AUGMENTING COMPUTER MUSIC WITH JUST-IN-TIME COMPILATION

Wesley Smith, Graham Wakefield

University of California Santa Barbara
Media Arts and Technology
whsmith|wakefield@mat.ucsb.edu

ABSTRACT

We discuss the potential of just-in-time compilation for computer music software to evade compromises of flexibility and efficiency due to the discrepancies between the respective natures of composition and computation and also to augment exploratory and generative capacity. We present a range of examples and approaches using LLVM compiler infrastructure within the LuaAV composition environment and measure its performance against static compilation.

1. INTRODUCTION

Striking a balance between dynamic flexibility and desirable efficiency is one of the fundamental challenges of creative software design. Maximally efficient code follows ordered patterns determined before execution begins, granting the compiler more freedom for optimization. Software design therefore typically demands a compromise in order to break naturally open-ended domains into efficient precompiled modular chunks that can be freely reassembled at run-time. The price to pay follows from the immutability of the precompiled chunks and the restriction to a predefined granularity.

On the other hand, composition is a complex indeterminate activity resistant to a priori characterization, a domain often explored and defined gradually by provisional approaches and in reflective manners. Systems cannot anticipate all possible situations flawlessly; ideally they should evolve in the hands of users [5], [7].

The need to increase flexibility while maintaining efficiency is exacerbated in situations where processing must not terminate, such as autonomous and interactive generative systems, and live-coding performance [3], [4]. Extending the capacity of a program to be redefined at run-time embodies a shift away from the notion of computer as a bounded tool for a bounded task toward the notion of an always-on workspace or environment. Run-time augmentation is thus relevant to composition in general, since radically decreasing the latency and maximizing the overlap between action and perception may increase interactive fluidity and reduce conceptual load.

In this paper we discuss the potential of just-in-time (JIT) compilation to relax tensions of run-time flexibility and efficiency. JIT compilation allows the compilation of arbitrary new structures and functions into efficient machine instructions at run-time, effectively allowing the efficient extension and redefinition of an active program as part of its execution. JIT compilation can lead to much greater execution efficiency than interpreted modular chunks by avoiding the evaluation of code step by step, reducing address lookups, optimizing deterministic or probable paths, and translating operations to platform-specific routines[1].

Besides the benefits of efficiency and flexibility, expressing abstract algorithms for JIT compilation may also:

- adapt to run-time user/environment demands,
- grant portability between systems [6],
- offer protection against dependency obsolescence.

2. RELATED WORK

The Max family [14] offers a clear example of the compromise between flexibility and efficiency in an always-on environment. A modular architecture supports the modular assembly of pre-compiled black-box objects into run-time interpreted *patches* by means of a graphical data-flow idiom.

SuperCollider 3 [10] also grants the user flexibility to define kernels of state and synthesis function (*SynthDefs*) from low-level primitives at run-time. Kernels are created using the interpreted *SCLang* language and stored in a byte-code representation. When instantiated on the *SCServer* virtual machine, these byte-codes are interpreted to call the appropriate pre-compiled, optimized library functions and generate audio data.

Perhaps a closer antecedent to our work is the Kyma system as described in 1988 [11]. The software featured a capacity to define synthesis kernels (*SoundObjects*) programmatically using the interpreted Smalltalk-80 language, as highly modular tree-like hierarchical data structures of generators and transforms. Parsing these data structures at run-time generates assembly language code (synthesis function and data registers) for instantiation on a dedicated DSP microprocessor.

[1] Note that efficiency is not only to be judged by time: run-time code generation could adapt algorithms to match available or reduce needed memory.

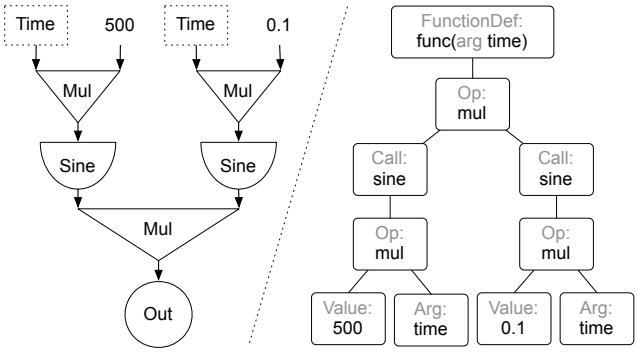

Figure 1. An oscillator with amplitude modulation: synthesis graph (left), and equivalent abstract syntax tree (right.)

3. INTEGRATING LUAAV & LLVM

Before detailing our explorations into the potential of JIT compilation, we first describe the LuaAV [12] environment and the LLVM (Low Level Virtual Machine [9]) compiler infrastructure upon which our investigations were carried out.

LuaAV is a platform for computational composition and audiovisual research with which users can define custom application environments and execute and manage user-authored scripts. These scripts make use of the grammar and vocabulary of the Lua language [8] along with extensions we have provided for timing, audio drivers, windowing and OpenGL, MIDI, OSC etc.

A persistent issue throughout the development of LuaAV has been the reconciliation of the static and the dynamic. Basing our work upon an interpreted dynamic language already granted extensive run-time re-configurability, such as dynamic data structures, garbage collection, code evaluation, first-class functions, etc. JIT compilation promises the extension of this kind of dynamic flexibility to what would otherwise demand ahead of time compilation, such as efficient synthesis routines.

LLVM offers a flexible, target independent set of C++ libraries encompassing compiler workflow, which can be embedded within an executing application for JIT compilation. The workflow follows the standard design of using a *front-end* to convert code expressed in a specific language or format into in intermediate representation (IR), to apply transformations (such as optimizations) to this IR, and then convert this IR to executable instructions for a particular *back-end* target platform.[2] The significant contribution of LLVM is the agnostic portable IR language as *linguistic switchbox* connecting all stages of compilation,[3] and mirroring this IR in the library application programming interface (API.)

To explore the potential of JIT compilation we have embedded the LLVM libraries within LuaAV. We created template bindings to make the majority of the LLVM API directly available to scripts in the Lua language, with which the following investigations were implemented.

Listing 3.1 Code generation of the graph in Figure 1.

a) Concise constructor form (Lua):

```
local dag = sin(time() * 500) * sin(time() * 0.1)
```

b) Linearized list of static single assignments (pseudo-assembly):

```
@1 <- 500
@2 <- time()
@3 <- mul(@1, @2)
@4 <- sine(@3)
@5 <- 0.1
@6 <- time()
@7 <- mul(@5, @6)
@8 <- sine(@7)
@9 <- mul(@4, @8)
```

c) Generated instructions (LLVM IR):

```
define double @func(double %time) {
entry:
  %multmp = mul double %time, 5.000000e+02
  %calltmp = call double @sine(double %multmp)
  %multmp1 = mul double %time, 1.000000e-01
  %calltmp2 = call double @sine(double %multmp1)
  %multmp3 = mul double %calltmp, %calltmp2
  ret double %multmp3
}
```

3.1. Expression trees

Expression trees offer a simple yet general form to describe an algorithm, consisting of a hierarchical directed acyclic graph (DAG) of operations by connecting outputs to inputs. They are applicable to computer music in the form of stateless synthesis generators and are an appealing starting point for our work since parsing an expression results in an abstract syntax tree (AST) of identical structure (Figure 1.)

In particular, we treat the expression tree as a function of time, mapping temporal index (a read-only global value) to sample value output, resulting in a callable function equivalent to the C function signature *double (*expr)(const double t)*. We make use of Lua's flexibility to provide succinct constructors of expression tree data structures (Listing 3.1a).

To convert the AST into LLVM IR, we linearize the tree by depth-first traversal to produce a series of single assignment instructions, ensuring that the arguments of each operator or call have already been assigned (Listing 3.1b.) Since LLVM IR is static single assignment (SSA), it is straightforward to define the body of JIT-able function using our binding (Listing 3.1c.) This native function is then made available for use within the audio process to synthesize sound.

Storing the synthesis expression as a hierarchical data-

[2] Back-end JIT targets include x86 and PPC (32/64-bit) and ARM, front-end parsers exist for C and Objective-C, and the libraries are available for OSX, Linux and Windows under a BSD-style license.

[3] The idea is not new: a common intermediate language (UNCOL [13]) was proposed fifty years ago, but not widely implemented. LLVM is a mature project with support from many groups including Apple and Adobe[1].

Listing 3.2 Definition and instantiation of a new kernel *phasor* (cheap cosine by vector rotation.)

```
local phasor = Synth {
  name = "phasor",
  default = {
    a = 1, b = 0,
    rate = 0.01
  },
  process = function()
    a:store(a - b * rate)
    b:store(b + a * rate)
    out:store(a)
  end,
}

-- instantiate this synth:
phasor { rate = 0.1 }
```

structure opens up a variety of interesting possibilities, such as the genetic programming of graphs as life-forms within an evolutionary landscape [2].

3.2. Stateful synthesis graphs

Expression graphs however are limited because they have no internal state. Extending our model to support stateful objects such as filters and variable oscillators calls for the run-time generation of data-structures to maintain state across function calls, and associated routines to allocate and free memory appropriately. In order to generate optimal algorithms it is also vital to distinguish immutable state, state updated at control rates, and state updated per sample.

The parsing and code generation is encapsulated in a higher-level function *Synth* which provides a template to define a synthesis kernel (akin to a Kyma SoundObject or Supercollider SynthDef) and register it with the executing application. Encapsulating code generation and kernel registration within a higher level template not only eases the conceptual load on the end-user, it can also provide a good deal of the memory and type safety essential to avoid crashes in run-time code generation!

A *Synth* kernel is defined by a name, an optional set of default values, and a function to describe the synthesis algorithm in a similar fashion to the previous example. The Synth function returns a Lua-bound native function to allocate and accurately register an instance of the kernel with the LuaAV real-time audio synthesis scheduler. Arguments to this function can specify distinct parameter values for each instance (see Program 3.2.)

In order to support stateful kernels, the parser from the previous example was extended to recognize *variable* nodes as potential function arguments. Variables have associated metadata for identification (name), storage (local to parsed kernel or global to application), rate (constant, control or signal) and an initial value drawn from the defaults table if defined. Referencing variables inserts appropriate *load* and *store* instructions to the AST. Prior to code generation,

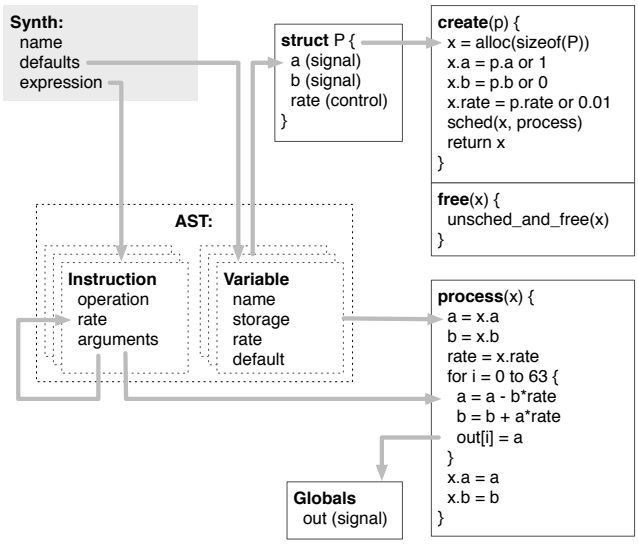

Figure 2. Schematic overview of the code generation and pseudo-code of the results of Program 3.2.

the parser traverses the AST and promotes the rate of each instruction the highest rate of all its argument nodes (a form of type checking/coercion.)

After parsing the code generator traverses the AST and collates all local storage variables to determine the necessary state for the kernel. The list of local variables is used to define a data structure (akin to a C struct) and generate a function to allocate memory and set default values and to free memory accordingly.

The code generator is then ready to produce the *process* function from the AST; this function takes a pointer to the data structure as its argument. The process function begins with instructions to load references to this data structure's members, followed by all of the non-signal rate instructions. This is followed by an a inner loop block for signal rate processing into which all the signal-rate instructions are inserted. Global signal pointers in inner loop store instructions are offset by the loop iterator.

The LLVM optimization passes promote most of the load and store instructions to register variables where possible, trim unused code and restructure expressions. In general, the performance of JIT compiled functions and equivalent precompiled C++ code is similar.

Several performance measurements are listed in Table 3.2. The Phasor is implemented as in Program 3.2, while the Biquad implements a minimal two-pole, two-zero filter. The

	gcc	gcc -03	JIT	JIT-opt	Synth()
Phasor	1.087	0.717	0.937	0.863	2774.5
Biquad	4.020	2.355	3.670	3.275	3443.7

Table 1: Median time (microseconds) over 1 million tests.

JIT columns measure execution of *process()* calls (block size = 64 samples) as JIT-compiled by LLVM, while the gcc columns measure pre-compiled C++ versions of the same algorithms. The final column measures the entire *Synth()* call. Measured with gcc version 4.0.1, LLVM revision 71029, on a 2 GHz Intel Core Duo running Mac OS 10.5.6.

3.3. C code generation

While these examples are simple for clarity's sake, the principles can be extended to far more complex structured objects and methods, and not restricted to the domain of generating time-domain audio samples. Nevertheless, at a certain point it may be preferable to specify algorithms with alternative syntax.

Within the Lua-LLVM binding we also have a proof of concept using the Clang front-end for compilation of matrix processing kernels into LLVM IR. The procesing kernels describe an element of computation to be applied across a domain similar to GLSL fragment shaders or Adobe Pixel Bender kernels.

Kernels are described in a C-like syntax together with meta-data containing information such as parameter inputs and what the valid matrix formats for the particular kernel are. A processing kernel can have an arbitrary number of parameters along with N input matrices and N output matrices. The current prototype operates on matrices containing up to 3 dimensions of 1 to 4 channels of data that is either 8-bit or 32-bit per channel. The implementation is similar to the expression tree implementation described in section 3.1, except that instead of emitting IR it emits valid C code for subsequent evaluation and code-generation by Clang.

4. DISCUSSION

We have described a range of approaches to the run-time JIT compilation of synthesis routines and processing functions. These approaches free the LuaAV composition environment from the hard boundary between static and dynamic computational elements that a bytecode representation entails.

By embedding JIT compilation into the run-time environment, we have given it the capacity to *augment* itself: admitting new primitives and connections and thus integrating new qualities into the unfolding computational process.

In general terms, augmentable environments are heterogeneous in nature and permit structural feedback that otherwise would not be possible. There is clear potential for computation to not merely serve the development of new compositional forms, but to also become a meta-material worthy of exploration in and of itself. We have only just begun to explore the possibilities of this class of computational space, but from our current experiences we feel it to be a vast and fruitful terrain on which to set out.

5. REFERENCES

[1] "2008 llvm developers' meeting," http://llvm.org/devmtg/2008-08/.

[2] "fastbreeder," http://www.pawfal.org/Software/fastbreeder/.

[3] N. Collins, "The analysis of generative music programs," *Organised Sound*, vol. 13, no. 3, pp. 237–248, 2008.

[4] N. Collins, A. Mclean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Organized Sound*, vol. 8, no. 03, pp. 321–330, 2003.

[5] G. Fischer, E. Giaccardi, Y. Ye, A. G. Sutcliffe, and N. Mehandjiev, "Meta-design: a manifesto for end-user development," *Communications of the ACM*, vol. 47, no. 9, pp. 33–37, 2004.

[6] M. S. O. Franz, "Code–generation on–the–fly: A key to portable software," 1994.

[7] E. Giaccardi and G. Fischer, "Creativity and evolution: a metadesign perspective," *Digital Creativity*, vol. 19, no. 1, pp. 19–32, 2008.

[8] R. Ierusalimschy, L. de Figueiredo, and W. Celes, "The evolution of lua," *Proceedings of the third ACM SIGPLAN conference on History of Programming Languages*, 2007.

[9] C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," in *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, 2004.

[10] J. McCartney, "Rethinking the computer music language: Supercollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[11] C. A. Scaletti and R. E. Johnson, "An interactive environment for object-oriented music composition and sound synthesis," *SIGPLAN Not.*, vol. 23, no. 11, pp. 222–233, 1988.

[12] W. Smith and G. Wakefield, "Computational audiovisual composition using lua," *Communications in Computer and Information Science*, vol. 7, pp. 213–228, 2008.

[13] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel, "The problem of programming communication with changing machines: a proposed solution," *Communications of the ACM*, vol. 1, no. 8, pp. 12–18, 1958.

[14] D. Zicarelli, "How i learned to love a program that does nothing," *Computer Music Journal*, vol. 26, no. 4, pp. 44–51, 2002.