COMPILING MPEG 4 STRUCTURED AUDIO INTO C

John Lazzaro and John Wawrzynek

CS Division
UC Berkeley
Berkeley, CA, 94720
{lazzaro, johnw}@ cs.berkeley.edu

ABSTRACT

Structured Audio (SA) is an MPEG 4 Audio standard for algorithmic sound encoding, using the programming language SAOL. The paper describes a SA decoder, sfront, that translates a SAOL program into a C program, which is then compiled and executed to create audio. Performance data shows a 7.6x to 20.4x speedup compared to the SA reference MPEG decoder.

1. INTRODUCTION

MPEG 4 Structured Audio (SA) [1] [2] is a normative standard for the algorithmic coding of sound. The standard defines an audio signal processing language, SAOL, whose programs may be controlled by the musical control languages SASL and MIDI. SA encoders create programs using these languages; SA decoders execute these programs to create sound.

The classic design trade-offs for programming language execution apply to SAOL. At one extreme, SAOL programs may be directly interpreted. Direct interpreters are simple programs to write, and have the run-time advantage of low startup latency; the downside of direct interpretation is slow execution performance. The MPEG reference implementation for SA is a SAOL interpreter.

An alternative to interpretation is the direct compilation of SAOL into the machine code of the target machine. Compilers offer better execution performance, relative to interpreters, because the CPU executes the program directly. Interpreters, in contrast, involve a level of indirection: the CPU executes a program (the interpreter) that in turns executes a second program (the SAOL program). However, compilers incur higher startup latency, and are more complex than interpreters to develop and optimize.

Between these two extremes in the design space lie several hybrid approaches to SAOL execution. One approach, described in [3], uses the virtual machine concept: SAOL is translated into the instructions of a virtual processor, whose instruction set is a good target language for SAOL. This virtual-machine code is then interpreted, a task more amenable to efficient implementation than interpreting SAOL.

A second hybrid approach, which we describe in this paper, is the translation of SAOL into a C language program, which is compiled into native machine code and executed directly on the target machine.

Since the translated C program is executed directly on the underlying machine, the SAOL-to-C approach maintains the core performance advantage of pure compilation. However, by using C as an intermediate format, the SAOL-to-C approach offloads the complex task of machine code generation to the system C compiler. C is a good language for expressing the constructs that a native SAOL compiler would generate, excepting SIMD resource management.

We have written the experimental SA decoder *sfront*, which is based on SAOL-to-C translation. We begin this paper with an introduction to sfront, including benchmarks comparing sfront and the MPEG reference decoder on a small suite of test programs, showing speedups of 7.6x to 20.4x. The remainder of the paper describes the methods sfront uses to translate SAOL into efficient C code.

The paper assumes a working knowledge of the SAOL language; readers unfamiliar with SAOL should consult [1] and [2].

2. SFRONT

Sfront is a program that translates a Structured Audio program into a C language program. To generate sound, the C program sfront generates is compiled into native machine code and executed.

Sfront is written in pure ANSI C, and creates pure ANSI C files that use standard file formats for audio I/O. If an ANSI C compiler exists for a platform (including the freely-available GNU C compiler), sfront will run on the platform as a file renderer. In addition, sfront has an audio and control driver structure, so that library code may be included into the C file sfront creates for platform-specific applications.

Under the Linux platform, sfront has audio and control drivers to support audio microphone input, audio out-

	vowel [10 s]	beat [60 s]	claps [25 s]	pc [67.3 s]
saolc	40.1 s	159.4 s	31.2 s	902.4 s
sfront	2.8 s	8.6 s	4.1 s	44.2 s
speedup	14.3x	18.5x	7.6x	20.4x

Figure 1: Sfront performance comparison vs. MPEG reference implementation saolc, for rendering (to a file) four SAOL programs shipped with sfront. Bracketed times below program names are size, in seconds, of program output. All times are user times, for a 450 MHz PIII, 128 MB RAM, Linux 2.2.17; sfront times are for sfront, gcc, and C program execution. The pure ANSI C programs sfront produces do not do assembly-language SIMD access, and the gcc version (egcs-2.91.66) does not do extensive SIMD vectorization.

put, and MIDI control input. These drivers let sfront create real-time interactive software synthesizers and low-latency audio signal processors.

Sfront has been available for download via the Internet since March 1999 [4]. Over 12,000 downloads of sfront have occurred since its release, and an online book and reference manual for SAOL programmers has received over 140,000 page views.

Figure 1 compares the performance of sfront and the MPEG reference compiler *saolc*, for file rendering of four SAOL programs (written by E. Scheirer) that are shipped with the sfront distribution. Sfront runs these examples 7.6 to 20.4 times faster than saolc. See Figure 1 caption for benchmarking details.

In each example in Figure 1, the sfront running time is significantly less than the length of the audio file, measured in seconds of audio produced. This suggests that real-time streaming from the C program sfront produces would work. In practice, when sfront is configured for real-time streaming, the **vowels, beats,** and **claps** examples stream without glitches. The **pc** example has concentrated CPU bursts due to note flurries, and real-time streaming audio glitches during these moments. We plan to implement (non-normative) note dropping to handle these issues.

Sfront uses three properties of the SA standard to generate efficient C code: SAOL's static memory model, the opportunity to implement SA libraries using generator techniques, and the preponderance of implicit loops in SAOL programs. In the next three sections, we describe each method in detail.

3. MEMORY MODELS

A C fragment that accesses a variable directly usually executes faster than a C fragment that accesses a variable via one or more levels of indirection. Indirection is slow for several reasons: the added work for the CPU to do the indirection, the effect of indirection of caches, and the difficulty of removing indirection during C compilation.

Sfront capitalizes on properties of the SAOL language to generate C code with as few indirect memory references as possible, with the goal of speeding up program execution. In this section, we describe these SAOL properties in detail.

The memory usage of a SAOL instrument and its opcalls is essentially static. The language does not support recursion, automatic variables, or dynamic memory allocation of variables; all array sizes are known at compile time. While the language supports run-time sizing of wavetables, analysis of a collection of SAOL programs shows that this feature is rarely used: the size of most tables is known at compile time.

Sfront uses these facts to reduce memory indirection. Instead of using the standard dynamic-stack-and-heap approach to memory management for a SAOL instrument and its opcalls, it declares fixed-size C global arrays to hold the signal and table variable state for a SAOL instrument and its opcalls.

Sfront translates the SAOL code that accesses signal variables and tables into C statements that access these C global arrays using constant array indices. For opcalls, this technique requires generating unique C code for each opcode call; for example, three calls to the SAOL core opcode **buzz** in an instrument will result in three C functions, each pointing to different areas of global memory. This basic idea (global arrays accessed with constant indexing) is also used to represent and access all SAOL state in the global block (global signal variables, tables and audio buses).

These techniques eliminate most memory indirections in effects instruments, since C code can be generated which accesses one variable frame for the entire program. For non-effects instruments, local SAOL variable access requires a single indirection, whereas global SAOL variable access occurs without indirection.

Sfront reduces memory indirection during C code generation in other ways. Oparray calls with static indices are converted into opcode calls, to eliminate an indirection. Sfront eliminates the indirection inherent in passing tables and call-by-reference variables into SAOL opcodes, by backtracking up the SAOL calling stack during C code generation. Sfront promotes local tables to global tables if the size and state of the tables are fixed, eliminating a memory indirection. Finally, sfront eliminates the overhead of copy-in/copy-out semantics of SAOL global signal variable importation, substituting direct access if safety permits.

Note that most of the methods described above have safety caveats: in some rare situations, eliminating a level of memory indirection results in incorrect SAOL semantics. Sfront analyzes SAOL code before generating C code, to catch these illegal edge cases.

4. GENERATOR TECHNIQUES

About 30 percent of the SA standard describes the SAOL library, which includes 104 core opcodes, 16 wave table generators, 25 standard names, and 3 I/O related SAOL statements. Sfront generates custom C code for each invocation of a library element in a SAOL program. Sfront optimizes the code produced for each use of a library element, based on the *attributes* of the invocation. These attributes include the rate, width, size, constancy, and integral nature of the parameters, as well as the number of parameters.

Before C code generation begins, sfront analyzes the SAOL program, to determine these attributes for each library element call. Sfront also performs an extensive set of constant-folding optimizations before code generation, across SAOL statement, rate, and opcode boundaries. Sfront uses these attributes to craft optimal C code for SAOL library elements, using methods described below.

4.1. Wavetable Initialization

An analysis of a collection of SAOL programs shows that most wavetable generators have constant parameters, and thus can be evaluated at compile-time. Sfront detects this common case, and computes the table values at compile-time for moderate-sized arrays (8912 elements or less). Constant C arrays are generated to hold this data.

For all other tables, custom C functions are generated to create the table at run time, whose code is simplified to reflect constant parameter values. For the **sample** generator, which reads in data from a WAV or AIFF file. sfront constructs a custom parser matched to the file.

4.2. Library Inlining

For simple library elements, sfront does not create a C function; instead, the opcall is replaced with an inline snippet of C code. 40% of the core opcodes, 66% of the SAOL I/O statements, and 100% of the standard names are inlined at least some of the time.

The decision to inline, and the nature of the inline code, is often influenced by the library element attributes. For many math and pitch opcodes, constant parameters result in the compile-time evaluation of the opcode, resulting in an inlined constant. For table opcodes, a tablemap argument might disallow inlining, while an integral or constant table index value might significantly simplify the C code by eliminating interpolation. I/O related library elements such as the **input** standard name are extensively customized, based on the audio signal path.

4.3. Opcode Customization

Sfront uses attributes to fine-tune the C code created for core opcodes. Many core opcodes use a phasor to play out tables values (e.g., **oscil**) or for driving waveform calculation (e.g., **aphasor**). Sfront customizes the code for phasors, based on the rate and constancy of the **cps** parameter.

Many core opcodes involve audio-rate filtering, under the control of coefficients (e.g. **biquad**) or parametric variables (e.g., **lopass**). Sfront generates C code for these filters that significantly improves the performance for i-rate and constant control. For parametric control, this customization is done by generating different C output depending on parameter attributes. For coefficient-based filters, sfront uses a code style that C compilers can easily optimize if coefficients are constant.

The envelope core opcodes (e.g. **aline**) have an arbitrary number of opcall parameters; each segment of the envelope requires several parameters. Sfront creates C code that is customized to the number of segments: each segment adds a label to a C **switch** statement that coordinates envelope playback.

Sfront also uses attributes to guide C code generation for the most complex opcodes in the SAOL library: the Fourier opcodes **fft** and **ifft**. If the transform size is known at compile time, sfront customizes the W_{ij} tables and scaling constants; C code for overlap-add and custom windows is inserted only if these options are selected.

5. IMPLICIT LOOPS

SAOL has one explicit looping construct, the **while** statement. An analysis of a collection of SAOL programs shows that explicit loops are rarely used in SAOL code. Instead of writing looping code directly, SAOL programmers use several language features to implicitly specify loops: array arithmetic expressions, wavetables library elements, and automatic statement scheduling based on rate semantics.

Sfront creates explicit C loops for these implicit SAOL loops that are efficient by construction. In Section 4, we describe how sfront implements wavetable libraries. We discuss other implicit loop forms below.

5.1. Array Arithmetic

SAOL fully supports array arithmetic, including vector-vector and vector-scalar logical, arithmetic, and conditional expressions, and vector assignment. SAOL programmers can use array arithmetic to write multi-channel audio processing without explicit loops. Update rules for complex structures such as oscillator and filter banks may be written as a single statement.

Sfront currently translates SAOL array arithmetic into C code by fully unrolling the implicit array loops; the C global arrays that hold the signal variable state are accessed with a minimum of indirection. The result is a large block of straight-line C code that is easy for C compilers to schedule onto functional units.

5.2. Rate Scheduling

Every SAOL program executes a pair of implied nested loops: an outer loop, which executes i-rate and k-rate statements, and an inner loop, which executes a-rate statements. A SAOL decoder determines the rate of each statement in an instrument code block, and schedules the statement for execution at the appropriate rate in these nested loops.

Rate scheduling can result in inefficient code, if a statement scheduled to run at a-rate contains k-rate or i-rate subexpressions. This construction results in redundant expression evaluation. Sfront eliminates this inefficiency before C code generation begins. Statements in SAOL instruments are analyzed line by line, and slower-rate subexpressions are moved to assignment statements that run at the slower rate.

SAOL decoders are free to structure the a-rate computation in the inner nested loop to maximize instruction level parallelism, as long as normative semantics are not altered. Some decoders [3] use *blocked execution* techniques to exploit this freedom, rearranging the inner loop to compute many a-cycles of a single statement in a block, to better utilize functional units.

In contrast, sfront currently takes a simpler approach, and generates separate functions for the i-rate, k-rate, and single-sample a-rate code in an instrument. These functions are called by higher-level customized scheduling functions that sequence instrument execution for the complete SAOL program.

When we examine the C code sfront produces for a SAOL a-rate statement block in a non-trivial instrument, we generally find a dense mass of straight-line FPU-intense arithmetic computation. Modern C compilers tend to find and use instruction-level parallelism in this sort of code well. However, we hope to experiment with blocked execution techniques in future sfront work, to measure the FPU resource utilization and the memory cache behavior of this approach to inner-loop organization.

6. CONCLUSIONS

In this paper, we described the main techniques sfront uses to create efficient C code: reducing memory indirection, customizing SAOL library elements, and creating efficient C loops for implicit SAOL loops. In future work, we hope to retarget sfront to generate native code for popular instruction sets.

7. ACKNOWLEDGEMENTS

We thank Eric Scheirer for his extensive guidance during the development of sfront, and we thank the members of the SAOL email lists for numerous discussions. This work supported by Defense Advanced Research Project Agency contract number DABT63-C-0048.

8. REFERENCES

[1] Scheirer, E. D., and Vercoe, B. L. (1999). “SAOL: The MPEG-4 Structured Audio Orchestra Language.” *Computer Music Journal* 23:2, pp 31-51.

[2] International Standards Organization (1999). International Standard ISO 14496 (MPEG-4), Part 3 (Audio), Subpart 5 (Structured Audio). Geneva, CH: ISO.

[3] Le Bourhis, L., Zoia, G., Mattavelli, M., Mlynek, D. J. (1999). “An Efficient Host/Co-Processor Solution for MPEG-4 Audio Composition.” *International Conference on Consumer Electronics*, pp. 26-27, Los Angeles, California, June.

[4] Lazzaro, J. and Wawrzynek, J. (1999). www.cs.berkeley.edu/∼lazzaro/sa/