A Just-in-Time Compiler for Csound Opcodes

Victor Lazzarini[1]*

Department of Music
Maynooth University, Ireland,
emailvictor.lazzarini@mu.ie

Abstract. This paper introduces the development of a just-in-time compiler that is ready to be deployed as a library of plugin opcodes for Csound. It describes a module compiler, which can take C or C++ code, compile it and make it available inside a running instance of Csound. The set of opcodes also provide means of calling functions from within Csound and to instantiate C++ classes and run these as opcodes within the system. Tests have indicated that JIT-compiled C++ code can match the efficiency of, or even outperform, existing Csound opcodes. The paper is completed by a discussion of this project as part of a wider, more ambitious, plan to provide a just-in-time compiler for user-defined opcodes.

Keywords: Just-in-time compilers, extending Csound, LLVM

1 Introduction

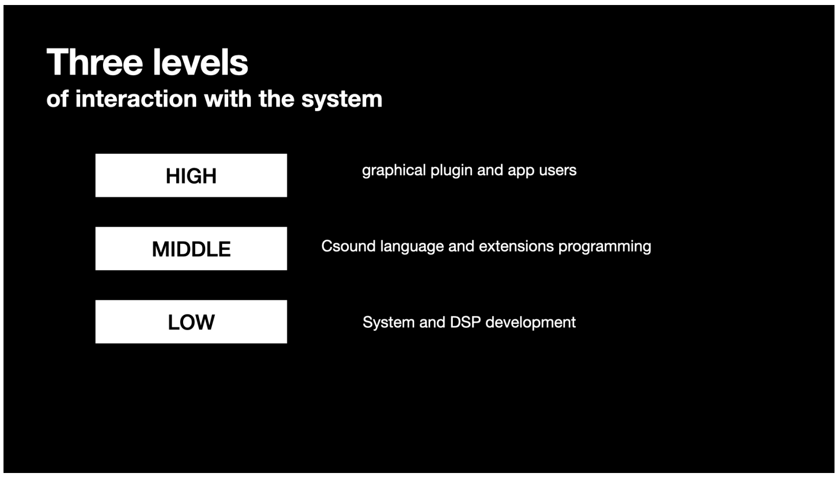
Csound [1] is a mature Sound and Music computing system, developed over more than 30 years. It is supported by world-wide user community, running in almost every commonly available platform, from mobile and embedded devices to supercomputers. It is capable of hard-realtime as well as offline operation, and it is deployed as a library with a comprehensive application programming interface (API). As a programming language, it is continuously evolving, with many new features added in recent years, and it supports extensions through various means. In the latest decade, it has also become extensively used in commercial software applications, beyond its usual role in music research.

In particular, it is relevant to point out that Csound supports three distinct levels of user interaction (Fig. 1) [1, p.15]. While the top level may not involve any form of programming, Csound users are mostly used to working at the middle level, through developing instruments and user-defined opcodes (UDOs). At the lowest level, it is possible to use deployment languages to embed or extend the Csound system itself.

As noted earlier, Csound is highly extendable through

- Its own language: UDOs.

* The author would like to thank Michael Gogins for his work with LLVM JIT compilation, which has provided the initial basis for this study.



Three levels
of interaction with the system

HIGH	graphical plugin and app users
MIDDLE	Csound language and extensions programming
LOW	System and DSP development

Fig. 1. Three Levels of User Interaction.

- External languages, e.g. python and Faust
- C/C++ dynamic libraries (plugin opcodes)

This paper describes a new addition to these forms of system extension, which leverages the low-level virtual machine (LLVM) compiler technology [2] to provide a just-in-time (JIT) compilation mechanism to Csound. This can be considered as a stand-alone project ready for deployment or as the first stage of a more ambitious plan to deliver a JIT compiler for Csound UDOs.

2 Just-in-time Compilers

Compilers are software tools that translate code written in a given language into another representation, which may be, for instance,

- a DSP graph (as in the Csound compiler)
- an intermediate representation (IR)
- assembler code
- binary (executable) code

depending on the compiler, the language, the target system, etc.

Normally, a compiler is invoked as a self-standing tool to build a piece of software, usually in the form of binary executable code. Most software we use are created in this way. This is done in advance of the target software being run by a user.

A JIT compiler, on the other hand, produces a binary executable as the user runs the software, slightly before a particular piece of code needs to be employed. It is given code in some form and produces an executable that can immediately be used.

Similarly to a JIT compiler, a language interpreter can also take code and run it immediately. However, such a software does not translate the code into a binary executable form, but instead it runs pre-compiled commands defined by the language in question.

There is a significant difference in performance: JIT-compiled code should run very nearly or perhaps exactly as a natively compiled (ahead of time) binary. It has the advantage of an interpreted code in that there is no build stage, the code can be run directly as given.

The LLVM infrastructure provides support for the development of a JIT compiler, which can take an input consisting of IR code and provide a binary executable corresponding to it. In order to produce the IR, we can leverage the Clang compiler tools, which can take C or C++ code as input. The combination of these two toolchains allows us to put together an extension to Csound that implements a JIT compiler.

3 A JIT Compiler for Csound

The most common way to extend Csound by the means of a binary executable is to provide plugin opcodes in a dynamic load library (Fig. 2). Plugins are loaded into the Csound engine when this is initialised by a host frontend, and are then available for use in Csound code.

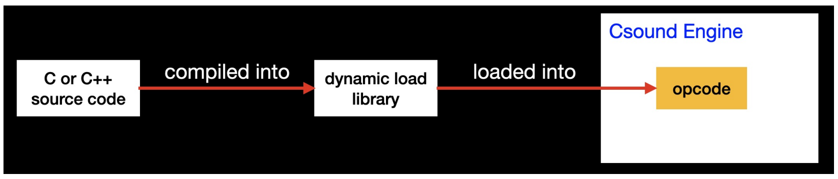


Fig. 2. Plugin Opcodes.

With a JIT compiler, instead, we can send C or C++ code to be compiled as required and it is immediately available for use (Fig. 3). The JIT compiler is itself given as a dynamic library to Csound and is accessed via a set of opcodes. The first one of these is the *module compiler*.

3.1 Module Compiler

There are two stages performed by the module compiler. The first one translates C or C++ to LLVM IR (bitcode), using clang, and the second, the actual JIT compiler, translates bitcode into an executable object inside Csound (Fig. 4).

A trivial example, *amp*, demonstrates its use. This is based on a C-language opcode that changes the gain of an input signal. For this, we need three components:

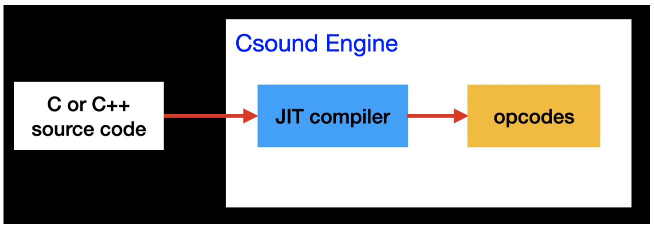


Fig. 3. JIT Compiler.

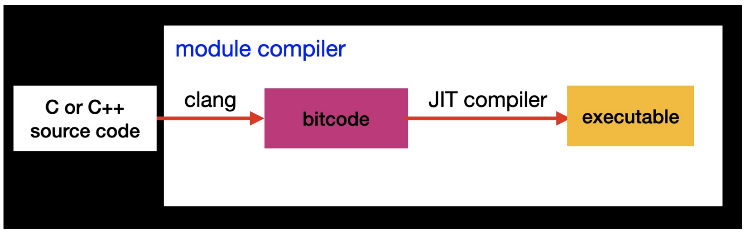


Fig. 4. Module Compiler.

1. The C source code for the opcode (passed as a text string)
2. The module compiler invocation, taking the source code.
3. The actual Csound code using the new opcode.

The C source for the `amp` opcode is given as a Csound string constant,

```
SCode = {{
  #include <csdl.h>
  typedef struct dataspace {
     OPDS h;
     MYFLT *out, *in, *gain;
  } DATASPACE;

  static int perf(CSOUND *csound, DATASPACE *p) {
    MYFLT *out = p->out,*in = p->in, g = *p->gain;
    uint32_t n, nsmps = CS_KSMPS;
    for(n=0; n < nsmps; n++)
       out[n] = in[n]*g;
    return OK;
  }

  int module_init(CSOUND *csound) {
    csound->AppendOpcode(csound,"amp",sizeof(DATASPACE),0,2,
        "a","ak",(SUBR) NULL,(SUBR) perf);
    return OK;
  }
 }}
```

The C module compiler opcode takes the C code and an entry point function name as strings,

```
ires,ihandle c_module_compile SCode, "module_init"
```

Its first output contains the function return value. The second is a handle to an executable object, which can be used to invoke code from the C module. The module compiler is used with C or C++ code to provide binaries for execution.

After compilation, module_init() is run and the opcode amp is added to the engine. We can now use it in an instrument,

```
SCscode = {{
    instr 1
     out(amp(oscili(0dbfs,A4)),0.5)
    endin
    }}
ires = compilestr(SCscode)
```

since Csound is already running, we need to send the new code to be compiled by the engine before it can be run.

We can also call any C or C++ module functions with the signatures,

```
int func(CSOUND *csound, OPDS h, MYFLT *out[], MYFLT *in[]);
extern "C" int
 func(CSOUND *csound, const OPDS &h, MYFLT *out[], MYFLT *in[]);
```

at init aand/or perf-time using

```
// C (init)
ir1[,ir2, ...] c_module_fcall ihandle,Sfunc[,...]
// C++ (init)
ir1[,ir2, ...] cxx_module_fcall ihandle,Sfunc[,...]
// C  (perf)
xr1[,xr2, ...] c_module_fcallk ihandle,Sfunc[,...]
// C++ (perf)
xr1[,xr2, ...] cxx_module_fcallk ihandle,Sfunc[,...]
```

3.2 Compilation errors

As with any C/C++ code, we are always subject to coding errors and typos that may result in a module that cannot be compiled. In this case, there is no viable bitecode and an execution instance is not created. In that case, the opcode compiler returns a non-zero code as its first argument, which should be checked before any attempts to call the code. Additionally, a compilation error would yield an invalid handle. Therefore any attempt to run the code would result in either on an init error (in the case of function calls), or in the case of newly-added opcodes, a Csound parser error as the opcode has not been successfully added to the code.

Performance-time errors and segmentation faults, unfortunately, cannot be satisfactorily protected against and may cause the Csound engine to crash. This is of course the case for any C/C++ that is added to Csound in one way or another, so there is nothing particularly special or surprising here.

3.3 C++ opcode objects

In addition to function calls, it is possible to construct and run C++ objects at i, k, or a rates (or a combination of these). For these, the code needs to provide a class implementing the opcode processing, and an entry function to instantiate objects of this class

```
struct OpcodeObj : JITPlugin {
  OpcodeObj(OPDS h) : JITPlugin(h) {};  // constructor
  int init()  { return OK; }  // called at init-time
  int perf() { return OK; }   // called at perf time
};

auto entry(OPDS h) {
 return new OpcodeObj(h);
}
```

Once the object is defined in the C++ code it can be run by passing the entry point name and the JIT handle to the appropriate cxx_opcode_[i,k,ik,ia] opcode,

```
 // i-time only
ires[,...] cxx_opcode_i ihandle,Sentry[,...]
// perf-time only ksig input/output
ksig[,...] cxx_opcode_k ihandle,Sentry[,...]
// perf-time only any type
xsig[,...] cxx_opcode_a ihandle,Sentry[,...]
// i-time, perf-time i/k
k/ivar[,...] cxx_opcode_ik ihandle,Sentry[,...]
// i-time, perf-time any
xvar[,...] cxx_opcode_ia ihandle,Sentry[,...]
```

and since these opcodes are already in place in the system as part of the JIT compiler plugin, there is no need to compile new Csound code to use them as in the other example.

An opcode object version of the *amp* example looks like this

```
Scode = {{
 struct Amp : JITPlugin {
    Amp(OPDS h) : JITPlugin(h) {};
    int perf()  {
        for(int n = offset; n < nsmps; n++)
             outargs(0)[n] = inargs(0)[n]*inargs[1];
        return OK;
    }
 };
auto amp(OPDS h) { return new Amp(h)};
}}

gires,gihandle cxx_module_compile SCode
```

A running instance is obtained from the entry point factory amp. We pass this to cxx_opcode_a, along with the audio and control signals

```
instr 1
 out(cxx_opcode_a(gihandle,"amp",oscili(0dbfs,A4),0.5))
endin
```

Any number of distinct instances of *amp* can be run, just like any opcode in the system.

3.4 Performance

But is this any good?

While the main aim of this project has not been to examine performance in detail, it is worth looking at how the JIT code performs. A basic comparison between an internal and a JIT-compiled should provide a general idea. For this we can select one of the C++ class examples, for instance, the `DelayLine`, and then compare it to an equivalent existing opcode, which in this case is `comb`. We have also included in the test a UDO implementation of the same process. This should be useful to demonstrate how useful the kinds of gains we should expect once we are able to roll out the technology directly through Csound programming, as outlined later in this paper.

The code for the `DelayLine` example is as follows:

```
SCode = {{
 #include "../jitplugin.h"
 #include <vector>

 struct DelayLine : JITPlugin {
   std::vector<MYFLT> delay;
   std::vector<MYFLT>::iterator iter;
   DelayLine(OPDS h) : JITPlugin(h), delay(0) { };

   int init() {
    if(inargs[1] > 10000)
      return csound->init_error("delay time too long\\n");
    delay.resize(csound->sr() * inargs[2]);
    iter = delay.begin();
    return OK;
  }

  int perf() {
    csnd::AudioSig in(this, inargs(0));
    csnd::AudioSig out(this, outargs(0));

    std::transform(in.begin(), in.end(), out.begin(),
                         [this](MYFLT s) {
      MYFLT o = *iter;
      MYFLT g = inargs[1];
      *iter = s + g*o;
      if (++iter == delay.end())
        iter = delay.begin();
      return o;
    });
    return OK;
  }
};

 extern "C" {
   auto delayline(OPDS h) {
    return new DelayLine(h);
   }
}
}}

gires,gihandle cxx_module_compile SCode

instr 1
 prints "\n***\nRunning JIT DelayLine C++ opcode\n***\n\n"
 idt = 0.5
 kg = 0.5
 a1 diskin "fox.wav",1,0,1
 a2 cxx_opcode_ia gihandle,"delayline",a1,kg,idt
    out a2*0.5
endin
```

The equivalent code using Csound code solely (opcode and UDO) is

```
instr 2
 prints "\n***\nRunning comb opcode\n***\n\n"
 idt = 0.5
 kg = 0.5
 a1 diskin "fox.wav",1,0,1
 a2 comb a1,-3*idt/log10(kg),0.5
    out a2*0.5
endin

opcode DelayLine,a,aki
 ain, kg, idt xin
 ids = idt*sr
 adel[] init ids
 kp init 0
 aout init 0

 kn = 0
 while kn < ksmps do
  aout[kn] = adel[kp]
  adel[kp] = ain[kn] + adel[kp]*kg
  kp = kp != ids - 1 ? kp + 1 : 0
  kn += 1
 od

 xout aout
endop

instr 3
 prints "\n***\nRunning DelayLine UDO\n***\n\n"
 idt = 0.5
 kg = 0.5
 a1 diskin "fox.wav",1,0,1
 a2 DelayLine a1,kg,idt
    out a2*0.5
endin
```

We have run these instruments separately for 100 seconds on a 2.9GHz Intel Core i9 processor under MacOS 12.3.1, and averaged the CPU times for 10 runs. Tests have shown that the CPU performance of the `comb` opcode and the JIT-compiled `DelayLine` are very close, while the UDO is significantly slower. In this particular case, it is even possible to say that the JIT-compiled C++ class has the edge over the existing Csound opcode. Table 1 summarises the results.

Table 1. CPU timings reported for each one of the three comb filter instruments running for 100 seconds.

instrument	CPU time in seconds
instr 1 `DelayLine`	0.776
instr 2 `comb`	0.783
instr 3 `UDO`	1.86

4 The Direction of Travel

This project set out to investigate the feasibility of providing a JIT compiler to allow Csound to be extended on-the-fly. It has demonstrated not only that this is possible, but that it can actually be a means of providing fairly efficient implementations of new DSP algorithms without the need to create external dynamic libraries. This is ready to deployed as is, but it also completes the first stage of a wider, more ambitious project, which is to allow users with no knowledge of C or C++ to extend Csound natively.

The most common way for Csound users to extend the language is to create new opcodes in the form of UDOs. This has a low entry requirement, as it uses the Csound language itself as a means of programming. However, UDOs are as efficient as any instrument code, which is slower than natively-compiled binaries (as the Csound compiler is more akin to a language interpreter). If we can use a JIT compiler for UDOs, then we should make these much more computationally efficient.

Most of the components for this are already in place. The Csound parser can produce an abstract syntax tree (AST) from a UDO. The module compiler can take a C++ class and produce an instantiable binary (Fig 5). The missing piece is an AST to C++ compiler, which can translate a UDO into a C++ class.

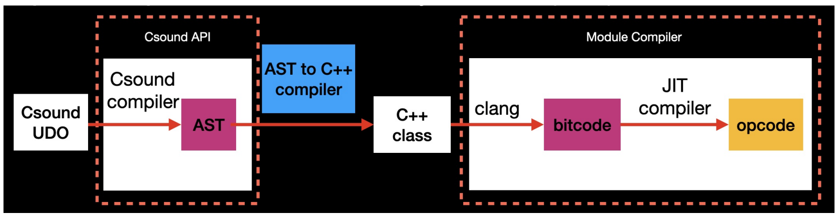


Fig. 5. An UDO Compiler.

5 Conclusions

Csound is a very powerful sound and music computing system, which runs everywhere.It is used in research and commercial applications by a world-wide user community. It is highly extensible through various languages.Its language is easy to program in, posing only light demands on the user, with a gentle learning curve.

The new LLVM/Clang module compiler [3] allows C/C++ code to be used directly in Csound, embedded as strings in its orchestra. With it, it is possible to add new natively-compiled opcodes to the system on-the-fly, without having to compile and load external dynamic libraries. It is also possible to run C/C++ functions defined in the module, and instantiate and run C++ opcode classes very efficiently. It provides an important stepping stone for the development of a UDO compiler that will take Csound code and produce an executable binary.

References

1. Lazzarini, V. et al.: Csound: A Sound and Music Computing System. Springer (2016)
2. LLVM Project, `http://llvm.org`
3. LLVM Project, `https://github.com/vlazzarini/opcode_compiler`