option of the FAUST compiler, which produces a declarative description of the FAUST DSP program, that we can use to generate perfectly-tuned C++ for the embedded platform. An example of the generated code can be found in Appendix A.

At the time when we first implemented the FAUST integration, the new memory manager interface was not available, and the equivalent declarative JSON memory blocks description was not available either. We decided to implement a simple greedy algorithm for now and enhance memory management in future work.

4. IMPLEMENTATION

4.1. Overview

ERBUI ignores most layout information dictated from the FAUST program, such as `hgroup` or `vgroup`, the orientation in `hslider` and `vslider`, or the `style` of a primitive. This allows a clean separation between the ERBUI definitions presentation layer and the FAUST program computation layer.

ERBUI and ERBB introduce a few new keywords specifically for FAUST:

- `faust` represents a lexical scope for definitions specific to FAUST,
- `bind` represents a lexical scope in the FAUST scope to define a binding,
- `init` represents a lexical scope in the FAUST scope to define an overridden initial value for a FAUST primitive,
- `value` represents a scalar value ,
- `address` represents the fully-qualified name of a primitive in a FAUST program,
- `property` represents the property name in an ERBUI compound type.

4.2. Name Mapping

Let's consider the following FAUST program:

```
import("stdfaust.lib");
process = dm.flanger_demo;
```

When running `faust` with the `-json` option we get for example for the Speed pot:

```
{
    "type": "hslider",
    "label": "Speed",
    "address": "/FLANGER/0x00/Speed",
    "meta": [
        { "1": "" },
        { "style": "knob" },
        { "unit": "Hz" }
    ],
    "init": 0.5,
    "min": 0,
    "max": 10,
    "step": 0.01
}
```

The address field `/FLANGER/0x00/Speed` represents the fully-qualified name of the Bypass checkbox in the `dm.flanger_demo`, and is guaranteed to be unique in the whole program.

The following ERBUI source code expresses a binding from a Pot with label SPEED on the Eurorack module, with the Speed FAUST primitive inside `dm.flanger_demo`.

```
control speed Pot {
    faust { bind {
        address "/FLANGER/0x00/Speed"
    }}
    position 6hp, 34mm
    style rogan.6ps
    label "SPEED"
}
```

An example of compound types in ERBUI is a dichromatic LED. It has a `r` and `g` property which denotes the red and green LED part intensity, respectively:

```
control led_bi LedBi {
    faust {
        bind {
            property r
            address "/Phaser/LED Red"
        }
        bind {
            property g
            address "/Phaser/LED Green"
        }
    }
    position 10hp, 80mm
    style led.3mm.red_green
    label "LED"
}
```

4.3. Implicitly-defined Default Binding

If no user-defined bindings are provided for a control, it is defined by the ERBB transpiler, and has the same effect as a user-defined binding with address `"{module}/{control}"` where `{module}` is the name of the module in which the control is defined, and `{control}` is the name of the control. That is, it defines the binding that matches the address FAUST would generate for a primitive in the root function.

For example the FAUST program:

```
// LowPass.dsp

import("stdfaust.lib");

fc = hslider("freq", 1000, 100, 10000, 1);
process = fi.resonlp(fc,1,0.8);
```

Will generate the address `Lowpass/freq` for the `hslider` primitive, and the ERBUI source code:

```
module LowPass {
    control freq Pot {
        position 6hp, 34mm
        style rogan.6ps
        label "FREQ"
    }
}
```

implicitly defining the default binding with the same address `Lowpass/freq`, matching the FAUST generated one.

4.4. Initial Value

When a FAUST primitive in a function of the standard library needs to have an initial value different from the one originally defined in

the library functions, the `init` keyword can be used to override the FAUST-defined value, at module scope:

```
module Flanger {
   ...

   faust { init {
      address "/FLANGER/Delay Controls/Delay
      ↪  Offset"
      value 1
   }}
}
```

4.5. Parameter Value Mapping

The implementation takes into account the minimum and maximum value of a property, as well as its scale (e.g. linear, logarithmic or exponential).

The type of the FAUST primitive is used to provide a scalar value to FAUST following the expectation of the end-user. For example, a `checkbox` type will inform Eurorack-blocks to take the boolean value and convert it to a scalar value, while the `button` type will inform Eurorack-blocks to deliver a scalar value that corresponds to a *rising-edge* of the boolean value.

4.6. Audio Input/Output Mapping

The FAUST DSP defines several buffers per channel, for both input and output. They are mapped to the ERBUI file by order, so that the first audio input channel would map to the first `AudioIn` definition in the ERBUI file, for example.

4.7. Audio Files

FAUST supports only planar (non-interleaved) audio formats, so the ERBB transpiler generates a planar representation suitable for FAUST. This representation can have a performance impact while fetching data, because of its access pattern and the underlying memory chip. But we think this is good enough for a first implementation.

We also ignore the audio file URL described in the `soundfile` metadata and refer to the ERBB definition for that.

4.8. Memory Management

Three regions of memory can be commonly used by the firmware:

- The fast 128KB `DTCMRAM` which runs at full processor speed,
- The 512KB `AXI SRAM` which runs at half of the processor speed,
- The slow 64MB `SDRAM` which is external to the processor.

In general, values that are read and written often, such as filter histories should be written in the fastest RAM. Since this should be small and directly part of the DSP state, we elect the fast `DTCMRAM` to be our stack memory and put the DSP state on it [14].

The rest of the available RAM should be used for everything else.

[14] This is not always the case, unfortunately, as some delay lines might be stored on it.

4.9. Custom Memory Manager

In C and C++, the FAUST compiler produces a class (or a struct in C), to be instantiated to create each DSP instance. The standard generation model produces a flat memory layout, where all fields (scalar and arrays) are simply consecutive in the generated code (following the compilation order).

On audio boards where the memory is separated as several regions (like SRAM and SDRAM) with different access times, it becomes important to refine the DSP memory model so that the DSP structure will not be allocated on a single region of memory, but possibly distributed on all available regions. The idea is then to allocate parts of the DSP that are often accessed in fast memory and the other ones in slow memory.

This is done by using a special `-mem` compilation option which adapts the way the C++ code is generated, and interacts with an external `dsp_memory_manager` object which will distribute the different needed memory zones (like tables or delay lines) on different memory regions.

A custom `dsp_memory_manager` which implements a simple monotonic allocator heuristic has been implemented: since FAUST doesn't allocate or deallocate memory during the computation phase, we can use a greedy algorithm which allocates portions of memory in the `AXI SRAM` as long as they fit in it, and use the `SDRAM` for everything else.

4.10. Code Generation

Both ERBB and ERBUI have non-ambiguous grammars which make them suitable for use with a Parsing Expression Grammar. We generate an Abstract Syntax Tree from the input ERBUI or ERBB file, which goes to an analyzer, and then multiple generators.

The analyzer provides the semantic analysis, and in the context of FAUST, verifies for example that the number of audio inputs and outputs of the FAUST DSP file matches the ones defined in the ERBUI file.

We have two generators for FAUST, one for ERBB and one for ERBUI:

- The ERBB generator generates the C++ FAUST integration layer, as well as audio sample bindings,
- The ERBUI generator generates the C++ UI primitive bindings.

Those generators rely on the fact that the order of definition in the JSON file follows the same order of binding calls (such as `addButton`, `addVerticalSlider` or `addSoundfile`). This allows to avoid to maintain a nested state, as `Box` are `open` and `close` following the program primitive tree structure, which would require some allocations while binding the FAUST primitives to their Eurorack-blocks counterparts.

For example for language binding, we need to adapt between Eurorack-blocks internal storage, which can have various types (like a `float` or boolean) and the FAUST representation which is always a scalar `float` value. For this we have a table of `float*` values, for which each element points to the internal FAUST value representation, and which table is as big as the number of FAUST UI primitives:

```
// In code_template.h of the Erbb Faust
↪  generator
size_t decl_index = 0;
std::array <float*, %faust.widgets.length%>
↪  parameters;
```

`%faust.widgets.length%` is a syntax in the template to indicate the generator to replace this sequence of characters by the numbers of FAUST UI primitives. We deduce this value from the FAUST JSON generated files which allow us to count the total number of widgets in a FAUST program.

We have a representation of the FAUST JSON tree in python, that we deduced from the JSON file:

```
class FaustDsp:
    def __init__ (self):
        self.nbr_inputs = 0
        self.nbr_outputs = 0
        self.widgets = []

class FaustWidget:
    def __init__ (self, type_, address, min_val,
    ↪  max_val, scale):
        self.type_ = type_
        self.address = address
        self.scale = scale
```

Once we have read the FAUST JSON file and created the corresponding python structure, generating the C++ code from the template is just text replacement:

```
# (Simplified generator code for readability)
def generate_module_declaration (self, path,
↪  faust_dsp, module):
    path_template = os.path.join (PATH_THIS,
    ↪  'code_template.h')
    path_output = os.path.join (path, '%s.h' %
    ↪  module.name)

    with open (path_template, 'r',
    ↪  encoding='utf-8') as file:
        template = file.read ()

    template = template.replace
    ↪  ('%faust.widgets.length%', str (len
    ↪  (faust_dsp.widgets)))
    ...

    with open (path_output, 'w',
    ↪  encoding='utf-8') as file:
        file.write (template)
```

Since the order of UI primitives in the JSON file is the same as the FAUST generated C++ code, we can keep track of which index in the `std::array` represents which UI primitive in the FAUST program and which control in Eurorack-blocks. Then, as this information is kept in the generator, the C++ code generation is rather straightforward. For example the FAUST function `addButton` is implemented as:

```
void Adapter::addButton(const char* /* label_0
↪  */, float* zone)
{
    push_parameter (zone);
}
```

And we can see that we completely ignore the label, and only keep the memory zone of where FAUST stores the scalar value. Binding is then straightforward as well:

```
void Adapter::push_parameter(float* zone)
{
    parameters [decl_index] = zone;
    ++decl_index;
}
```

Every time we want to process a new block of audio samples, we need to get the Eurorack-blocks UI control values and set them in the FAUST memory zone before we call the FAUST DSP `compute` function. As we explained above, since we keep a mapping between FAUST UI primitive indexes and Eurorack-blocks control names, we can simply generate the code below. For example, we know that the 0-based index 3 in the FAUST program maps to the control `speed` in Eurorack-blocks.

```
void Adapter::preprocess ()
{
    *module.adapter.parameters[1] = ((0.000000f +
↪   1.000000f * (module.ui.invert.position_first
↪   () ? 0.f : (module.ui.invert.position_center
↪   () ? 0.5f : 1.f))));
    *module.adapter.parameters[3] = ((0.000000f +
↪   10.000000f * module.ui.speed));
    *module.adapter.parameters[4] = ((0.000000f +
↪   1.000000f * module.ui.depth));
    *module.adapter.parameters[5] = ((-0.999000f
↪   + 1.998000f * module.ui.feedback));
    *module.adapter.parameters[6] = ((0.000000f +
↪   20.000000f * module.ui.flange_delay));
    *module.adapter.parameters[8] = ((-60.000000f
↪   + 70.000000f * module.ui.level));
}
```

The above code also contains the parameter value mapping, which takes into consideration the meta minimum and maximum value as defined in the FAUST code. We can also generate the C++ mapping code directly. One will note that the mapping function is affine, and therefore the simplest representation, as we can do pre-calculations in the code generator. This is also useful for output UI primitives like `hbargraph`, as the mapping contains a division, but we can therefore pre-compute it at build time, and keep an affine function as well:

```
// Generated code
void Adapter::postprocess ()
{
    module.ui.led = (0.500000f + 0.333333f *
↪   *module.adapter.parameters[2]);
}
```

Sample binding with the FAUST `soundfile` primitives is implemented exactly with the same concepts.

5. CONCLUSION

The resulting FAUST compatibility layer is incredibly thin and matches totally our abstraction without requiring any changes to it. In particular, the clean separation between the presentation layer and computation layer allows immediate benefits for FAUST user every time a new UI feature is developed into ERBUI, with, in most cases, no changes to the ERBB FAUST integration.

We identified and solved two main user stories:

- FAUST users who write new programs benefit from the implicitly-defined default binding, as long as they use primitive labels that can be also ERBUI or ERBB identifiers,

- FAUST users can port past programs without any changes by using our new FAUST-specific keywords.

In our opinion, this represents the "best of both worlds" while providing a cohesive abstraction.

Audio sample usage benefits from the Eurorack-blocks automation that compiles the audio sample directly into the program, simplifying greatly samples management for small samples.

6. FUTURE WORK

The current implementation for memory management expects the FAUST DSP state to fit on the 128K stack of the Daisy platform. This is not the case for DSP modules such as the FAUST reverbs. Additionally, some work has been done on FAUST to include memory management hints into the JSON file, so that our allocator could make better decisions on where to allocate memory, at build time, using a proper bin-packing heuristic. We will address both issues in future work.

The current implementation for `soundfile` doesn't support parts. For this, we will support the Cue/Splice markers in the WAV audio format, which is exactly meant for this purpose. Furthermore, this will allow having continuous data stored in memory rather than just a view on those data, which simplifies the implementation.

7. ACKNOWLEDGEMENTS

The authors would like to thank Eliott Paris who kindly reviewed the earlier version of this manuscript and provided valuable suggestions and comments.

8. APPENDIX A

```
// Flanger_erbui.hpp
// Implements name binding, parameter value mapping, DSP computation
// Comments were removed for readability

void Adapter::addButton(const char*, float* zone)
{
   push_parameter(zone);
}

void Adapter::addCheckButton(const char*, float* zone)
{
   push_parameter(zone);
}

void Adapter::addVerticalSlider(const char*, float* zone, float, float, float, float)
{
   push_parameter(zone);
}

void Adapter::addHorizontalSlider(const char*, float* zone, float, float, float, float)
{
   push_parameter(zone);
}

void Adapter::addHorizontalBargraph(const char*, float* zone, float, float)
{
   push_parameter(zone);
}

void Adapter::addVerticalBargraph(const char*, float* zone, float, float)
{
   push_parameter(zone);
}

void Adapter::addNumEntry(const char*, float* zone, float, float, float, float)
{
   push_parameter(zone);
}

// name binding runtime "algorithm"

void Adapter::push_parameter(float * zone)
{
   parameters[decl_index] = zone;
   ++decl_index;
}

// parameter value mapping

void Adapter::preprocess()
{
   *module.adapter.parameters[1] = ((0.000000f + 1.000000f * (module.ui.invert.position_first () ? 0.f : (module.ui.invert.position_center () ? 0.5f : 1.f))));
   *module.adapter.parameters[3] = ((0.000000f + 10.000000f * module.ui.speed));
   *module.adapter.parameters[4] = ((0.000000f + 1.000000f * module.ui.depth));
   *module.adapter.parameters[5] = ((-0.999000f + 1.998000f * module.ui.feedback));
   *module.adapter.parameters[6] = ((0.000000f + 20.000000f * module.ui.flange_delay));
   *module.adapter.parameters[8] = ((-60.000000f + 70.000000f * module.ui.level));
}

void Adapter::postprocess()
{
   module.ui.led = (0.500000f + 0.333333f * *module.adapter.parameters[2]);
}

// DSP computation

void  Flanger::process()
{
   adapter.preprocess();

   const float* const in[] = {
      &ui.audio_in[0], &ui.audio_in2[0]
   };

   float* const out[] = {
      &ui.audio_out[0], &ui.audio_out2[0]
   };

   auto in_faust = const_cast <float**> (in);
   auto out_faust = const_cast <float**> (out);

   dsp.compute(erb_BUFFER_SIZE, in_faust, out_faust);
```

```
    adapter.postprocess();
}
```