

# 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 an approach to computer music software design emphasizing run-time augmentation through just-in-time compilation. With examples using the LuaAV composition environment and the LLVM compiler infrastructure to generate native user-defined functions and structures at run-time. Significant benefits for computational composition are outlined and discussed.

## 1. INTRODUCTION

Striking a balance between dynamic flexibility and desirable efficiency is one of the fundamental challenges of software design. Maximally efficient code is that which follows patterns determined before the application is launched. Design therefore typically demands *knowing in advance* what flexibility may be desired in order to break naturally dynamic domains into efficient pre-compiled modular chunks that can be reassembled at run-time. Such pre-compiled chunks remain immutable and restricted to a predefined granularity.

Composition however is acknowledged as a complex and ill-defined problem space resistant to complete characterization. It is a domain which we explore and define gradually and provisionally, often with a reflective manner, and towards which our approach may change radically over time. Systems to support creative exploration cannot anticipate all possible situations or handle all situations flawlessly; ideally they should evolve in the hands of users [6], [8].

Extending the capacity of a program to be modifiable at runtime (*interactive programming*) assists fluidity by radically reducing the latency and maximizing the overlap between action and perception. It is a shift away from the notion of computer as a bounded tool for a bounded task, toward a notion of an active always-on workspace or environment. Such open-endedness can expand the space of creative discovery for interactive computer music and arts in general, and most particularly for generative systems and live-coding [4], [5], [14].

In this paper we discuss the potential of just-in-time (JIT) compilation to open up this space. JIT compilation allows the compilation of arbitrary new structures and functions into efficient machine instructions at runtime, 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<sup>1</sup>.

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

- adapt to runtime user/environment demands,
- grant portability between systems [7],
- offer protection against dependency obsolescence.

## 2. RELATED WORK

The Max family [19] is a good example of balancing flexibility and efficiency in an always-on environment. A modular architecture of pre-compiled black-box support modular assembly into runtime interpreted *patches* by means of the graphical data-flow idiom.

SuperCollider 3 [12] grants the user flexibility to define kernels of state and synthesis function (*SynthDefs*) from low-level primitives at run-time using the interpreted *SCLang* language, and store them 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.

The Kyma system [15] in 1988 featured a capacity to define synthesis kernels (*SoundObjects*) using the interpreted Smalltalk-80 language as hierarchies of generators and transforms. *SoundObjects* are then parsed to generate assembly language code (synthesis function and data registers) for instantiation on a dedicated DSP microprocessor.

## 3. INTEGRATING LUA AV & LLVM

LuaAV [17] 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 [9] along with extensions

<sup>1</sup>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.

we have provided for timing, audio drivers, windowing and OpenGL, MIDI, OSC etc.

One of the major design issues in LuaAV has been where to situate the static-dynamic divide. Basing our work upon an interpreted dynamic language already grants extensive runtime re-configurability, such as dynamic data structures, garbage collection, code evaluation, first-class functions, etc. JIT compilation promises the extension of dynamic flexibility to what would otherwise demand ahead of time compilation, such as efficient synthesis routines. To this end, we have embedded the LLVM (Low Level Virtual Machine [10]) compiler infrastructure within LuaAV.

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 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.<sup>2</sup> The significant contribution of LLVM is the agnostic portable IR language as *linguistic switchbox* connecting all stages of compilation,<sup>3</sup> and mirroring this IR in the library application programming interface (API).

To explore the potential of JIT compilation we have created bindings to the majority of LLVM API in the Lua language. In this section we outline preliminary investigations using these capabilities within the LuaAV environment.<sup>4</sup>

### 3.1. Expression trees

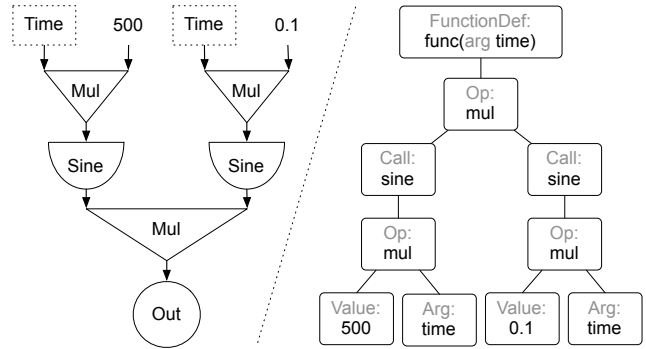
Expression trees offer a simple yet general form to describe an algorithm, consisting of a hierarchical directed acyclic graph (DAG) of operations by connecting outputs to inputs. They are applicable to computer music in the form of stateless synthesis generators and are an appealing starting point for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

For our investigation, we treat the expression as a function of time, mapping time (an immutable 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 (Figure 2a).

<sup>2</sup>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.

<sup>3</sup>The idea is not new: a common intermediate language (UNCOL [18]) was proposed fifty years ago, but not widely implemented. LLVM is a mature project with support from many groups including Apple and Adobe[1].

<sup>4</sup>Note that the Lua language is not itself being JIT compiled, but rather used to control JIT compilation of arbitrary structures and functions. Projects for just-in-time compilation of Lua do exist, but were not suitable for our goals: [13] and [11] lacked portability and [3] was not significantly faster than the stock interpreter at the time of writing.



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

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 (Figure 2b). Since LLVM IR is static single assignment (SSA), it is straightforward to define the body of JIT-able function using our Lua-LLVM binding (Figure 2c). This function is then available for use within the audio process to synthesize sound.

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

```

local dag = { expr = "mul",
  { expr = "sine",
    { expr = "mul",
      500,
      "time"
    }
  },
  { expr = "sine",
    { expr = "mul",
      0.1,
      "time"
    }
  }
}

-- or in short-hand: --
local dag =
mul(sine(mul(500,time())),sine(mul(0.1,time())))

```

**Figure 2.** Code generation of the graph in Figure 1: (a) concise constructor form, (b) linearized list of static single assignments, (c) generated LLVM IR.

### 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 runtime generation of data-structures to maintain state across

**Program 3.1** Definition and instantiation of a new kernel *phasor* (cheap cosine by vector rotation.)

```

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

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

```

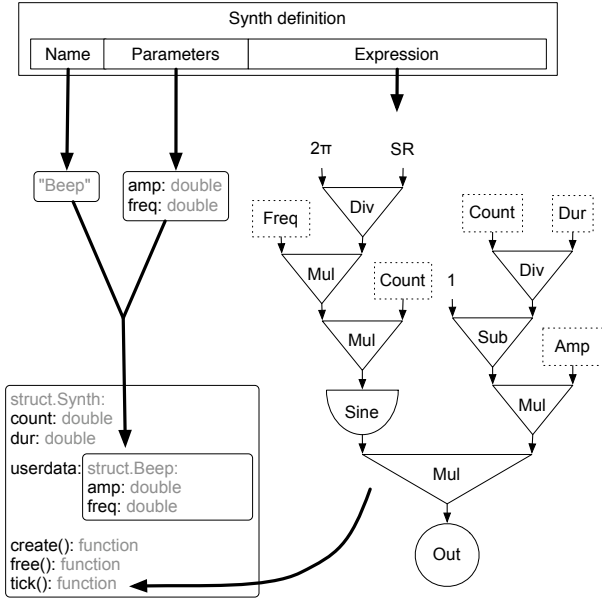
function calls, and associated routines to allocate and free memory appropriately. In order to generate optimal algorithms it is also vital to distinguish immutable state, state updated at control rates, and state updated per sample.

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 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 memory and type safety essential to avoid crashes in runtime 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 3.1.)

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) and rate (constant, control-rate or signal-rate.) Referencing variables insert appropriate *load* and *store* instructions to the AST. Prior to code generation, the parser traverses the AST and promotes the rate of each instruction the highest rate of all its argument nodes (a form of type 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 functions to allocate and free memory accordingly. The default value table is used to generate variable assignment instructions in the allocator function.



**Figure 3.** Schematic overview of the *Synth* definition in Program 3.1.

|        | 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.

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 pointers to this data structure’s members, followed by all of the non-signal rate instructions. This is followed by an inner loop block for signal rate processing into which all the signal-rate instructions are inserted. Pointers in store instructions within the inner loop are offset by the loop iterator.

The LLVM optimization passes promote most of the load and store instructions to register variables where possible, trim dead-code and restructure expressions. In our tests we found there to be little significant difference in CPU time between JIT compiled functions and equivalent C++ code.

Several performance measurements are listed in Table 3.2. The Phasor is implemented as in Program 3.1, while the Biquad implements a minimal two-pole, two-zero filter. The 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, there is no barrier against far more complex structured objects and additional methods, nor should it be conceived as 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.

Aside from using the Lua-LLVM binding, we also have a proof of concept using a Lua binding to the Clang front-end for compilation of matrix processing kernels into LLVM IR. The processing kernels are shader-like in that they 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 instead of emitting IR it emits C code for evaluation by Clang.

## 4. DISCUSSION

We have described above a range of approaches to the runtime JIT compilation of synthesis routines and processing functions that free the LuaAV composition environment from the hard boundary between static and dynamic computational elements that a bytecode representation entails. By augmenting the runtime environment with the capacity to admit new primitives and connections through JIT compilation, we have enabled the integration of new qualities to the computational system. Such environments are heterogeneous in nature and permit structural feedback loops that otherwise would not be possible. We have only just begun to explore the possibilities of this class of computational space where computation becomes a meta-material, serving not just in the development of new compositional forms, but also the exploration of new computational paradigms in and of themselves, but from our current experiences we feel it to be a vast and fruitful terrain to set out on.

## 5. REFERENCES

- [1] "2008 llvm developers' meeting," <http://llvm.org/devmtg/2008-08/>.
- [2] "fastbreeder," <http://www.pawfal.org/Software/fastbreeder/>.
- [3] "llvm-lua," <http://code.google.com/p/llvm-lua/>.
- [4] N. Collins, "The analysis of generative music programs," *Org. Sound*, Jan 2008.
- [5] N. Collins, A. Mclean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Org. Sound*, vol. 8, no. 03, p. 10, Dec 2003.
- [6] G. F. et. al., "Meta-design: a manifesto for end-user development," *Communications of the ACM*, vol. 30, no. 2, 2004.
- [7] M. Franz, "Code-generation on-the-fly: A key to portable software," *moldovacc.md*, Jan 1964. [Online]. Available: <http://moldovacc.md/acoulichev/th10497.pdf>
- [8] E. Giaccardi and G. Fischer, "Creativity and evolution: a metadesign perspective," *NDCR*, vol. 19, no. 1, pp. 19–32, Mar 2008.
- [9] R. Ierusalimschy, L. de Figueiredo, and W. Celes, "The evolution of lua," *Proceedings of the third ACM SIGPLAN conference on History of Programming Languages*, Jan 2007.
- [10] C. Lattner and V. Adve, "The llvm compiler framework and infrastructure tutorial," *Lecture Notes In Computer Science*, Jan 2005.
- [11] F. Mascarenhas and R. Ierusalimschy, "Efficient compilation of lua for the clr," *Proceedings of the 2008 ACM symposium on Applied computing*, Jan 2008.
- [12] J. McCartney, "Rethinking the computer music language: Supercollider," *Comput. Music J.*, vol. 26, no. 4, pp. 61–68, 2002.
- [13] M. Pall, "LuaJIT," <http://luajit.org/>, 2007.
- [14] J. Rohrhuber, A. de Campo, and R. Wieser, "Algorithms today notes on language design for just in time programming," *context*.
- [15] C. Scaletti and R. Johnson, "An interactive environment for object-oriented music composition and sound synthesis," *Conference on Object Oriented Programming Systems Languages and Applications*, Jan 1988.
- [16] B. Shneiderman, "Creativity support tools: accelerating discovery and innovation," *portal.acm.org*, Jan 2007. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1323689>
- [17] W. Smith and G. Wakefield, "Computational audiovisual composition using lua," *Transdisciplinary Digital Art. Sound*, Jan 2008.
- [18] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel, "The problem of programming communication with changing machines: a proposed solution," *Commun. ACM*, vol. 1, no. 8, pp. 12–18, 1958.

[19] D. Zicarelli, "How i learned to love a program that does nothing," *Computer Music Journal*, Jan 2002.