A VIRTUAL DSP ARCHITECTURE FOR MPEG-4 STRUCTURED AUDIO

Giorgio Zoia and Claudio Alberti

Integrated Systems Laboratory (ISL/LSI)
Swiss Federal Institute of Technology
{giorgio.zoia, claudio.alberti}@epfl.ch

ABSTRACT

The MPEG-4 Audio standard provides a toolset for synthetic Audio generation and Audio processing called 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 the case of an interpreted decoding. This paper describes the design of a virtual DSP architecture able to exploit the data level parallelism contained in many audio synthesis and processing algorithms and to consistently reduce the implementation overhead.

1. INTRODUCTION

Since their origin, software sound synthesis and software digital audio signal processing have astonishingly evolved in functionality and acceptance [1]. The reasons for that are various: the impressive increase in computational power even in low price personal computers, the great contemporary popularity of computer generated music, and finally the migration of a musician's education towards the use of electronic and software oriented tools. It is a milestone in this evolutionary process that the new MPEG-4 Audio standard provides a toolset for Audio synthetic generation and Audio dsp, namely Structured Audio (SA, [2]). Moreover, SA is surrounded by a higher-level language for scene description (BIFS, [3]), so that a complete virtual audio environment can be described.

SA and its SAOL (Structured Audio Orchestra Language, see [4,5]) C-like programming language are conceived for multimedia and downloadable applications, even if they keep a general structure very close to that of similar tools. Unlike its predecessor CSound [6], SA has a sample-by-sample (s-b-s) execution structure: this essentially means that syntax and semantics of statements and operators are defined for a single sample, not for a block of samples of length B_l

$$B_l = \text{srate/krate} \qquad (1)$$

where srate and krate are the sampling-rate and the control-rate respectively. If this makes possible a correct implementation of basic functions like recursive filters, on the other hand it introduces a relevant overhead in the case of an interpreted implementation, the most suitable for embedded real-time engines.

We present in this paper the design of a virtual DSP architecture based on a platform independent profiling of the SAOL language; the DSP is 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 next section it is shown how it was possible to estimate and profile the computational complexity of typical algorithms in a platform independent way, and SA decoding issues are discussed; the results of the profiling phase are used to define a virtual architecture, conceived to be easily optimized on modern superscalar devices. In the last part experimental results are presented that validate the proposed approach: speed-up factors in the order of 20 are achieved for typical algorithms by our SAINT (SA INTerpreter) decoder over the sample-by-sample MPEG-4 reference software on WindowsNT and Solaris platforms.

2. FUNDAMENTAL ISSUES IN SA DECODING

A systematic approach to the implementation of the SA decoder must necessarily move from a complexity analysis of its typical applications, which may include any kind of audio application. Therefore it is necessary to provide suitable metrics to profile Structured Audio as far as possible in a platform independent and implementation independent manner, not to loose in generality, staying at the same time close to the Structured Audio normative text. In other words, the problem lies in measuring, by uniform criteria, different implementations of the SA decoder (see [4]), rather than identifying SA with a particular implementation on a specific platform, as often done in the past for computer music languages.

This section introduces a new method for measuring decoding complexity of normative Structured Audio programs: this method has been adopted in the MPEG-4 standard to define SA Levels of complexity and for SA Conformance testing [12].

The SA standard does not specify any algorithm, but rather the correct way to decode SAOL instructions, i.e. to execute statements, expressions, "core opcodes" (the standard SAOL library of Audio functions) and routings among instances of the different instruments; it follows that the computational complexity that corresponds to the decoding process cannot be described either in terms of a statistical model, for instance mean value and variance, nor in terms of a worst case/best case model. The actual decoding complexity associated with each performance can theoretically range from a very low value, near to zero, to a very high one, depending on the SAOL algorithm, on the SASL (Structured Audio Score Language) score and on the runtime dynamic changes of the control parameters (SA, as any

MPEG-4 Audio-Visual Object, can be included in interactive multimedia scenes; for more details on MPEG-4 BInary Format for Scenes, or BIFS, see [3]). It is clear that, in such a context, it is not possible to extract complexity estimations from an analysis of the encoded material, but it is necessary instead to execute or simulate the SA program. Since the implementation of a decoder can be software- and/or hardware-based, it is important to choose some parameters, called here all together the *complexity vector*, in a way that they can be useful to profile the widest possible range of implementations.

It will be shown later that a classical profiler, which provides overall results for the complete decoding, is not useful and its results are not meaningful in a real-time, time-variant and interactive context: the complexity vector must be estimated as a function of time. This means that the complexity vector $\mathbf{C}_v$ must be calculated as a discrete function:

$$\mathbf{C}_v = \mathbf{C}_v(t_i) \qquad (2)$$

where t_i is a generic instant along the whole execution time axis, and i is characterized by a suitable granularity.

To measure the complexity of programs encoded in SA, the analysis of a specific performance must be carried on considering only the number of SAOL operations and their corresponding memory requirements, making abstraction of the unpredictable overhead coming from a specific decoder solution. In the SA standard some of the statements and core opcodes of SAOL are not specified in full details but only in terms of behavioural description (mainly when they are in relation with interpolation, 3-D and effect processing). Therefore, it is necessary to carefully separate, relying on the normative text, what is mandatory in the decoding process from what is left open to the implementers. In fact, the first set of functionality corresponds to a precise algorithm or theoretical number of operations per SAOL instruction, while the second set can be classified only by "macro-oriented" criteria. All the operators, statements and core opcodes in SAOL have been grouped in classes of operations; the most complex opcodes have often been decomposed in sequences of a few simpler operations. Each class of operations (arithmetic, logical, advanced mathematical operators, interpolations, and so on) constitutes an element of the complexity vector. According to the goal of profiling, the vector can be made longer or shorter. Exhaustive details about this classification are described in the MPEG-4 Conformance Test standard [12]; in this case the complexity vector is composed by 11 elements.

2.1. Abstract profiling of Structured Audio

The principles explained above has been integrated in the actual Structured Audio reference decoder (saolc, [13]); as execution engine, it has an interpreter of the SAOL language and then it well supports enhancements for a profiling along the performance time axis.

The SA abstract profiler works as follows. Three counters are associated to each parameter belonging to the complexity vector: the first is reset every k-cycle (control cycle), the second every srate samples (one decoded second in performance time), the third always increments its value until the end of the performance. In such a way the first counter adds the vector parameter values over B_l samples, as defined in (1): multiplying by krate, in order to have an operations-per-second basis, this counter provides a profiling at a time granularity of one control cycle; the second counter is used to store the parameters added during the last second; the third counter gives the global number of operations; in the case of allocated memory, the reported value is sampled immediately before the output instant.

The necessary flexibility of the tool is simply provided by an input matrix, composed by one line for each reconfigurable SA feature and one column for each potential parameter of the complexity vector. For instance, in the MPEG-4 Conformance test the first 11 columns have entries different from zero. For each feature, it is possible to specify how many units of each parameter are necessary for the execution. Thanks to this flexible mechanism, the operations necessary for each abstract operation can easily be configured by editing the input file; in theory each architectural solution can be approximately simulated in this way.

Experiments were conducted for a wide class of examples in different conditions; among the considered one: synthesis using wavetables (piano and drums), synthesis in FM, synthesis by mixed techniques (wavetables+FM), physical modelling, processing for a professional digital mixer stripe (shelving and bell filters), 3-D Audio rendering algorithms (HRTF filtering, calculation of early echoes), reverberation. These examples come from computer music literature (for instance [7,8]), multimedia audio literature ([1] et al.), or industry. The scores used for simulations were always rather complex, with high degree of polyphony (for synthesis) and many changes in control parameters, to force different subtrees in the programs to be executed.

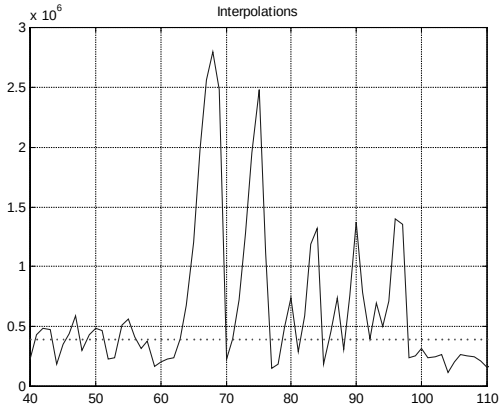

Figure 1 *Number of interpolations in "Claire de lune" by C. Debussy. 110 seconds of score time are extracted from the complete profiling. Horizontal axis is time in score seconds; the dotted line is the mean value along the complete decoding*

A typical output for interpolations in a classic piano piece is shown in Figure 1. It is obvious from this example that the mean value, represented by the dotted line, is not of any importance to guarantee real-time performances: only the most critical, worst-case intervals must be considered. This result alone, if a good quality interpolation is assumed, shows why the SA decoding can result in a heavy task even for a fast CPU, and indeed the MPEG-4 reference software in this case is far from a real-time

performance on a general-purpose computer. Experimental results also show that a time granularity of 1 second is enough for a consistent simulation and that a fast control rate is only required to achieve acceptable reaction times to changes.

The abstract profiling of many SA programs revealed interesting features of typical synthesis and processing algorithms, well-known ones and more subtle ones, above all concerning their use of the set of SAOL core opcodes.

In SAOL the 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 tablereads 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 and other opcodes provide a specific functionality, and often their algorithms are left open to implementers: as a consequence they require a dedicated core function.

2.2. Feedback analysis in Structured Audio

A second phase of the analysis was dedicated to the study of the possibility of a *block-by-block* (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 [8]. In SA, what can 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 by a simple graph analysis only in a few situations. The most obvious is when an audio variable is assigned to a new value after its first use. These 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 introduced delay in the real-time synchronization of the complete MPEG-4 decoding process (see [3]) is tolerable.

The two main results of subsections 2.1 and 2.2 proof, confirming intuition, that in most cases an efficient implementation of the SA decoder can be obtained by the design of a multimedia device, or at least a DSP, based on a vectorial instruction set.

3. THE SAINT VIRTUAL DSP

The SA decoder denominated SAINT aims at two main objectives: first of all to develop an easily understandable 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 [8,9]. 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) type of parallel architecture. The statistical analysis described earlier invited to concentrate on a 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 final decision was to design a virtual DSP with an ALU able to execute the SA instructions in a 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.

Aiming at a tool very similar in its software architecture to a hardware device, the first effort has been to divide the complete decoding in only two layers: the compiler/scheduler 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 run the first phase in a general purpose processor, and to 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 method calls after a specific resource allocation, which means allocation of the method codes and their sequence.

In practice a transcoder is necessary from the SAOL code to an intermediate format to be passed to the computational engine; here the core opcodes are possibly 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 returned values are stored in intermediate registers according to their rate; this also permits to avoid waste of time in useless evaluation functions when the actual parameter rate is lower than allowed by the opcode definition. On one hand, the core opcode decomposition 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 it introduces an additional number of instructions to execute; this is experimentally proved as not being too heavy if virtual methods for code interpretation are properly designed: see next section for experimental results.

The generated block of code, for an instrument or a user-defined opcode, is additionally split into three different blocks, according to the rate of the statements to be executed (initialisation, control and audio or sampling rate).

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

In the following of this section the two main features of the architecture of the SAINT virtual DSP will be introduced, namely its memory structures and the instruction set.

3.1. Memory Structures

There are two main groups of memory structures, relating respectively to the *Instrument* and its *Instances*. The Instrument

memory contains first the Instruction Memory (IM), i.e. the space allocated to contain the several instructions for both the instruments and the related user defined opcodes. Opcodes, either core or user defined, are expanded inline into the main block, except in the case of *oparray* (an array of "processing cells"), when a static and self-standing "opcode space" is required for each element of the array. Besides instructions, the Instrument memory also contains registers; there are two main types of register: general-purpose registers that contain intermediate calculations from expressions, and specific registers, which mainly contain SA global standard variables plus some architecture-specific variables used for code processing: the *PC*, *start*, *end* and the *block* size (for block processing), *ret_address* (to return from subblocks). The Instance memory space contains: memory space for local variables, memory space for actual parameter lists, local standard names, allocated space for buffers and delay lines in filters and memory-related opcodes. In SAINT, the memory structures for the instance are visible to the instrument block through indirect addressing, and the switching among the possible several active instances is done by changing the content of the base address.

3.2. The virtual DSP Instruction Set

The instruction set of the SAINT virtual DSP is composed by two main groups: *macroinstructions* and *instructions*. The former constitute the core of the SAINT processing; they represent the instruction set that is directly executed by the ALU of the machine. All of them are defined in vectorial form, where the block of data on which the instruction is executed is defined by the value of the special registers start and end. Unlike in the case of e.g. the Java machine [14], in SAINT there is no stack to work on, all the instructions directly operate on memory locations, and then they are defined with (normally) two or three addresses where to load and store data. For instance:

madd x,y,z;

adds the two vectors y and z and stores the result in x;

mmin s,t,u;

calculates the value-per-value minimum of the t vectors referenced starting from u and stores the result in s; and so on. Macroinstructions are conceived to behave similarly to Java native methods, and then optimized blocks of native code for the real hardware can be loaded to execute the SAOL code.

A first group of the virtual DSP macroinstruction set is composed by the SAOL set of expression operators and statements, with the exception of while, which is replaced by an if/jump_back.

The core opcodes constitute the part of SA 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. As it was for the MPEG-4 complexity analysis, the objective is to isolate the computationally more complex and frequent routines to give them an entry in the instruction set. For instance, many of the mathematical methods are important and used intensively, but some of them are seldom used and can be decomposed in one or a few lines.

For instance, an interesting example of redundancy is given by table operations: tableread, tablewrite and oscillators, which constitute the core of a majority of musical and processing algorithms (among others wavetables and FM, i.e. the most popular synthesis methods), are based on two main operations, interpolation and modulo increment of the phase, other than the unavoidable memory accesses and scalings. Considering, as an example, the case of the *doscil* core opcode, which loops once over a wavetable, the functionality is decomposed as shown in Figure 2:

```
i_reg[1] = get_par(t, 1);
     // get table sampling rate

i_reg[2] = div(i_reg[1], s_rate);
     // table_sr / sr

a_reg[1] = phasor(0, 1, i_reg[2], 1, 100);
     //Phases

i_reg[3] = get_par(t, 2);
     // get table size

a_reg[2] = mul(a_reg[1], i_reg[3]);

result = interp(t, a_reg[2]);
     // interpolate
```

Figure 2 Decomposition *in SAINT bytecode of the SAOL doscil core opcode. Lines in italic are executed at srate, lines in normal font are executed at the init rate.*

Three vectorial operations, at third, fifth and sixth line are executed every control cycle for a block of e.g. 100 samples. The other oscillators and tableread are implemented in a similar way.

The general criteria adopted to define 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 macroinstructions in the set. This is not a great problem in software, while in a hypothetical hardware implementation some aspects still need to be further investigated. In the end, 53 macroinstructions are enough to represent all the opcodes. Considering statements and operators, in the current definition the macroinstruction set for the virtual *ALU* (arithmetic-logic unit) is composed of about 70 elements. Different macroinstructions for different rates are not needed since the vector length is flexible.

To understand the role of the few *instructions* it is better to analyse, as an example, the execution of an instrument's control cycle. The block of code at the control rate is first executed; of course, the macroinstructions are used in scalar mode (vector size is 1). If the block at srate is executable completely b-b-b, this second group of instructions is executed in sequence like the previous one, except that now vector size is not one. Otherwise it is necessary to execute in s-b-s some parts of code: instructions *p_set*, *p_inc*, *p_jump*, *p_return* are used to manage special registers like *start* and *end* to execute the block of code, as shown

in Figure 3. Other instructions are used to access global variables, and in general for communication with the scheduler. This latter, in the proposed architecture, is nothing else than a "hardwired" master DSP able to coordinate the complete real-time process.

```
// block of code at control rate
…

// first b-b-b audio block
p_set end 1   // set end to 1

sbs: p_jump block end length;
      //jump to block if values of end and block are equal

// s-b-s block of code
p_inc start  // increment start
p_inc end    // increment end
p_return sbs       // go back to sbs

block:
//second b-b-b audio block
…
```

Figure 3 *Example of the SAINT virtual DSP block of code: execution of a control cycle of an instrument. The prefix "p_" is used to differentiate instructions from macroinstructions.*

4. EXPERIMENTAL RESULTS

The virtual DSP architecture described in the previous sections has been implemented in C (compiler) and C++ (execution unit). Several measurements on different versions of the decoder have been conducted. The SAINT tool has been compiled on two different platforms, an Intel Pentium II at 400 MHz with 128 MB of RAM running Windows NT4, and a Sun UltraSPARC 2 at 360 MHz with 256 MB of RAM running SunOS 5.6. The decoder was compiled on SUN using the default SunOS cc/CC compiler, while the PC version was compiled using BorlandC++ 5.02; optimization for speed was introduced. Many different groups of simulations have been conducted, taking as a result the decoding time elapsed until the end of the performance.

The first example that we report here 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 [1]. The mean polyphony of the score file, considering the effect of sustain, is approximately 3.5, the score duration is 18.5 seconds. The decoding times for the PC platform are shown in Figure 4. In the graphic, the six columns from right to left are respectively associated to: 1) the reference software as of DIS (draft international standard) version; 2) the SAINT decoder without any optimization; 3) the SAINT decoder with a b-b-b execution, when possible; 4) the previous decoder with the flattened structure and intermediate registers; 5) for the PC platform, the SAINT decoder with the "Optivec" free downloadable vectorial libraries for Pentium; 6) the duration of the complete score file.

The chosen interpolation factor is 3: for this, C++ code is based on harmonic functions, while the vectorial libraries use spline interpolation. The SAINT decoder without any optimization apparently works more than twice faster than the reference software. This huge gain comes first of all from a better instruction and data management, since all variables and memory spaces are addressed directly by pointers inside instruction "objects" themselves; secondly there is a more efficient interpolation function (based on McLaurin series), which in this example is the predominant operation. 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.

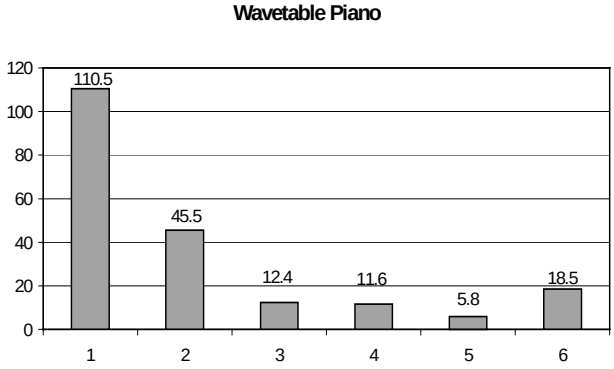

Figure 4. *Experimental results for different decoding approaches: I. Y-axis is time in seconds. The values of the six columns from left to right are the decoding time for: 1) the MPEG-4 SA reference software; 2) the SAINT decoder without any optimization; 3) the SAINT decoder with a block-by-block execution, when possible; 4) the previous decoder with the flattened structure for interpretation; 5) the SAINT decoder with the "Optivec" free downloadable vectorial libraries for Pentium; 6) the duration of the complete score file*

A second synthesis example is an FM-generated brass with wavetable-generated attack (nearly 1 second); the frequency modulation part of the algorithm is based on the *oscil* core opcode (oscillator over a table that does not have its own base frequency), in a way very similar to examples tested in literature [11]; the orchestra contains a reverberation effect computationally similar to that used for the wavetable piano. Experimental results are reported in Figure 5. It is noticeable that in this example, always stereo at 44100 Hz, even the SAINT decoder with vectorial libraries does not overpass too much the necessary speed for a real-time performance. In particular, for interpolation the factor is always 3. In this case, the profiler shows that, in one second of score, peaks are present of $7x10^5$ interpolations, more than $3.5*10^6$ multiplications and $3x10^6$ mathematical methods, without considering tests and other floating-point operations. Due to this purely mathematical content, the gain with vectorial libraries is impressive.

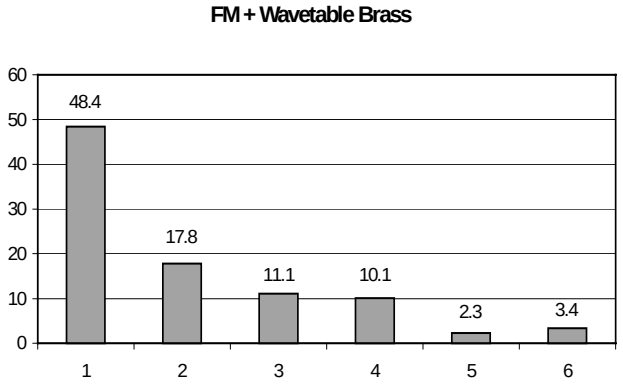

Figure 5 *Experimental results for different decoding approaches: II. Y-axis is time in seconds. The values of the six columns from left to right are the same of Figure 2.*

The physical bass example implements a waveguide synthesis model at 44100 Hz. The algorithm makes heavy use of the lopass, allpass, and tableread opcodes, other than mathematical ones, and is characterized by an s-b-s subblock between two vectorial blocks. The results for a short monophonic score without any additional processing are reported in Figure 6. It is noticeable that, since interpolation is no more predominant, the gain over the reference software is reduced to a factor lower than 2 and that the presence of an important s-b-s block of code again reduces the speed-up factor due to b-b-b processing. Moreover, the vectorial libraries do not contain the appropriate functions and we were not able to produce meaningful results for this case.

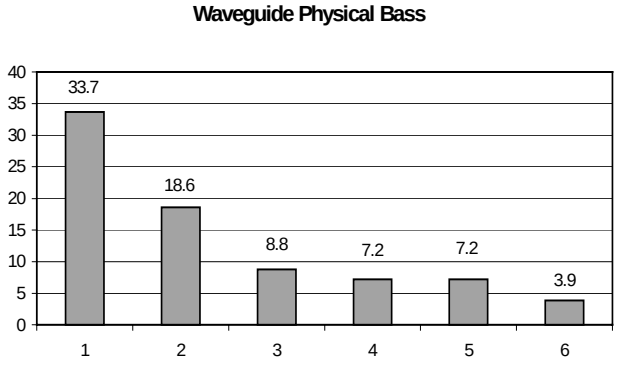

Figure 6 *Experimental results for different decoding approaches: III. Y-axis is time in seconds. The values of the six columns from left to right are the same of Figure 2.*

5. CONCLUSION

We have shown in this paper how the MPEG-4 SA decoding process has been analysed in a platform independent way and how the proposed method has been used for the MPEG-4 standardization process. Afterwards the design of a virtual DSP architecture has been presented, based on the results of the complexity analysis; this architecture can exploit the data level parallelism contained in many audio algorithms. Experimental results prove the effectiveness of the approach and its suitability for implementations on modern superscalar DSPs and multimedia processors. Future work will be dedicated to the final specification of the scheduler and to allow several virtual DSP to work in parallel under its control. This will permit to build a system able to exploit further degrees of parallelism.

6. REFERENCES

[1] Roads, C., 1996. "The Computer Music Tutorial". Cambridge, MA: MIT Press.

[2] 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.

[3] Scheirer, E., J. Huopaniemi and R. Väänänen "AudioBIFS: The MPEG-4 Standard for Effects Processing." Proc. DAFX98 Workshop on Digital Audio Effects, Barcelona, Nov. 1998

[4] Scheirer, E.: "SAOL: the MPEG-4 Structured Audio Orchestra Language". In Proceedings of the International Computer Music Conference. Ann Arbor, MI, October 1998.

[5] Scheirer E.D., B. L. Vercoe: "SAOL: The MPEG-4 Structured Audio Orchestra Language." Computer Music Journal 23 (2) : 23-35, 1999.

[6] B. Vercoe: "CSound: a Manual for the Audio Processing System". Cambridge, MA: MIT Media Laboratory, 1993

[7] Zoia, G. "A method for Complexity Measurements in Structured Audio". ISO/IEC JTC1/SC29/WG11 (MPEG98) document M3602, Dublin - July 1998.

[8] Dannenberg, R. B., N. Thompson: "Real-Time Software Synthesis on Superscalar Architectures". Computer Music Journal 21 (3) : 83-94, 1997.

[9] Espasa R., M. Valero: "Exploiting Instruction- and Data-Level Parallelism". IEEE Micro, September - October 1997 : 20-27.

[10] Flynn, M. J.: "Computer Architecture: Pipelined and Parallel Processor Design". Sudbury, MA: Jones and Bartlett Publishers, 1995.

[11] Pope, S. T.: "Machine Tongues XV: Three Packages for Software Sound Synthesis". Computer Music Journal 17 (2): pag. 23-54. MIT Press, 1993.

[12] ISO/IEC JTC1/SC29/WG11 (MPEG99) document N3067-sub3. "Information Technology - Coding of Audio-Visual objects. Part 4: Conformance. Subpart 3: Audio Conformance". MPEG-4 Audio International Standard.

[13] MIT Media Laboratory - MPEG-4 SA Homepage - http://sound.media.mit.edu/mpeg4

[14] Lindholm T. and F. Yellin "The JAVA Virtual Machine Specification". 2nd Edition (JAVA series), Addison Wesley, 1999.