AN EFFICIENT BLOCK-BASED INTERPRETER FOR MPEG-4 STRUCTURED AUDIO

G. Zoia, C. Alberti

Integrated Systems Laboratory, ISL/LSI
Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland

ABSTRACT

The MPEG-4 Audio standard provides a toolset for Audio synthesis and Audio processing, i.e. Structured Audio (SA). SA permits to describe algorithms through its Structured Audio Orchestra Language (SAOL) programming language. Unlike some other languages of the same type, SAOL has a sample-by-sample execution structure, and this makes particularly important the overhead computation in case of an interpreted decoder implementation. This paper describes the design of an efficient virtual architecture able to exploit the data level parallelism contained in many Audio synthesis and processing algorithms and to consistently reduce the implementation overhead through a block-by-block execution.

INTRODUCTION

The new MPEG-4 Audio standard provides a toolset for Audio synthetic generation and Audio digital signal processing, namely Structured Audio (*SA*, [1]). SA is based on its *SAOL* (Structured Audio Orchestra Language, see [2]) C-like programming language. Unlike its predecessor CSound [3], SA has a *sample-by-sample* (s-b-s) execution structure. Variables are divided into init- control- and audio-rate ones, and statements can be executed at these three different and programmable rates; but syntax for a-rate instructions is not defined for blocks of length audio-rate/control-rate, rather for each single sample. If this makes possible a correct implementation of basic functions like recursive filters, on the other hand it introduces a relevant overhead in case of an interpreted implementation, the most suitable for embedded real-time engines.

We present in this paper the design of a virtual Arithmetic and Logic Unit (ALU), based on a platform independent SAOL language profiling, able to exploit the block-based data level parallelism contained in many audio synthesis and processing algorithms, and to consistently reduce the implementation overhead. In the first part the fundamental issues for an efficient SA decoding are briefly discussed; in the second part the results of the first study phase are exploited to define a virtual architecture; this is conceived to be easily optimized on modern superscalar processors and it is able to introduce a consistent acceleration, particularly when implemented algorithms do not contain feedback loops. In the last part experimental results are presented that validate the proposed approach: speed-up factors in the order of 20 are achieved in typical algorithms by our *SAINT* (SA INTerpreter) decoder over the sample-by-sample MPEG-4 reference software, on WindowsNT PCs and Solaris UNIX workstations.

FUNDAMENTAL ISSUES IN SA DECODING

The SA normative text does not specify algorithms, but rather the correct way to decode SAOL instructions, i.e. to execute statements, expressions, core opcodes (the built-in standard SA library) and routings among instruments; it follows that the computational complexity corresponding to the decoding process cannot be described neither in terms of a statistical model, for instance mean value and variance, nor in terms of a worst-case/best-case model. Actually, the decoding complexity associated with each single SA performance can theoretically range from a very low value, near to zero, to a very high one, exactly as it happens for a C or Java program. A special profiler has been conceived, which is able to count the several classes of SAOL operations, from simple mathematical operators up to more complex non-normative routines; the latter can be added-up as they are, or flexibly decomposed into more elementary ones; each parameter can be calculated at three different time granularities along the performance time axis [4]: control rate, second-per-second and total count.

The profiling of typical and reference SAOL programs revealed interesting features of typical synthesis and processing algorithms, above all concerning their relationship with the set of SAOL core opcodes. The two most interesting results of the described analysis are: a) the efficient implementation of the core opcodes through decomposition and b) the confirmation of the benefit from a *block-by-block* (b-b-b) execution scheme.

In SAOL the defined standard core opcodes are 105, but a careful analysis of them all, validated by the profiling to verify consistency, reveals that the number of *core functions* necessary to optimise them is much smaller, nearly the half. For instance, the oscillators and table reads can be reduced to two basic operations, interpolation and phasor, i.e. increment with modulo check; many specific conversion operators can be translated into a longer (in terms of number of operations) combination of simpler operations, since they are not often used; some filters present evident redundancies, and so on. On the other hand effects, mathematical operations, most of the filters, some signal generators, etc. provide a specific functionality, and often their algorithms are left open to implementers: as a consequence they require a dedicated core function.

A particular regard was dedicated to the study of the possibility of a b-b-b execution in SA, without altering the output of the normative s-b-s language specification. Efficiency of a block based execution over a sample based one has been previously proofed in literature [5]. In SA, what could prevent from executing b-b-b is the presence of an explicit feedback in the

SAOL code. By explicit feedback we intend here a feedback programmed using more than one line of code, while an implicit one is for instance the case of the *iir* or *flanger* core opcodes, where the feedback is hidden at a lower level. Explicit feedbacks have been detected only in four situations. The first possibility is when an audio variable is assigned to a new value after its first use; an increment is the only considered exception. The second case is an occurrence of the *tablewrite* core opcode executed at the audio rate, since using *tablewrite-tableread* combinations it is possible to modify and use an "object", i.e. a table, in different parts of the code; the same thing is true for the *fracdelay* core opcode managing delay lines, which has a structure similar to the tablewrite-tableread couple because of its object-oriented concept. The last case is a *while* loop executed at audio rate. All of these four cases have to be detected and treated in a special way, while the rest of the code can be executed on a possibly large b-b-b basis. Of course, this can be done if the delay introduced in the real-time synchronization of the complete MPEG-4 decoding process is tolerable.
These two main statistical results proof, confirming intuition, that in most cases an efficient implementation of the SA decoder can be obtained by the design of a virtual machine, or at least a virtual ALU, based on a vectorial instruction set.

GENERAL ARCHITECTURE OF SAINT

The SA decoder denominated *SAINT* aims at two main objectives: first of all to develop an interpreter in the most efficient way, in order to limit the overhead due to instruction interpretations; secondly to conceive an instruction set that best matches the parallelism exploitable in many state-of-the-art DSPs, processors and multimedia processors [6,7]. Concerning this last issue, SA intrinsically provides two possibilities for parallel computation: the first is a parallelism at the data level, that can be exploited when it is possible to work on vectors (blocks) of data, as previously described; the second is a parallelism at the instruction level, but only when different instances of the same instrument are active: this more precisely induces a SIMD (Single Instruction on Multiple Data) parallel architecture. The statistical analysis described earlier convinced to concentrate on the data level parallelism, which is almost always present, easy to exploit and to port on different platforms: all of the modern VLIW and SIMD architectures permit good speed-up factors for this kind of parallelism. The decision was to design a virtual ALU able to execute the SA instructions in vectorial form, with a variable length of the vector, from 1 (for s-b-s execution) to N, which is normally the length of the control cycle in samples.
The other main goal of SAINT is to preserve simplicity and effectiveness in the execution engine. The usual structure of an interpreter has been modified in several aspects, in order to limit the overhead and maintain a good portability. Aiming at a tool very similar in its software architecture to a hardware device, the first effort is to divide the complete decoding in only two layers: the scheduler/decoder layer and the instruction layer. The main reason for that is to be able to easily split the complete process into two separable parts, the compiler/control task and the real processing task; once this is accomplished, it is not difficult to keep the first, general purpose part in a common processor, and execute the intensive processing possibly in the same CPU, but with the same effectiveness in a separate co-processor, single or even distributed; this is achieved through a simple sequence of monodirectional remote method calls, after a specific resource allocation, which means allocation of the method codes and their respective calls.
In practice it is necessary, as a first step, to build a transcoder from the SAOL code to an intermediate format to be passed to the computational engine; statements and core opcodes are translated in the appropriate short sequence of macroinstructions and then interpreted by the execution unit. While doing that, the SAOL *compiler* is also able to break all the nested calls, theoretically infinite in the number of allowed levels; the vectors of values are stored in intermediate *registers* according to their rate. This flattening procedure also permits to avoid waste of time in useless evaluation functions when the actual parameter rate is lower than allowed by the opcode definition. The generated block of code, for instruments and opcodes, is additionally split into three different blocks, according to the rate of the statements to be executed (initialisation, control and audio or sampling rate).
For instance, let's consider the following SAOL example, which sends to the output bus the note *note* obtained from the wavetable *tmap[no]* that has base frequency *base*:

```
output(loscil(tmap[no], cpsmidi(note),
cpsmidi(base),loop, len)*amp);
```

Italic font represents here the opcodes at init-rate that convert from MIDI notation to cycles per second. The following listing gives a representation of a normal interpreter structure for the above code. Indentation represents a nested call:

FULL INTERPRETER APPROACH

```
eval_statement(outbus)
     eval_expression(star)
          eval_var(amp)
          eval_core_opcode(loscil)
     eval_table(tmap[no])
     eval_var(no)  // memory access
     eval_core_opcode(cpsmidi)
     eval_var(note)  // memory access
     eval_core_opcode(cpsmidi)
     eval_var(bass)  // memory access
          eval_var(loop)
          eval_var(len)
     eval_bus(bus)
```

In the next graphic the execution of the same line of SAOL code is instead represented with the virtual ALU approach. Again italic font is used to emphasize init rate instructions:

VALU APPROACH

```
i_reg[1] = eval_minus(var,69);
i_reg[2] = eval_slash(i_reg[1],12);
i_reg[3] = eval_pow(2,i_reg[2]);
i_reg[4] = eval_gettune(tmap[no]);
i_reg[5] = eval_star(i_reg[3],i_reg[4]);
. . .  // 2nd cpsmidi formula calculation
k_reg[1] = eval_var(tmap[no]); // k_rate
```

```
a_reg[1] =
eval_phasor(i_reg[5],i_reg[11],loop,len);
a_reg[2] = eval_interp(k_reg[1],
a_reg[1]);
a_reg[3] = eval_star(a_reg[2], amp);
eval_outbus(a_reg[3], bus);
```

After the decomposition, the block of code at the audio rate is checked for explicit feedbacks; the current compiler gives the possibility to label a certain number of contiguous lines as s-b-s to be executed in such a fashion, after and before two blocks executed b-b-b.

On one hand, the core opcode decompositions and the creation of intermediate registers permit to flatten the block of code and to split it properly into three blocks. On the other hand this introduces an additional number of instructions to execute; experimental results proof that this is not a heavy drawback if virtual methods for code interpretation are properly designed.

THE VIRTUAL INSTRUCTION SET DEFINITION

The first part of the virtual instruction set is composed by the SAOL set of expression operators and statements, with the exception of *while*: this is replaced by *if* / *jump_back* because, if the guard expression is a composite one and the intermediate registers mechanism is adopted, the expression to be checked begins *before* the *while* itself.

The core opcodes are the construct of SAOL in which the majority of the computation is usually executed. Indeed they constitute a heterogeneous set of functionality, and they describe very frequent and demanding operations as well as rarely used and specific ones. The objective is to isolate the computationally more complex routines to give them an entry in the instruction set table; the remaining group of opcodes is less meaningful and will not have a dedicate entry. For instance, the complete group of pitch converters is translated into the corresponding sequence of elementary operations, by mean of the definition formulas. Another example is the group of table operations *tableread*, *tablewrite* and oscillators, which constitute the core of a majority of musical and processing algorithms (wavetables, FM, and many of the most popular synthesis methods, among others). All of them are based on two main operations, interpolation and phase modulo increment, together with the unavoidable memory accesses for interpolations. In the case of e.g. the *doscil* core opcode, which loops once over a wavetable, after the boundary check the functionality is decomposed as follows:

```
i_reg[1] = get_par(t,1);  // get table_SR
i_reg[2] = div(i_reg[1], s_rate);
a_reg[1] = phasor(0, 1, i_reg[2], 1,
100);  //phases
i_reg[3] = get_par(t, 2);  // get size
a_reg[2] = mul(a_reg[1], i_reg[3]);
res = interp(t, a_reg[2]); // interpolate
```

where 3 vectorial operations, at line 3, 5 and 6, are executed at each control cycle for a block of e.g. 100 samples, if this is permitted by the algorithm. The time wasted in additional calls can be recovered avoiding tests on the audio-rate operations. The other oscillators and tableread are implemented in a similar way. The majority of the filters have an open implementation, and again they require a specific instruction; only *allpass* and *comb* can be unified, and the two different *fir* and *iir*, in the case of simple or tabulated coefficients. Fifty-three macro-instructions are enough to represent all the opcodes.

The general criteria adopted to introduce a new instruction in the set were first of all the statistical results of the profiling phase, then the normative text and the implicit feedback loops: in fact, it is not wise to break them into explicit ones. The last two issues force, in a certain sense, to keep some complex instructions in the set. This is not a great problem in software, while in a hypotetical hw implementation some aspects still need to be further investigated. Considering statements and operators, the present definition of the virtual ALU is composed of about 70 instructions. Only a single numeric format, 32-bit floating-point, is normative in SA: different instructions for different rates are not useful if the vector length is flexible.

To run a control cycle, the SA scheduler must invoke methods to execute the control- and audio-rate blocks of code. Since all nested calls have been flattened by the compiler, it only has to call functions one after the other, specifying the variables to be used and where to store the results: the scheduler works as a fetch/decode unit. In the case of a test instruction, i.e. an *if* or a *while*, the scheduler has to receive back the result of the operation, in order to decide if a subblock of code has to be executed or not, exactly as it happens in program counters for jump instructions.

EXPERIMENTAL RESULTS

The virtual ALU architecture described in the previous sections has been implemented in C (compiler) and C++ (execution unit). Different measurements on different versions of the decoder have been conducted. The SAINT tool has been compiled on two different platforms, an IBM/Cyrix at 200 MHz (Pentium compatible) with 64 MB of RAM running Windows NT4 and BorlandC++ 5.02, and a Sun UltraSPARC 2 UNIX workstation with 256 MB of RAM running SunOS 5.6 and its default cc/CC compiler. Code has been optimized for speed. Five different groups of simulations have been considered, measuring the decoding time elapsed until the end of the performance.

We report here two examples: the first is a common wavetable synthesis algorithm, where a stereo piano at 44100 Hz is generated from monophonic wavetables, and filtered by a reverberation based on a classic scheme with two allpass and four comb filters [8]. The mean polyphony of the score file, considering the effect of sustain, is approximately 3.5, the score duration is 18.5 seconds. The comparative results for the PC platform are shown in Figure 1. In the graphic, the six columns from right to left are respectively associated to: a) the MPEG-4 reference software; b) the SAINT decoder without any optimization; c) the SAINT decoder with a b-b-b execution, when possible; d) the previous decoder with the flattened structure for interpretation; e) for the PC platform, the SAINT decoder with the "Optivec" free downloadable vectorial libraries for Pentium; f) the duration of the complete score file in a real-time reproduction.

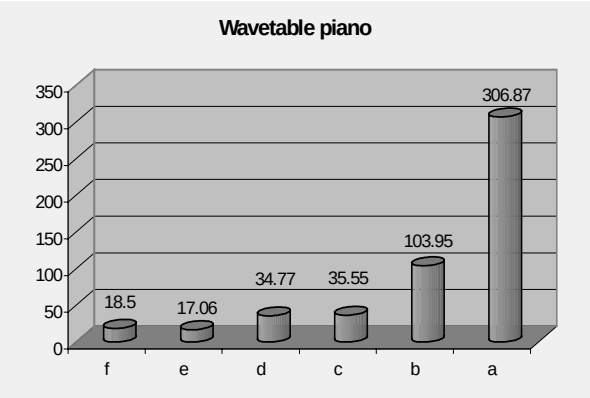

Figure 1 Experimental results for different approaches

The chosen interpolation factor is 3: C++ code is based on harmonic functions (MacLaurin series), while the vectorial libraries use spline interpolation. The b-b-b execution introduces a speed-up factor of nearly 3, here with a block length of 441. When the block of code is flattened, without nested calls, performances do not vary relevantly: this is a good result, because it permits to simplify the execution without penalty in speed, even if the total amount of functions calls has increased. Finally, the introduction of vectorial libraries on some basic functions (in this case only for interpolation, mathematical operators and summing bus) shows how this approach can be effective: consider in fact that parallelism is exploited here only at the software level, while the vectorial instruction set can be optimized with a much greater efficiency on a truly parallel co-processor.
The other synthesis example is an FM generated clarinet; the frequency modulation part of the algorithm, very similar to examples previously tested in literature [9], is essentially based on the *oscil* core opcode. The orchestra contains a reverberation effect computationally similar to that used for the wavetable piano. Experimental results are reported in Figure 2.

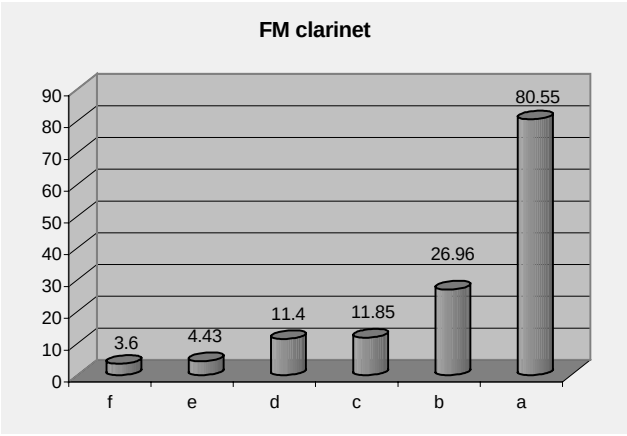

Figure 2 Experimental results for different approaches II

It is noticeable that in this example, always stereo at 44100 Hz, even the SAINT decoder with vectorial libraries does not reach enough speed for a real-time performance. In this case the profiler shows that, in one second of score performance, peaks are present of $7x10^5$ interpolations, more than 10^6 divisions and $2x10^6$ other mathematical opcodes, without considering other floating-point operations. In particular, the interpolation factor is always equal to 3.

ENVISAGED EVOLUTION OF SAINT

We have presented in this paper an efficient solution for the MPEG-4 SA decoder. What we consider as the most important evolution of SAINT is the complete migration of the proposed virtual ALU structure towards a complete SA virtual machine architecture. This further evolution will allow implementing a complete SA remote machine, able to allocate and execute programs, after having received the blocks of code from the general-purpose MPEG-4 terminal.

References

1. ISO/IEC JTC1/SC29/WG11 (MPEG98) document N2503-sub5. "Information Technology - Coding of Audio-Visual objects. Part 3: Audio. Subpart 5: Structured Audio". MPEG-4 Audio International Standard.
2. Scheirer E.D., B. L. Vercoe: "SAOL: The MPEG-4 Structured Audio Orchestra Language." Computer Music Journal 23 (2) : 23-35, 1999.
3. Vercoe, B.: "CSound: a Manual for the Audio Processing System". Cambridge, MA: MIT Media Laborytory.
4. Zoia, G. "A method for Complexity Measurements in Structured Audio". ISO/IEC JTC1/SC29/WG11 (MPEG98) document M3602, Dublin - July 1998.
5. Dannenberg, R. B., N. Thompson: "Real-Time Software Synthesis on Superscalar Architectures". Computer Music Journal 21 (3) : 83-94, 1997.
6. Espasa R., M. Valero: "Exploiting Instruction- and Data-Level Parallelism". IEEE Micro, September - October 1997 : 20-27.
7. Flynn, M. J.: "Computer Architecture: Pipelined and Parallel Processor Design". Sudbury, MA: Jones and Bartlett Publishers, 1995.
8. Roads, C., 1996. "The Computer Music Tutorial". Cambridge, MA: MIT Press.
9. Pope, S.: "Machine Tongues XV: Three Packages for Software Sound Synthesis". Computer Music Journal 17 (2) : 23-54, 1993.