DSP2JS

A C++ framework for the development of in-browser DSPs

Oliver Major
Fraunhofer IIS
Am Wolfsmantel 33
91058 Erlangen, Germany
oliver.major@iis.fraunhofer.de

ABSTRACT

We present DSP2JS, a new framework for the development of audio signal processors for web platforms using Emscripten and the WebAudio API. In particular, the goal is to abstract common functionality in a configurable layer that manages the communication between a JavaScript application and DSP code written in C or C++. The framework includes functionality for the creation, connection and management of processing units, runtime profiling, buffer management, buffer conversion and a configurable build system.

The proposed three-step development of a signal processor with DSP2JS allows for external libraries to be included, making it possible to port existing code to the framework. The generated artifacts can then be used in a web page and invoked via an interface similar to native WebAudioNodes. The optional omission of WebAudio bindings via a bare-build mode potentially opens up the core framework to further DSP applications, even outside of the audio domain.

We examine the multilayered architecture of the core framework and the build system, also discussing design and implemetation decisions.

1. INTRODUCTION

We present a novel method to develop and port Audio DSP code to the web. The DSP2JS framework addresses issues of bridging audio processing, based on C or C++ to browser-based platforms with efficient buffer management and provides a familiar interface for JavaScript developers using the WebAudio API. It uses the Emscripten compiler to transpile the code to WebAssembly or asm.js. Furthermore, it provides JavaScript binding code for WebAudio using the AudioWorklet specification [2] to enable web developers to integrate DSP2JS-based packages into their existing or new projects.

DSP2JS is developed and maintained as an internal tool at Fraunhofer IIS, although discussions to open-source it are ongoing. Since a significant amount of our software is written in C, the framework has a main focus on porting existing libraries. To facilitate this, one of the design goals of the project is to be a functionally rich layer between the JavaScript application and the DSP code and bridge the gap between these two parties in a largely opaque manner with as little code as possible. It uses object-oriented design principles to achieve a relatively high abstraction level, being configurable and flexible. As such, it performs the most general tasks like buffer management, conversions between fixed-point and floating-point representations and profiling, which are independent of the application logic of the DSP.

This distinguishes DSP2JS from the currently available Web Audio Modules [3]. These allow the user to define callbacks for most important events using a C API, but do not provide any functionality on their own. While developers using Web Audio Modules get the most essential, yet flexible framework and have to define their own binding code and build process, DSP2JS aims to be a more abstract, but still performant alternative.

Figure 1 shows a general overview of the framework and its elements. The user interacts with a web page or JavaScript application, which can prompt the global DSP2JS object to spawn new AudioWorkletNodes representing processing units of a specific kind. They can then be inserted into an audio graph like any other WebAudio Node.

For every Node, the JavaScript runtime spawns a corresponding AudioWorkletProcessor in the rendering thread. This object has access to the compiled and instantiated DSP2JS Module, which it uses to create an internal representation of the processor inside WebAssembly. The AudioWorkletProcessor forwards global parameters (e.g. samplerate and data types), local parameters (e.g. channel count and profiler settings) and user-defined parameters (e.g. the gain in a custom gain node) to the WebAssembly object. When it is called to process an audio buffer, it copies the buffer into the WebAssembly memory and calls the framework's process function.

The DSP2JS core is the hub of the framework. It is implemented in C++ and intercepts calls from the AudioWorkletProcessor to WebAssembly to perform common tasks like profiling and buffer conversion before forwarding calls to the underlying processor. Custom calls are left unaltered. The processor then only needs to perform the specific DSP operations. If an external library can be compiled to WebAssembly, it is also possible to link it in and call it from the processor class.

In this paper we describe the framework from an implementer's point of view in Section 2. We dive into more technical details in Section 3, before discussing some of the decisions made in the project in Section 4.

Web Audio Conference WAC-2018, September 19–21, 2018, Berlin, Germany.

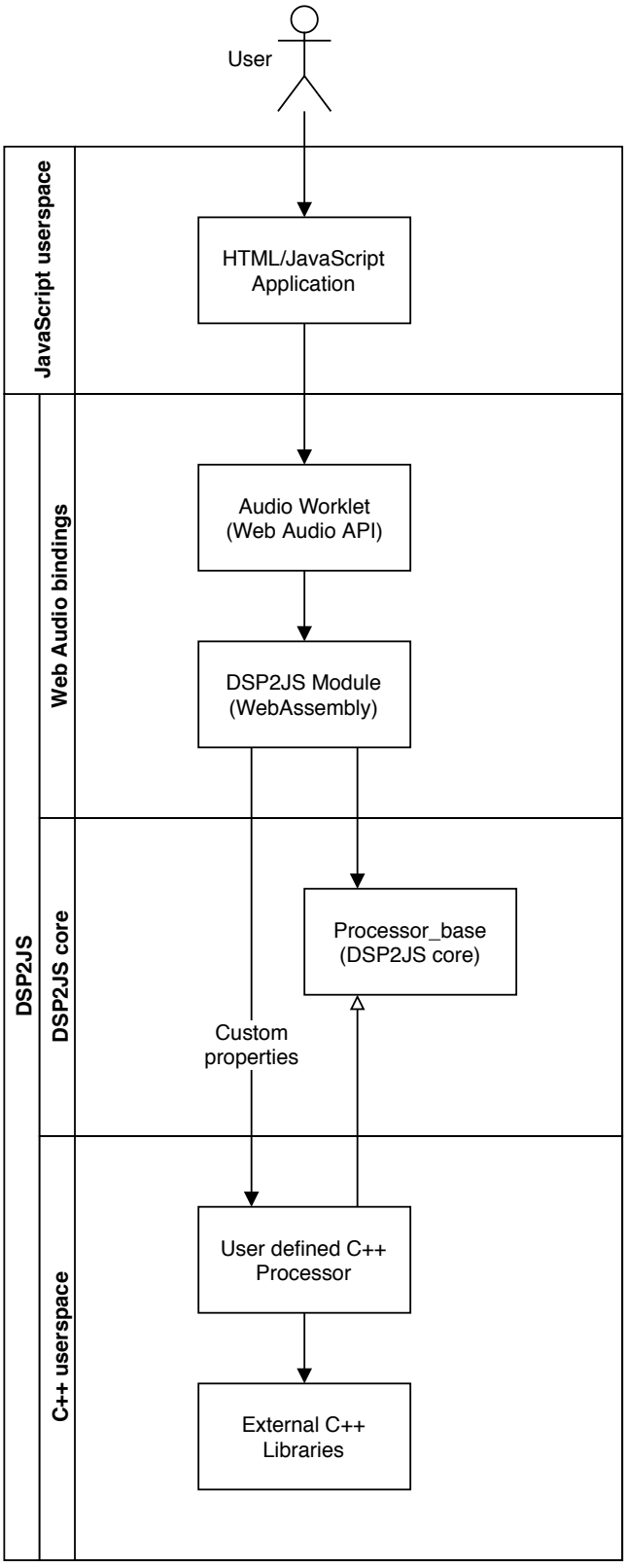

Figure 1: Overview of the DSP2JS framework

2. DEVELOPMENT CYCLE

The DSP2JS framework aids in the development of C and C++ based Audio DSP code. A *DSP developer* writes some application logic and calls DSP2JS's build system to generate two deliverable JavaScript files. A *web developer* can then include the artifacts to have the DSP functionality available as a WebAudio Node. In this section we describe the process of working with DSP2JS-based packages from these two perspectives.

2.1 Web developer

We illustrate the web developer's point of view with an exemplified gain reduction plugin in Listing 1.

```
1 const GRN = DSP2JS.GainRedNode;
2
3 async function runDemo() {
4   let actx = new AudioContext();
5   await GRN.registerContext(actx);
6
7   let src  = new OscillatorNode(actx);
8   let proc = new GRN(actx);
9
10  src.connect(proc)
11     .connect(actx.destination);
12  src.start();
13
14  proc.activateProfiler();
15  proc.reduction = 2;
16 }
17
18 runDemo();
```

Listing 1: GainRed.html

The web developer is given a package consisting of two JavaScript files. `GainRedNode.js` has to be loaded on a website using a script tag and now gives the developer access to the global `DSP2JS` object. This object contains a field `GainRedNode` which has to be registered with an AudioContext, as seen in line 5. Optionally, the developer can pass the path to the second file, `GainRedProc.js`, if it is not in the root folder. Afterwards, a Node can be created and connected like a regular AudioNode, as seen in lines 8 and 10. The node also provides access to DSP2JS functions and parameters (such as `activateProfiler` in line 14) and user-defined functions and parameters (such as `reduction` in line 15).

2.2 DSP developer

The development process for the DSP developer to create the artifacts consists of three steps, which are again exemplified with the gain reduction plugin.

The first step is to create the C++ processor like in Listing 2. The developer has to include the file `DSP2JS.h` and implement a derived class from `Processor_base`, as seen in line 5. This class has to provide a constructor which configures the base class with buffer properties such as data type and interleaving mode (line 9). The implementation has to override the `process` method, taking two arguments of type `BufferProxy` (line 15), casting them to the specified type (lines 16 - 17) and performing the DSP operations (line 20). The method is declared `const` to enforce public parameters not to change during the call. Internal state that is altered in the processing has to be marked `mutable`. At any time, the class can ask the framework for its internal parameters, such as buffer length, types and channel counts (line 18).

```cpp
#include <emscripten/bind.h>
#include "DSP2JS.h"
using namespace DSP2JS;

class GainRed : public Processor_base {
  short red = 1;

public:
  explicit GainRed() : Processor_base(DataType::INT16, InterleaveMode::Sequential) {}

  void setRed(short gr) {red = gr;}
  short getRed() const {return red;}

protected:
  void process(BufferProxy inProxy, BufferProxy outProxy) const override {
    auto ib = static_cast<short *>(inProxy);
    auto ob = static_cast<short *>(outProxy);
    auto  L = getBufferLength();

    for (int i = 0; i < L; ++i) {ob[i] = ib[i] / red;}
  }
};

EMSCRIPTEN_BINDINGS(GainRed) {
  emscripten::class_<GainRed, emscripten::base<Processor_base>>("GainRed")
    .constructor<>()
    .property("reduction", &GainRed::getRed, &GainRed::setRed);
}
```

Listing 2: GainRed.cpp

```json
{
  "name": "GainRed",
  "members": [
    {
      "name": "reduction",
      "type": "data",
      "init": 1
    }
  ],
  "options": {
    "numberOfInputs": 1,
    "numberOfOutputs": 1,
    "channelCountMode": "max",
    "channelCount": 1
  }
}
```

Listing 3: GainRed.json

The class can also hold custom data such as the gain reduction divisor `red` (line 6). Finally, the class and all parameters that shall be visible to the web developer need to be exposed by Emscripten's binding mechanism, as shown in lines 24 - 28.

The second step is the description of the gain reduction node as shown in Listing 3. Here, the DSP developer defines the public interface of the WebAudio Node as seen by the web developer. For this step, a .json file has to be provided with a `name` field, a `members` array and an `options` object with numberOfInputs, numberOfOutputs, channelCountMode and channelCount fields.

In case of external dependencies, the developer needs to compile all external libraries with Emscripten as preparation for the final build step. Listing 4 shows an example Makefile with the necessary configurations: the project name, DSP2JS root path, source path and optional paths for includes, external libraries and build artifacts. At the end, the developer includes `Makefile.base` from the DSP2JS folder. Running `make` will then generate the deliverable artifacts.

```
PROJECT_NAME=GainRed
DSP2JS_ROOT=../DSP2JS

SRC_DIR=./src
INCLUDE_PATHS=
EXT_LIBS=
BUILD_DIR=./dist

include $(DSP2JS_ROOT)/Makefile.base
```

Listing 4: Makefile

3. IMPLEMENTATION

In this section, we take a closer look at the technical details of how DSP2JS is implemented. A class diagram showing the general architecture of the framework can be seen in Figure 2, which will be described in the following.

3.1 DSP2JS core

At the lowest abstraction level and on the bottom right in the diagram, we find the `Data<T>` class template, which is used to represent a value in memory. The template parameter T is of enum type `DataType` and can be either of `INT8`, `INT16`, `INT32`, `FLOAT32` or `FLOAT64`. `Data<T>` is a thin wrapper around a corresponding base type such as `short` or `float` with additional functions for casting one data type to another under consideration of scaling and clamping of the values, thus providing a notion of an audio sample.

The `Buffer<T>` class template is the abstraction of an array of `Data<T>` values combined with metadata such as length, channel count and its interleaving mode. It also provides a mechanism for lazy allocation of the memory buffer to hold its values and to generate memory views for internal (C++) and external (JavaScript) use.

It derives from the `Buffer_base` class in what we refer to as the *base-generic-pattern* in this paper: an enum describes all the variations of the concept we want to implement, such

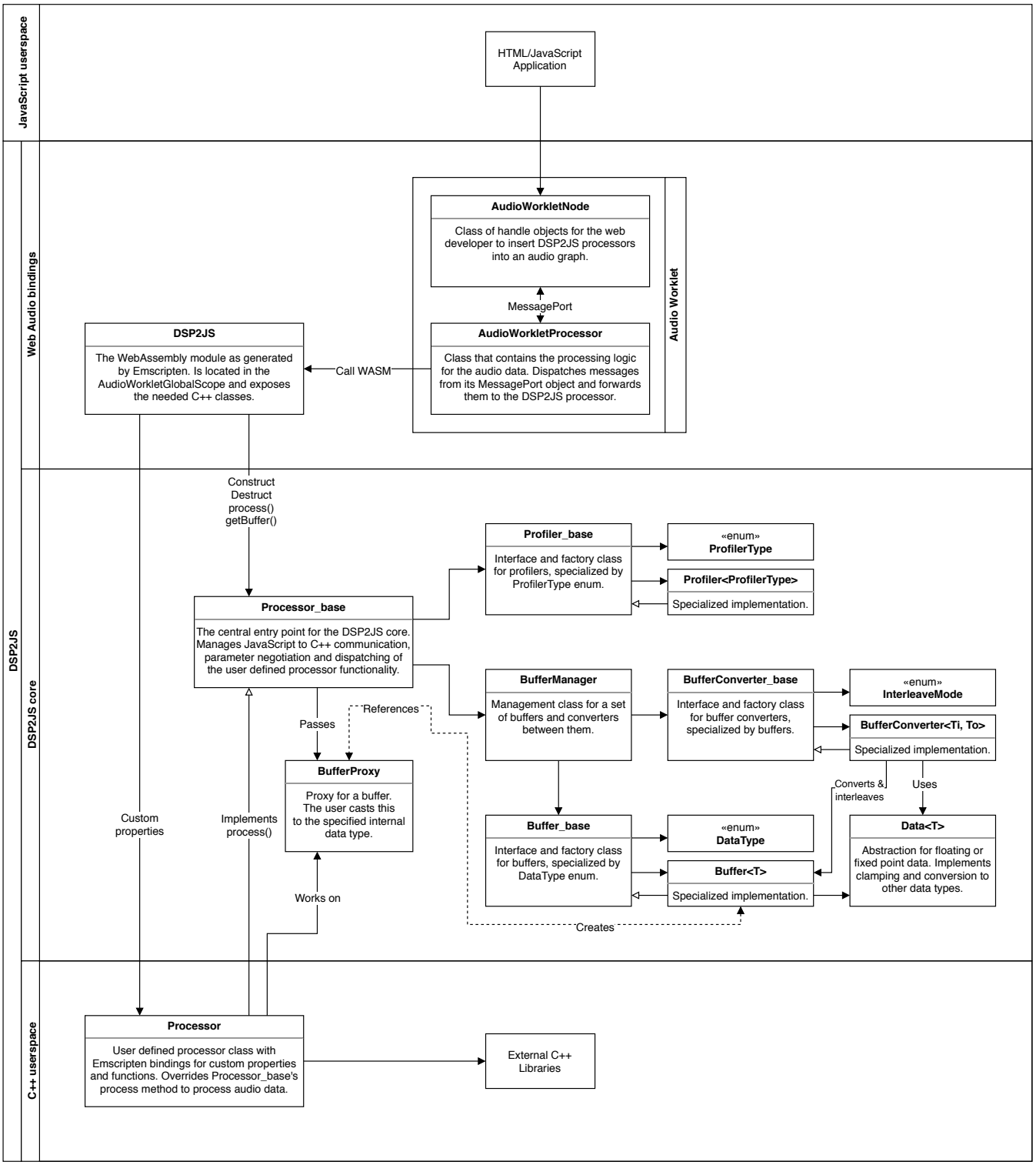

Figure 2: Detailed class diagram

as the data types in this particular case. The *generic class* is a class template which uses the enum for its specializations. It implements the concrete behavior we want to achieve, for example `Buffer<T>` for a specific data type T. It publicly derives from the *base class*, which is an abstract class defining the interface for its subclasses. Furthermore, it implements a factory method to generate subclass instances based on a parameter of the enum type. The user of the pattern only knows about this base class and invokes all behavior through it. Since all specializations of the generic class can be generated by the factory method, explicit instantiations will be placed in the compilation unit where the factory method is defined. Additionally, since the implementation of the subclasses uses generics, they can be optimized efficiently, while still being hidden behind a non-generic abstract interface, which works well with Emscripten's binding mechanism.

The same pattern is used for the `BufferConverter` classes: we define a set of functors with the abstract interface `BufferConverter_base` and specializations implementing `operator()` depending on the buffer types, the interleaving modes and the output buffer being identical or different from the input buffer. The specialization is chosen in the factory method, so the converters can be called efficiently without additional runtime checks.

All of the explained behavior for data abstraction is encapsulated in the `BufferManager` class. It manages a set of four buffers, one for each combination of input/output and internal/external buffers. The external buffers represent the buffers used by the JavaScript code to pass inputs in and get outputs out. Similarly, the internal buffers represent the buffers used in C++. To bridge between the internal and external buffers, the buffer manager uses two converters: one to convert the input data from its external to the internal representation and one to convert the internal representation of the output back to the external representation.

Another DSP2JS core functionality is profiling, implemented by profiler classes using the base-generic-pattern. We provide a runtime profiler for short-term and average load calculation and a dummy profiler in case no profiling is needed. The profiler can be chosen by passing a `ProfilerType` to the factory method, which can be either `RUNTIME` or `NONE`. The interface has start and stop methods using Emscripten's timing methods and a getData method for the extraction of the profile.

When asking a generic `Buffer<T>` class for an internal representation, it creates an object of the `BufferProxy` class, which is then passed to the DSP code's process function. A buffer proxy object references the buffer that created it, is non-copyable and can be cast to the base type of the referenced buffer. It does two checks: the cast has to be performed to the internal base type only and the destructor checks if a cast was performed at all. It is thus mandatory for the DSP programmer to take the two `BufferProxy` objects in the process function and cast them to the correct base type, otherwise the proxy would throw an error. Additionally, the buffer proxies help to hide the buffer classes, which should be used in the DSP2JS core only.

Lastly, the general interface of the core is defined by the `Processor_base` class. The DSP developer implements the processor deriving from this class and passes the internal buffer properties through the superclass constructor. The C++ interface is realized by the template method pattern, which is used to intercept calls to the DSP and inject behavior such as profiling and buffer conversion, as well as data such as the internal buffer views via `BufferProxy` objects into the calls. The class also exposes setters for external parameters, accessors for external buffers and a process function to the JavaScript code. It can be called by JavaScript through the WebAssembly module.

The `Processor_base` class manages all communication between the DSP's C++ code and the JavaScript's WebAssembly module. In the following section the calling conventions and usage is described from the JavaScript view.

3.2 WebAudio bindings

After the DSP2JS core is compiled into a static library, it gets linked to the DSP code via the Emscripten linker. This is done by the framework's Makefile.base, which has to be included in the DSP developer's Makefile. Depending on the optional `BARE_BUILD` flag, the framework can generate 3 types of output.

If `BARE_BUILD` mode is set to 2, DSP2JS links the DSP code, external libraries and the DSP2JS library into a WebAssembly file using Emscripten's `SIDE_MODULE` option. This can be used to obtain a maximum of control over the compilation and instantiation of the DSP code on a web page at the cost of having to implement all loading and binding code manually.

If `BARE_BUILD` mode is set to 1, Emscripten generates a JavaScript file in addition to the WebAssembly file. This file contains a minimal amount of wrapper code as generated by Emscripten with the option `MODULARIZE_INSTANCE`. The wrapper code asynchronously loads the WASM module, compiles and instantiates it, but does not create WebAudio bindings. This mode is suited if you want to use the DSP with your own WebAudio binding code or without the WebAudio API, possibly for testing or in non-audio related domains.

If `BARE_BUILD` mode is set to 0 or unset, the framework generates two JavaScript files, one containing an AudioWorkletNode and one containing the AudioWorkletProcessor with the respective names xxxNode.js and xxxProc.js.

The processor file is generated by Emscripten using the options `MODULARIZE_INSTANCE`, `SINGLE_FILE` and `BINARYEN_ASYNC_COMPILATION=0`. It is linked with a pre.js and a post.js file provided by the DSP2JS framework, which register an AudioWorkletProcessor and have access to the modularized instance of the DSP in the AudioWorkletGlobalScope. The file can directly be loaded by the AudioWorklet interface via `addModule`, which compiles and instantiates the WebAssembly module in the rendering thread. This has to be done only once, optimally at program startup, because new DSP will be instantiated from the same WASM instance using Emscripten's binding mechanism. The processor file also sets the correct external parameters inside `Processor_base` during loading.

The node file is generated from the node description in the .json file provided by the DSP developer. Besides the node name and the AudioWorkletNodeOptions, it defines a set of `members`, which can either be functions or data attributes. The file contains a class derived from AudioWorkletNode which sends events to the processor via its `MessagePort` object corresponding to the members. For the data members it additionally holds the last known values as sent back by the processor and makes them accessible with a getter method.

Finally, the node offers a static method for the registration of the processor at a BaseAudioContext.

In conclusion, the WebAssembly module is first used to set external parameters before we can use it to spawn processing units. They expose DSP2JS and DSP-specific functionality and can be invoked by a `process` function. Based on the `BARE_BUILD` setting, the framework assists the web developer with these tasks to a certain degree.

4. DISCUSSION

This section illustrates some of the decisions made in the development of the framework.

An important decision was the use of Emscripten's *embind* mechanism, as opposed to *cwrap*, *ccall* and *direct calls*. The measured runtime for 10 million function calls to an empty C++ function through the Emscripten-generated JavaScript interface can be seen in Table 1. The results show that ccall is unreasonably slow and is therefore not suitable for the given task. Direct calls and cwrap show the best performance, while embind is slower by a factor of approximately 2. Since embind can express more advanced C++ concepts like classes in JavaScript, and the performance impact is negligible in the presence of actual application logic[1], the more maintainable and flexible embind method is preferred in DSP2JS.

Method	Direct call	ccall	cwrap	embind
Time[ms]	70	1850	80	155

Table 1: Comparison of Emscripten binding methods (10M calls, WASM, -O3)

Another implementation detail is the use of lazy buffers as a countermeasure to manual activation and deactivation of buffers. Since the DSP2JS core should be able to first collect all neccessary metadata and then allocate the memory, the most sensible option was to implement them in a lazy fashion, allocating memory only if the metadata has changed and only when a buffer representation is requested. This takes the responsibility for explicit memory management away from both the JavaScript binding code and the DSP code and moves it to a more appropriate place.

Finally, a major design choice was to compile and instantiate the WebAssembly module on the rendering thread. A positive aspect is that it is easy to implement, manage and deploy. The WebAssembly code is written into the processor's JavaScript file directly via the `SINGLE_FILE` option and does not have to be hosted and transmitted separately. Since the AudioWorkletGlobalScope does not permit asynchronous requests, the `BINARYEN_ASYNC_COMPILATION` option is deactivated. The code in post.js can then always assume that the WebAssembly instance already exists.

The architecture of DSP2JS conforms to this behavior by being able to spawn several processor instances using only one WebAssembly instance. The JavaScript runtime compiles and instantiates the module once when `addModule` is called and before the AudioWorkletProcessor class is registered. This should optimally be done at program startup to avoid pauses and audio glitches at runtime.

A downside of this approach is that – at the time of writing – Emscripten's optimizer only supports ECMAScript 5 syntax, while the definition of an AudioWorkletProcessor is based on ECMAScript 6 class inheritance. It is thus currently not possible to build DSP2JS-based plugins with an optimization level higher than `-O1`. The support for ECMAScript 6 syntax in the Emscripten optimizer is still to be implemented [1].

As an additional fallback, the above issues can be avoided by using the `BARE_BUILD` mode with custom binding code.

5. CONCLUSION

We presented the DSP2JS framework, a flexible and configurable framework that takes responsibility for common tasks from the developers and performs them efficiently and in an opaque manner. It lets DSP developers implement or port DSP code to web platforms and generates artifacts usable by a web developer. The development process can be broken down into three steps:

1. writing the DSP code,
2. describing the JavaScript interface,
3. configuring the build.

In the last step it is defined what kind of bindings are generated. Depending on the `BARE_BUILD` mode, the web developer either gets binding code that can be integrated into the WebAudio API like native nodes, or a bare WebAssembly file for more control and potentially other applications.

We described the core framework in detail, which performs buffer management, conversion and profiling and hides the efficient internals behind the `Processor_base` class. The framework makes extensive use of the base-generic-pattern to yield performant generic implementations hidden behind an abstract and non-generic interface.

Finally, we discussed some of the design decisions concerning the framework.

6. ACKNOWLEDGMENTS

Thanks go to Simon Schwär who was of great help at putting our thoughts to paper, raising the quality bar by a magnitude. We also thank Oliver Scheuregger and Nikita Goddard for the solid groundwork laid for this project. Further thanks go to Jan Plogsties and Dr. Nikolaus Färber for their initiative to represent the paper and project. Finally, we thank the proofreaders of this paper.

7. REFERENCES

[1] [Feature Request] ES6 support for the uglifier. https://github.com/kripken/emscripten/issues/6041. Accessed: 2018-04-12.

[2] Web Audio API. https://webaudio.github.io/web-audio-api/. Accessed: 2018-04-12.

[3] Web Audio Modules | community site. https://www.webaudiomodules.org/. Accessed: 2018-04-12.

[1]For a buffer length of 128 samples and a sample rate of 48 kHz we expect 48000/128 = 375 process function calls, which should result in less than 1000 total function calls per second, considering other properties might usually have a polling rate of 60 Hz. This is negligible compared to 155 ms overhead for 10M function calls.