A CROSS-PLATFORM DEVELOPMENT TOOLCHAIN FOR JIT-COMPILATION IN MULTIMEDIA SOFTWARE

Jean-Michaël Celerier

SCRIME
Université de Bordeaux, France
`jeanmichael.celerier@gmail.com`

ABSTRACT

Given the relative stagnation in single-thread performance of many processors in the recent years, made even worse by the recent security findings such as SPECTRE or L1TF which led to restrictions in existing features and decreased performance for the sake of security, it is necessary to find new ways to improve the run-time performance of dynamic multimedia systems. In this paper, we present the introduction of a just-in-time compiler in the *ossia score* interactive score authoring and playback software. We discuss in particular the creation of a toolchain and software development kit for C++ just-in-time compilation on the three major desktop platforms, the challenges and benefits caused by the use of C++ in terms of standard library requirement, but also the benefits that the system offers in terms of live-coding.

Keywords: interactive scores, just-in-time compilation, toolchains

1. INTRODUCTION

Users of multimedia software demand two features which can be hard to reconcile. On one hand, they ask for more performance, the ability to run more tracks, add more effects, etc. On the other hand, they request more dynamic behavior, and easily extensible systems – in particular, systems which do not require the user to write Makefiles and set-up a compilation toolchain. But such a dynamic behavior generally comes at a cost: for instance, Javascript, Lua or Python are often integrated with media environments, such as *Blender*, *ossia score*, and *Renoise*. These languages can have undesirable properties in low-latency audio environments: they can cause spurious dynamic memory allocations, which prevents real-time guarantees to be ensured.

Ongoing advances in just-in-time compilation can to some extent reconcile these needs. The LLVM project [7] provides simple APIs to integrate compiler and assembler in C++ software, through the MCJIT and OrcJIT sub-libraries.

The benefits of just-in-time compilation have been known for a long time [2] ; of particular interest to us is the ability of just-in-time compilers to adapt to the exact CPU type available in the user's computer. This can lead to great performance improvements: modern compilers are able to generate correctly vectorized code for vector instruction sets, such as SSE, AVX, AVX-2, AVX-512 on x86-based platforms, or Neon on ARM platforms. But in the traditional compilation model, the author of the software has to know beforehand for which instruction set the software shall provide optimized routines, and either write them manually in assembler or with intrinsincs, use compiler-specific extensions such as GCC's function multiversioning [1] or resort to manual run-time dispatch to the correct function according to detection of the user's CPU. This leads to an increase in executable size for all the users of the software, and can be quite time-consuming for the developer. Thus, we propose to leverage JIT compilation for some of the most performance-critical parts of media software so that they can be compiled in the most optimal way for the user's CPU.

The proposed system simply compiles C++ code. This is in contrast with many approaches such as Faust [11] for audio signal processing, PostgreSQL [13] for improvement of the SQL query performance or the language created by Avramoussis et al. for transformation of geometry assets in the VDB format [1]. These systems all provide custom domain-specific languages (DSL) to solve a well-defined task. This has the advantage of freeing oneself from C and C++'s complicated legacy and generally simplify the language semantics, but also means that:

- A large amount of work must be provided by the new language authors.
- The language won't necessarily be subject to new advances in compiler development unless its authors keep working on it: while some optimization phases can occur at later stage if leveraging an existing compiler framework such as LLVM, some optimizations require actual knowledge of the language's semantics and thus cannot be applied generically to any DSL.
- The language may not be able to leverage the existing corpus of libraries available in C and C++.

The system is integrated in the *ossia score* software [6, 4] for media creation. Part of the motivation is to improve run-time performance while live-coding: the software currently features a Javascript engine which can be leveraged to provide new behaviors at run-time. While it is one of the software's user-base's favorite features, it comes at a cost: no real-time safety due to the Javascript engine performing many memory allocations, and huge "context switch" costs between the native code world, and the interpreted Javascript engine world. The objective is to improve the run-time performance, while retaining some of the properties provided by live-coding: for this, Thor Magnusson gives the hard criteria that a live-coding language should not take more than five seconds between code and sound [10].

We will first give a brief overview of the *OSSIA* project, and of the way just-in-time compilation is introduced into the system. Then, we will give some pointers towards the creation of a cross-platform toolchain which allows to support JIT compilation in the three major desktop operating systems, Linux, macOS and Windows. Some performance metrics will be discussed.

2. OSSIA PROJECT

ossia [2] is an open-source software suite composed of a library (*libossia*) and a graphical user interface (*ossia score*) for managing

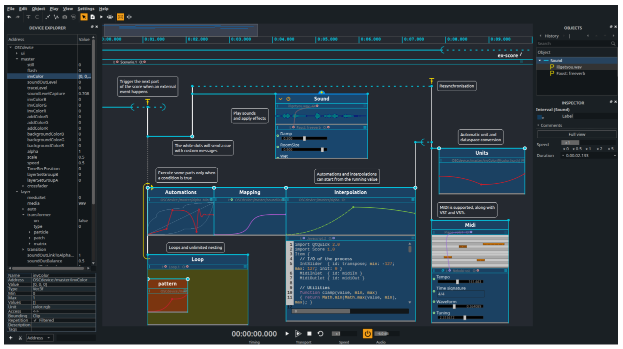
[1] `https://lwn.net/Articles/691932/`

[2] `https://ossia.io/`

communication, mapping and time-scripting between various software in interactive multimedia artworks. This toolset is cross-platform (Windows, macOS, Linux), cross-protocol (OSC, MIDI, ...). The libossia library has been ported to many creative coding tools (Ableton, Max/MSP, PureData, VVVV, Touch Designer, OpenFrameworks, Processing…). It simplifies connecting and controlling various digital production software together. Its main goals are to facilitate the development of time-centric interactive artworks and lower the barrier of entry to interactive media creation and authoring for emerging artists.

The *ossia score* software's execution engine is based on a dataflow architecture described in [3]. The user interface part leverages a modern C++ and Qt-based generic document framework which can be easily reused for other document-centric software. It features an extensible plug-in API, undo-redo with automatic recovery in case of crash, interface injection, serialization, selection handling and multiple document management. It is specifically well-tailored to hierarchical document structures and enforces strong typing practices.

This framework has been used in an unrelated software as a test of its flexibility: a point-and-click game editor (SEGMent, developed with Raphaël Marczak[3]).


File Edit Object Play View Settings Help
DEVICE EXPLORER
OBJECTS
INSPECTOR
Trigger the next part of the score when an external event happens
Play sounds and apply effects
The white dots will send a cue with custom messages
Execute some parts only when a condition is true
Automations and interpolations can start from the running value
Resynchronisation
Automatic unit and dataspace conversion
MIDI is supported, along with VST and VSTi.
Loops and unlimited nesting
Sound
Units
Automations
Mapping
Interpolation
Midi
Loop
pattern
ex-score
00:00:00.000

```
import QtQuick 2.0
import Score 1.0
Item {
  // I/O of the process
  IntSlider  { id: transpose; min: -127;
max: 127; init: 0 }
  MidiInlet  { id: midiIn }
  MidiOutlet { id: midiOut }

  // Utilities
  function clamp(value, min, max)
  { return Math.min(Math.max(value, min),
max); }
```

Figure 1: *ossia score*, the main software leveraging this framework

3. C++ JIT

We chose to extend *ossia score* with a C++ just-in-time compilation mechanism. The main motivations for this were:

- Using C++ allows reusing easily large amounts of existing code ; for instance digital signal processing libraries such as Gamma[12], KFR[4] or FFmpeg[5].
- Due to the amount of software built using C++, compiler optimisations for this language are still an active research topic [8, 9], which guarantees "free" performance improvements in the following years.
- *ossia score* was already integrating Faust, which itself uses LLVM, and thus acted as a gateway drug of sorts.

4. PLUG-IN AND PLUG-IN APIS

ossia score already provides multiple plug-in APIs: a simple API based on defining a unit generator with strong type-safety features relating to the input and output ports of the unit generator, and a low-level API which allows creating plug-ins that can modify every part of the *ossia score* software: menus, panels, etc.

The JIT system leverages the existing plug-in APIs: the same code can seamlessly be integrated either during the build of *ossia score*, or at run-time. We give thereafter a brief overview of these two APIs.

4.1. Safe process API

This API only gives the ability to provide a new unit generator to the system. Inputs, outputs and controls are given as C++ constant expressions, which generates the user-interface code at compile-time and guarantee type-safety. The necessary boilerplate being relatively low (for C++ code), it is viable to use in live-coding contexts. A specific unit generator, for now simply named "C++ Jit process" in the software, allows the user to input code using such API, which will be live-recompiled ; the corresponding node will be instantiated.

Algorithm 1 provides an example of a "gain" node, which has one audio and one floating-point input, one audio output, and applies the gain to the input.

Algorithm 1 : A naive gain implementation in the "safe" plug-in API. The inputs and outputs of the unit generator are declared in the `Metadata` struct. A compile-time mechanism ensures that the prototype of the `run` function conforms to the prototype, and that the types of the arguments are correct. This increases type safety at run-time when compared to the more traditional C-based solutions where the programmer has to manually cast the inputs of the unit generator into the correct type according to knowledge not part of the type system.

```
struct Node
{
  struct Metadata : Control::Meta_base
  {
    static const constexpr auto prettyName = "Gain";
    static const constexpr auto controls
      = std::make_tuple(Control::FloatSlider{"Gain", 0., 2., 1.});
    static const constexpr audio_in audio_ins[]{"in"};
    static const constexpr audio_out audio_outs[]{"out"};
  };

  using control_policy = ossia::safe_nodes::last_tick;
  static void run(
      const ossia::audio_port& p1, float g, ossia::audio_port& p2,
      ossia::token_request, ossia::exec_state_facade)
  {
    const double gain = (double)g;
    const auto chans = p1.samples.size();
    p2.samples.resize(chans);
    for (std::size_t i = 0; i < chans; i++)
    {
      auto& in = p1.samples[i];
      auto& out = p2.samples[i];

      const auto samples = in.size();
      out.resize(samples);

      for (std::size_t j = 0; j < samples; j++)
      {
        out[j] = in[j] * gain;
      }
    }
  }
};
```

[3]`https://scrime.u-bordeaux.fr/Arts-Sciences/Projets/Projets/SEGMent2-Study-and-Education-Game-Maker`
[4]`https://www.kfrlib.com`
[5]`https://www.ffmpeg.org`

4.2. General plug-in API

This API enables its user to introduce new elements in most parts of the software:

- New menus, panels, etc.
- Run-time additions to existing data types of the software.
- File loaders.
- Network and hardware protocols.

At the source code level, it mainly leverages the *Abstract Factory* design pattern. A plug-in can define a new interface, identified by an UUID. An example is given in algorithm 2:

Algorithm 2 : An example of interface definition in *ossia score*. This particular interface allows a plug-in to register the handling of new file types in the “Library” panel.

```
class LibraryInterface : public score::InterfaceBase
{
  SCORE_INTERFACE(LibraryInterface, "9b94d974-9f2d-4986-a62b-
       b69e51a4d305")
public:
  ~LibraryInterface() override;

  virtual QSet<QString> acceptedFiles() const noexcept;
  virtual QSet<QString> acceptedMimeTypes() const noexcept;

  virtual void setup(
      ProcessesItemModel& model
    , const score::GUIApplicationContext& ctx);
  virtual bool onDoubleClick(
      const QString& path
    , const score::DocumentContext& ctx);
  // ...
};
```

Plug-ins can then register implementations for these interfaces, which can be listed and accessed through a global context object.

The majority of the *ossia score* codebase is based on this API, the actual software being itself merely a set of plug-ins implemented on top of the base plug-in framework. The JIT extension discussed here is itself a plug-in [6].

The original plan for *ossia score* was to rely on this plug-in API to allow prebuilt extensions to be downloaded from a common repository. Due to the ongoing development of the software, no ABI (Application Binary Interface) stability guarantees are provided, which means that plug-ins must generally be recompiled against the source code of newer versions. This requires an extensive compilation architecture which could not only rebuild and publish new versions of *ossia score* but also the plug-ins regularly. Common service providers such as *Travis CI* and *Appveyor* do not provide enough capacity for this to be viable for an open-source, volunteer-led project.

Hence, the plan going forward is to distribute the plug-ins not included in the base software under source code form. The JIT system looks for addons on startup in the user library folder: for instance `~/Documents/ossia score library/Addons` and simply compiles all the source files of the addon together. This guarantees that API and ABI breakage do not cause subtle run-time errors since the add-ons are compiled against the exact source code that was used to build the software, the headers being shipped as part of the package: if the API has changed in a breaking manner, the add-on will not be compiled at all and the user warned.

[6]`https://github.com/OSSIA/score-addon-jit`

5. A CROSS-PLATFORM TOOLCHAIN

ossia score being a cross-platform software, it is necessary to ensure the same level of support on the three major operating systems: Windows, macOS and Linux. The endeavor was relatively straightforward on Linux thanks to the availability of the LLVM libraries and compilers in package managers. In particular, the Linux implementation of JIT compilation in *ossia score* is also able to use system libraries instead of the ones provided by the toolchain. The official release of *ossia score* is based on the AppImage mechanism which allows it to work on many distribution: as such, it is also necessary to build a recent toolchain to be able to target older systems, such as CentOS 7 or Ubuntu 12.04.

The complete toolchain, whose build scripts are available at `https://github.com/OSSIA/sdk` provides the following libraries:

LLVM 7.0.1 (8 svn on Windows due to previous versions not working) , Qt 5.12 , FFMPEG 4.1 , PortAudio , JACK headers , SDL2 , OpenSSL , Faust.

5.1. Uniform C++ standard library

The C++ parts of the toolchain are built against the *libc++* standard library implementation on all platforms. This is for two reasons: uniformity, and licensing. Using a single C++ standard library across all platforms guarantees less variance in behavior, which is still fairly common for instance across the various implementations in the implementation of standard algorithms, or complex libraries such as `<regex>`. Especially on Windows, the standard library headers provided as part of Visual Studio are not freely redistributable. This means that this would introduce an unacceptable dependency on a Visual Studio installation into *ossia score*. Hence, we use the system headers provided by the mingw-w64 project, along with the LLVM libc++ standard library. The build process implies a first build of the LLVM project, clang compiler and libc++ standard library, which are then used to boostrap a second set of LLVM libraries. This is needed due to the JIT implementation directly calling into LLVM’s OrcJIT API: if we linked directly against the first set of LLVM libraries, there would be a standard library mismatch which would in the best case fail to link properly, and in the worst case fail at run-time.

The *llvm-mingw* project[7] greatly simplified the creation of the Windows toolchain.

5.2. macOS and rpath handling

macOS is special in that libc++ is the default C++ library implementation. There is no equivalent to MinGW in the Apple world: the only implementation of system headers is the one provided by Apple. Those are not under a free license, to the exception of the C standard library and Mach kernel headers.

In addition, the customized clang / libc++ provided by Apple is slightly out-of-date when compared to other platform’s implementations and suffers from some artificial limitations: using various C++17 standard library types, such as `std::any`, `std::optional` or `std::variant` restricts the deployment to the latest in date version of macOS, 10.14, which is not acceptable for multimedia software users often restricted to older system versions for the sake of compatibility. The macOS version of the toolchain thus provides its own clang / libc++ build which overcomes this problem.

[7]`https://github.com/mstorsjo/llvm-mingw`

A custom-built clang-based toolchain on macOS will by default still link against the system libc++ implementation. The observed behavior is as follows:

- No arguments passed: the compiler hard-codes an absolute path to the system `/usr/lib/libc++.1.dylib`.
- `-L$SDK/lib -lc++ -lc++abi`: the compiler links the software to `@rpath/libc++.1.dylib`.

It is thus necessary to specifiy the rpath to get working binaries during development: `-L$SDK/lib -lc++ -lc++abi -Wl,-rpath,/sdk/lib`.

6. BENCHMARKING

We provide a few performance tests of the system: what advantages and what costs actually bring C++ JIT compilation. Benchmarks are run on two machines, both running Linux (Kernel: 4.20.8-arch1-1-ARCH):

- Machine 1: Intel(R) Core(TM) i7-6900K CPU @ 4.00GHz (Broadwell architecture, desktop).
- Machine 2: Intel(R) Core(TM) i7-8750H CPU @ 4.00GHz (Coffee Lake architecture, laptop).

6.1. Compile times

C++ is notorious for its slow compile times, due to large amounts of header files to include, and the cost of the template instantiation mechanism. More recent C++ standards being oriented towards compile-time computation of most values in a program also leads to an increase in compile times.

On the test machine, a simple node such as the one provided in 1 takes between 1.3 and 1.5 seconds to compile on an average of five runs. A generic test addon providing mock implementations of a few interfaces, comprised of 7 source files, 10 header files, for a total of 428 lines of code which themselves include part of the C++ standard library and Boost, takes between 4.5 and 5 seconds to compile on an average of five runs.

LLVM generates bitcode, which could be cached on-disk, and be used to make following start-ups faster. This optimization is not yet applied and a complete recompile cycle currently occurs for each addon on startup.

The current "interactive" performance characteristics, while much slower than what the Javascript interpreter provides, are thus still viable for some level of live-coding.

6.2. Run times: benchmarking gain adjustment

We discuss here the runtime improvements provided by the system. The following cases cases are tested:

- The *gain* node of algorithm 1 as provided pre-built in the *ossia score* binary, which must work on a variety of systems and thus is not optimized for any kind of vector instruction set outside of the x86-64 SSE2 baseline.
- The same gain node, passed in the system presented in this paper which operates at an `-Ofast -march=native` optimization level and is thus able to take into account the user's actual CPU features.
- A manually optimized version of the gain node, done with hand-written AVX intrinsincs.

We measure every time the time taken by the computation for various common buffer sizes. Figure 2 gives the measurements for the first machine, figure 3 for the second machine.

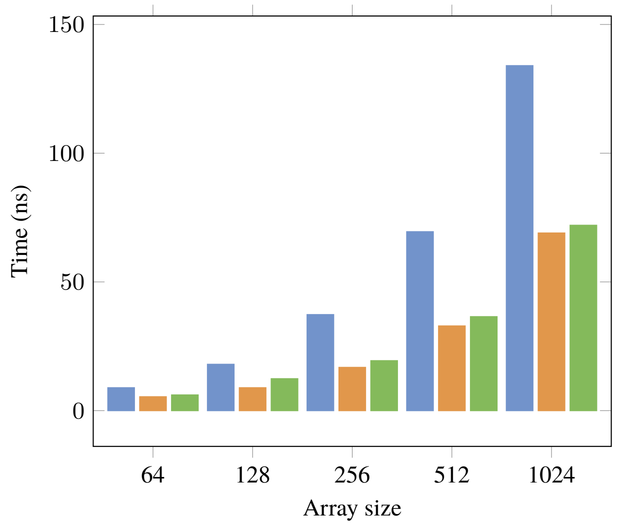


Figure 2: Broadwell CPU: average time in nanoseconds to compute a buffer. In blue: generic code with the default compilation settings. In orange: generic code while built with the JIT system. In green: manually-written AVX implementation.

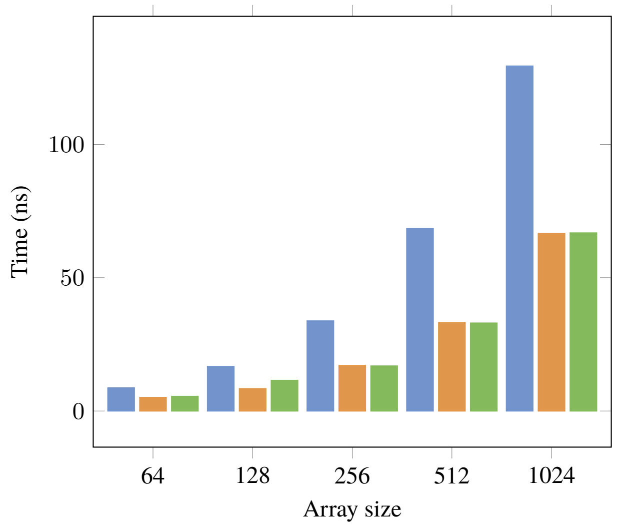


Figure 3: Results for the Coffee Lake CPU, following the same nomenclature than the Broadwell CPU.

Figure 4 presents the improvements between the two CPUs, in order to help the reader see the differences more clearly between figures 2 and 3.

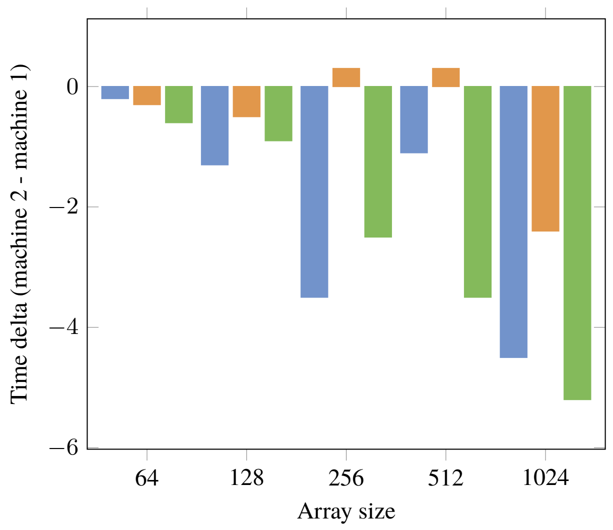


Figure 4: Performance difference between the Coffeelake and the Broadwell CPU: it is interesting to note that the buffer size heavily influences which workloads benefits the most from the CPU improvements.

6.3. Run times: benchmarking FFT

For this benchmark, we compare the run time of a Fast Fourier Transform algorithm implemented in the KFR library mentioned earlier. This library provides hand-optimized versions for many different instructions sets, ranging from SSE2 to AVX2. The results are presented in 1. The test is done on a large array: 16384 double-precision floating-point values.

Machine	Generic	JIT	Time saved
Broadwell	214 µs	144 µs	32.7%
Coffeelake	172 µs	107 µs	37.8%

Table 1: Performance increases yielded by using the proper instruction set.

6.4. Discussion

A few things are made apparent by the previous benchmarks:

- In simple cases, it is pointless to try to optimize better than what the compiler can: the manually-written AVX version is almost never faster than the simple for-loop version when optimized by the compiler.
- The improvement in that case is fairly expected: AVX is able to compute almost twice as many floats than SSE2 in the same time.
- In the more complex, hand-optimized case of the FFT, there are also important performance benefits.
- The C++ compile-times are certainly not negligible for large amounts of code. Potential paths for improvement could be the use of precompiled headers, or upcoming C++ modules.

In addition, we note that the system does not currently add any performance benefits – nor drawbacks – versus compiling the whole codebase at `-Ofast -march=native`. Thus, the system is mainly useful performance-wise in the case where the end-user is not able to rebuild the software himself. While on Linux systems this is generally not a problem (even though users may use old distributions with compilers unable to support recent editions of the C++ language required by *ossia score*), this is tremendously useful for Mac and Windows users where the default toolchain requires mutltiple gigabytes of disk space and takes a long time to install.

7. CONCLUSION

We presented the integration of a C++ just-in-time compilation system based on LLVM in an existing media authoring environment, *ossia score*.

There are multiple further steps that we would like to reach for the system:

- Correct live-reloading of addons. The main problem to handle is that a JIT-compiled addon may instantiate new objects in the system. These objects must be tracked, serialized and reloaded whenever the addon code change: else, due to the ABI of objects potentially changing, this will cause runtime crashes.
- Generation of cross-compiled code. An often requested feature for *ossia score* is to support embedded architectures. While the software already builds and run on such systems, it would be useful to generate a minimal executable for such platforms from a desktop machine, which only contains a given score with implementations optimized for the exact system being targeted.
- In longer time-scales, cross-unit-generator optimizations could be interesting: in particular, how can the system integrate with other languages also based on LLVM such as Faust ? The Mozilla team is currently researching cross-language inlining between C++ and Rust for instance. Combining multiple audio nodes written in different languages, and compile them together in a single dataflow graph may open further optimization opportunities.

Finally, the JIT denomination for the system could in practice be argued: since *ossia score* is itself an interpreter for a visual language, but the execution of the programs of this visual language are done only once every part of the system has been compiled to assembly: for reasons of safety, we prefer not to launch C++ compilations during the execution of a score, since it may seriously hamper the available performance of the system. The JIT process still allows this, but the user must be aware of the risks in doing so if the score already uses most of the machine's cores for instance.

8. ACKNOWLEDGMENTS

We would like to thank Thibaud Keller and the SCRIME & OSSIA teams for their tireless testing of new *ossia score* features, Martin Störsjo for the development of the llvm-mingw toolchain and Stefan Gränitz for simple examples on how to use LLVM's OrcJIT API.

References

[1] Nick Avramoussis et al. “A JIT expression language for fast manipulation of VDB points and volumes”. In: *Proceedings of the 8th Annual Digital Production Symposium (DigiPro’ 18)*. ACM. 2018.

[2] John Aycock. “A brief history of just-in-time”. In: *ACM Computing Surveys (CSUR)* 35.2 (2003), pp. 97–113.

[3] Jean-Michael Celerier. “Authoring interactive media: a logical & temporal approach”. PhD thesis. Bordeaux, 2018.

[4] Jean-Michaël Celerier et al. “OSSIA: Towards a Unified Interface for Scoring Time and Interaction”. In: *Proceedings of the International Conference on Technologies for Music Notation and Representation (TENOR)*. Paris, France, 2015.

[5] Jean-Michaël Celerier, Myriam Desainte-Catherine, and Jean-Michel Couturier. “Graphical Temporal Structured Programming for Interactive Music”. In: *Proceedings of the International Computer Music Conference (ICMC)*. Utrecht, The Netherlands, 2016.

[6] Théo De la Hogue et al. “OSSIA : Open Scenario System for Interactive Applications”. In: *Proceedings of the Journées d’Informatique Musicale (JIM)*. Bourges, France, 2014.

[7] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE Computer Society. 2004, p. 75.

[8] Juneyoung Lee et al. “Reconciling high-level optimizations and low-level code in LLVM”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 125.

[9] Juneyoung Lee et al. “Taming undefined behavior in LLVM”. In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 633–647.

[10] Thor Magnusson. “Algorithms As Scores: Coding Live Music”. In: *Leonardo Music Journal* 21 (2011), pp. 19–23.

[11] Yann Orlarey, Dominique Fober, and Stéphane Letz. “Faust: an efficient functional approach to DSP programming”. In: *New Computational Paradigms for Computer Music*. Paris, France, 2007.

[12] Lance Putnam. “Gamma: A C++ sound synthesis library further abstracting the unit generator”. In: *Proceedings of the Joint International Computer Music Conference (ICMC) / Sound and Music Computing Conference (SMC)*. Athens, Greece, 2014.

[13] Evgeniy Yur’evich Sharygin et al. “Dynamic compilation of expressions in SQL queries for PostgreSQL”. In: *Proceedings of the Institute for System Programming of the Russian Academy of Sciences* 28.4 (2016), pp. 217–240.