

A VIRTUAL MODEL FOR SIMULATION AND DESIGN OF ARCHITECTURES IN MPEG-4 AUDIO AND MULTIMEDIA CONTEXT

THÈSE N° 2362 (2001)

PRÉSENTÉE AU DÉPARTEMENT D'ÉLECTRICITÉ

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUES

PAR

Giorgio ZOIA

laurea in ingegneria elettronica, Politecnico di Milano, Italie
de nationalité italienne

acceptée sur proposition du jury:

Prof. D. Mlynek, directeur de thèse
Dr A. Basso, rapporteur
Dr U. Horbach, rapporteur
Prof. M. Kunt, rapporteur

Lausanne, EPFL
2001

ACKNOWLEDGEMENTS

I would like to express my gratitude to professor Daniel Mlynek first of all for the possibility that he offered me to pursue my PhD at the Integrated Systems Laboratory of the EPFL. After that, without his expert guidance and frank suggestions my work would have not been the same.

I would like to thank in a very special way dr. Marco Mattavelli, for years of unselfish encouragements, help, suggestions and guidance.

Another specially grateful thought is to Claudio Alberti. For years he has shared with tremendous skill and team spirit a considerable part of the work, problems and solutions of the Audio group at the ISL.

I would like also to thank the R&D Group of Studer Professional Audio AG for a fruitful and pleasant collaboration on several projects and activities.

I would like to thank Laurent Le Bourhis, Daniel Farre and Kasper Claes for their contributions.

I am grateful to Pierre-Yves Rausis and Paul Debeve for the coordination of the technical life at the Laboratory; and to Claude Cellier at Merging Technologies for showing enthusiasm and supporting my projects.

Many thanks to all my colleagues at the ISL, especially to Massimo Ravasi, and to my former and present office mates Alexander Schmid, Théo Randriamalazarivo and Antonio Romeo. Last but not least, many thanks to Alexandar Simeonov for strengthening the Audio group in competence and enthusiasm.

Finally, I would like to express my deepest love and gratitude to my parents and to my wife, Rosina, for their support and encouragement during all these years.

TABLE OF CONTENTS

Acknowledgements	i
Table of Contents	iii
List of Figures	vii
List of Tables.....	ix
Version Abrégée.....	xi
Abstract.....	xiii
Chapter 1. Introduction.....	1
1. CONTEMPORARY AUDIO AND MULTIMEDIA SYSTEMS	1
2. SCOPE AND GOALS OF THE WORK.....	2
2.1. Difficulties in the Design of Complex Media Applications and Systems ...	3
2.2. Goals of the Work.....	3
2.3. Contributions	5
3. ORGANIZATION OF THE DISSERTATION	5
Chapter 2. Processing Units for Multimedia: Simulation and Design	7
1. MEDIA PROCESSING: TRENDS OF GENERAL PURPOSE PROCESSORS AND DSPS	7
1.1. The Current Generation of Processing Units.....	7
1.2. Multimedia Processors	8
1.3. Future Trends of Processing Units	9
2. MULTIMEDIA APPLICATIONS: COMPLEXITY AND PORTABILITY.....	10
2.1. Complexity Issues in the Development of Software	10
2.2. Media Software and Portability.....	11
2.3. Remarks on Hardware/Software Partitioning	11
3. SIMULATION OF VERY COMPLEX DEVICES FOR MULTIMEDIA APPLICATIONS	12
3.1. Simulation of Computing Devices	12
3.2. Overview of Some Instruction Set and Computer Simulators	14
4. MODERN TRENDS OF DESIGN FOR EMBEDDED SYSTEMS AND SYSTEMS ON A CHIP	16

4.1.	Hardware/Software Codesign	17
4.2.	Block-based System Design	20

Chapter 3. Multimedia Applications: Standards for the Transmission of Interactive Scenarios 23

1.	THE MPEG-4 INTERNATIONAL STANDARD	23
1.1.	General Overview	23
1.2.	Audio Coding in MPEG-4	26
1.3.	The MPEG-4 Systems Layer: BIFS Scene Description	26
2.	STRUCTURED AUDIO CODING AND MPEG-4	28
2.1.	Structured Representations of Audio Content.....	28
2.2.	Structured Audio in MPEG-4	31
2.3.	Structured Audio as a General Coding Scheme	34

Chapter 4. Towards a Language-Oriented Virtual Model for Simulation of Real-Time Applications 37

1.	MULTIMEDIA SYSTEMS, SIMULATION AND COMPLEXITY	37
1.1.	Standards and Design of Applications	38
1.2.	MPEG-4 and the Design of Applications	39
2.	VIRTUAL ANALYSIS OF A PROGRAMMING LANGUAGE	40
2.1.	Programming Paradigms	40
2.2.	Paradigms and Multimedia Applications	42
2.3.	The Definition of the Virtual Model.....	43
2.4.	Functionality Abstraction in Imperative Languages	45
3.	DYNAMIC ANALYSIS OF A PROGRAM EXECUTION	49
3.1.	General Remarks	49
3.2.	Analysis by Virtual Machines.....	49
3.3.	Dynamic Analysis and SA	50
4.	CASE STUDY: STRUCTURED AUDIO ORCHESTRA LANGUAGE	50
4.1.	Parameters for Complexity Analysis of SA Programs	51
4.2.	The SA-based Abstract Simulator	55
5.	MPEG-4 SA CONFORMANCE TEST	59
5.1.	Complexity Vectors Again.....	59
5.2.	SA Conformance Test	59
5.3.	Extensions to AudioBIFS Conformance Test.....	64

Chapter 5. A Portable Virtual Approach for Digital Signal Processing: SAINT..... 67

1.	MPEG-4 SA DECODING ISSUES	67
1.1.	Execution Block-by-Block.....	67
1.2.	Feedback Analysis	68
2.	STRUCTURE OF THE SAINT SOFTWARE TOOL.....	69
2.1.	Towards a Virtual DSP Architecture.....	69

2.2.	The SA Decoder	69
3.	THE SAINT VIRTUAL DSP ARCHITECTURE	72
3.1.	Memory Structures	72
3.2.	The Instruction Set.....	73
4.	IMPLEMENTATION AND EXPERIMENTAL RESULTS	76
4.1.	Implementation of SAINT	77
4.2.	Experimental Results.....	78
4.3.	Another Comparison with Existing SA Decoders.....	84

Chapter 6. The Project ThreeDSPACE: a Complete 3-D Audio Rendering Framework for MPEG-4 Applications 89

1.	ADVANCED 3-D AUDIO APPLICATIONS AND AUDIOBIFS: SCOPE OF THREEDSPACE	89
1.1.	Overview of ThreeDSPACE	89
1.2.	BIFS and Advanced AudioBIFS (AABIFS).....	91
1.3.	Conclusions.....	94
2.	MULTIMEDIA EXTENSIONS TO SAINT	96
2.1.	From SAINT to BIFSAINT	96
2.2.	The Final BIFSAINT Architecture	98
3.	OVERVIEW OF WAVE FIELD SYNTHESIS	99
3.1.	Current 3-D Techniques	99
3.2.	Wave Field Synthesis	100
3.3.	WFS, MPEG-4 and ThreeDSPACE.....	102
4.	THE THREEDSPACE ACTUAL SYSTEM	103
4.1.	Hardware and Software Requirements	103
4.2.	The Loudspeakers	105
4.3.	The MPEG-4 Decoder End	105
4.4.	The Physical DSP Platform.....	106
4.5.	The Final System Structure.....	109
5.	PORTING SAINT ON A VLIW ARCHITECTURE.....	112
5.1.	The TriMedia Multimedia Processor	112
5.2.	Code Optimization for the TriMedia	115
5.3.	A Vectorial Library for the TriMedia.....	122

Chapter 7. A Macro-Oriented Virtual Approach for Simulation and Design of Multimedia Architectures 125

1.	MEMORY SIMULATION: A MACRO-ORIENTED APPROACH.....	125
1.1.	Precise Cache Memory Simulation.....	126
1.2.	Approximated Macro-oriented Cache Simulation	126
2.	A MACRO-ORIENTED VIRTUAL SIMULATOR FOR MULTIMEDIA ARCHITECTURES	127
2.1.	The Cache Memory Tracer.....	127
3.	EXPERIMENTAL RESULTS	129
3.1.	General Remarks	129
3.2.	Results for a PC Platform	131

3.3.	Results for the TriMedia Platform	142
4.	QUALITATIVE COMPARISON WITH OTHER SIMULATORS	145
4.1.	Comparison with Similar Simulators	145
4.2.	Possibility of Application to Other Programming Languages	146
5.	CONCEPTUAL EXTENSION TO THE AUTOMATIC DESIGN OF ARCHITECTURES	148
5.1.	Automatic High-level Design	148
5.2.	Block-based System Design	149
Chapter 8. Future Work and Conclusion.....		151
1.	A SYSTEM FOR ADVANCED APPLICATIONS: CARROUSO.....	151
1.1.	A Complete MPEG-4 Coding Chain	151
1.2.	Practical Application Scenarios	154
1.3.	Further Future Applications of the SAINT Simulator	154
2.	CONCLUSION.....	155
Appendix A. The SAINT Virtual DSP Instruction Set.....		159
1.	THE SAINT INSTRUCTION SET	159
1.1.	Macroinstructions	160
1.2.	Instructions	175
Appendix B. The SAINT Simulator Complexity Vector		177
References		179
Curriculum vitæ.....		189

LIST OF FIGURES

Figure 1. Relationship among the different parts of the work described in this dissertation	6
Figure 2 Basic steps for the codesign approach	17
Figure 3 An example of an MPEG-4 Audio-Visual Scene	25
Figure 4 Logical structure of the scene represented in Figure 3.	27
Figure 5 Example of Audio scene description using the BIFS nodes	28
Figure 6 Major elements of the MPEG-4 Structured Audio toolset	34
Figure 7 A graphical representation of the proposed approach.	44
Figure 8 Graphical representation of a practical hardware/software codesign approach	45
Figure 9 Number of interpolations in "Claire de lune" by C. Debussy.	57
Figure 10 Number of mathematical methods in "Claire de lune" by C. Debussy.	58
Figure 11 Number of MACs in "Claire de lune" by C. Debussy.	58
Figure 12 Number of interpolations in "SY008" test sequence.	61
Figure 13 Cross-section along the time axis at t=36s for the test sequence SY008.	62
Figure 14 SAOL code interpretation by a common approach.	71
Figure 15 SAOL code interpretation by the VM approach	72
Figure 16 Main memory structures of the SAINT virtual DSP architecture	73
Figure 17 Decomposition in SAINT bytecode of the SAOL doscil core opcode.	75
Figure 18 Example of the SAINT virtual DSP block of code.	76
Figure 19 Experimental results for different decoding approaches (I).	80
Figure 20 Comparative results for PC and Unix workstation platforms.	81
Figure 21 Experimental results for different decoding approaches (II).	82
Figure 22 Experimental results for different decoding approaches (III).	83
Figure 23 Experimental results for different decoding approaches (IV).	83
Figure 24 Performances for different block sizes on Intel platform.	84
Figure 25 Parameters normally manipulated by the content provider and perceived by the listener in an advanced 3-D Audio description.	90
Figure 26 The ellipsoids that define the Sound spatial rendering in both VRML and AudioBIFS.	91
Figure 27 An example of BIFS code.	92
Figure 28 An example of BIFS code.	95
Figure 29 The final BIFSAINT multi-DSP architecture	99
Figure 30 Model-based Wave Field Synthesis system.	101
Figure 31 Data-based Wave Field Synthesis system	102
Figure 32 The CreamWare Pulsar Audio board.	108
Figure 33 The Mykerinos soundboard block diagram	109
Figure 34 High-level block diagram of the complete ThreeDSPACE Audio system .	110
Figure 35 Block-diagram of the ThreeDSPACE rendering system.	112
Figure 36 The Philips TriMedia architecture	113

Figure 37 The TriMedia memory structures and relating interconnections	115
Figure 38 General C++ code for the SAINT lopass macroinstruction	116
Figure 39 Grafting optimization.	118
Figure 40 Performance of the lopass macroinstruction	120
Figure 41 Instruction Level Parallelism (ILP) in the case of the lopass macroinstruction for different levels of optimization.....	121
Figure 42 Instruction Level Parallelism for the compressor vectorial macroinstruction.....	123
Figure 43 Structure of the cache memory simulator.....	128
Figure 44 Estimated and measured decoding time for the Brass sequence with a block size of 50.	133
Figure 45 Estimated and measured decoding time for the Brass sequence with a block size of 100.	133
Figure 46 Estimated and measured decoding time for the Brass sequence with a block size of 441.	134
Figure 47 Estimated and measured decoding time for the Clarinet sequence with a block size of 100.	136
Figure 48 Estimated and measured decoding time for the Bass sequence with a block size of 100.....	137
Figure 49 Estimated and measured decoding time for the 2-channels Mixer sequence with a block size of 100.	137
Figure 50 Estimated and measured decoding time for the Parametric Mixer sequence with a block size of 100.	138
Figure 51 Percentage of the estimated decoding time over the actual measured time for the Brass sequence with a block length of 100.....	139
Figure 52 Percentage of the estimated decoding time over the actual measured time for the 2-channels mixer sequence with a block length of 100.....	140
Figure 53 Percentage of the estimated decoding time over the actual measured time for the Bass sequence with a block length of 100.....	140
Figure 54 Estimated and measured decoding time for the Clarinet sequence on the TriMedia processor with a block size of 64.	144
Figure 55 The high-level block diagram of the complete CARROUSO system.....	153

LIST OF TABLES

Table 1 Main features of some system simulators presented in literature	14
Table 2 Summary of the SPEC'95 CINT95 benchmark suite	37
Table 3 Algorithmic Synthesis Complexity Values for Levels	60
Table 4 AudioFX node Complexity Values for Levels	64
Table 5 A comparison between SAINT and sfront decoding approaches	85
Table 6 A comparison between SAINT and sfront	86
Table 7 Format of the control channel over the 64 24-bit samples	111
Table 8 The 27 TriMedia functional units	114

La recherche dans le multimédia pour l'électronique de consommation est dominée par le problème du temps de marché extrêmement court, ce qui signifie des estimations rapides de complexité et des conceptions rapides d'architectures. D'une part des applications de plus en plus sophistiquées et flexibles sont rapidement développées; d'autre part la croissance exponentielle dans la puissance de calcul des circuits intégrés semble être à peine capable de suivre les conditions requises par les applications temps réel, car leur complexité croît aussi de façon exponentielle.

La performance d'un processeur, souvent réduite par la vitesse limitée des mémoires et des bus, est ultérieurement diminuée par le temps perdu dans la communication entre les différents niveaux de l'application. De ce problème, naît la conception de cadres de développement intégrés pour l'intégration et la simulation. Les outils pour la simulation et l'analyse d'architectures sont diffusés par des laboratoires de recherche académiques et industriels surtout dans cette dernière décennie. La plus part d'entre eux sont conçus pour obtenir une simulation exacte de bas niveau des dispositifs supportés, au prix de forts ralentissements dans le temps de simulation et d'énormes tailles des fichiers de données à la sortie. Quelques autres, dans les dernières années, ont introduit des degrés d'approximation dans les simulations, pour accélérer le temps d'exécution et pour augmenter la flexibilité des outils afin de supporter les multiprocesseurs. Les modèles résultants, plus ou moins abstraits, ne sont de toute façon pas aptes à l'analyse des vraies applications orientées au multimédia, où les programmes sont habituellement disponibles dans un langage qui inclut des bibliothèques dédiées et les résultats significatifs sont seulement ceux mesurés en fonction du temps.

A un autre niveau, les outils pour le hardware/software codesign ou pour la conception de systèmes basé sur des blocs donnent des résultats plus utiles, mais pour être efficaces ils doivent s'appuyer sur des noyaux rigides qui ne permettent que peu de reconfiguration ; des outils très récents sont conçus pour projeter des systèmes complexes en manipulant des blocs à travers des langages de description de haut niveau.

Le modèle virtuel et son outil relatif proposés dans ce rapport s'inspirent au contraire d'une approche à l'analyse de la complexité qui veut être le plus possible indépendant de la plate forme. La méthode est basée sur les concepts de classes d'opérations abstraites et de simulation en fonction du temps de performance.

Le travail décrit dans ce rapport trouve son champ d'application dans le monde du multimédia, plus précisément dans les applications audio orientées multimédia. Les applications multimédia sont communément programmées par des langages impératifs ou orientés objets qui sont composés par plusieurs différents instructions, opérateurs et surtout bibliothèques standard. Une analyse de profil attentive des applications typiques permet de détecter les opérations et les fonctions fondamentales et de définir un jeu d'instructions virtuel, en regroupant

les opérateurs plus ou moins similaires et en subdivisant les fonctions en blocs de base qui les constituent. Le résultant jeu d'instruction virtuel ne correspond à aucun autre réel; au contraire il a comme propriété d'être facilement converti en un grand nombre de jeux existants. La simulation d'une architecture requiert donc la disponibilité de mesures ou d'estimations d'au moins un membre de chaque classe abstraite. Le nombre de classes, c'est-à-dire le vecteur de complexité, peut être adapté en longueur et en détail au degré nécessaire de précision et à la quantité disponible de mesures.

L'entrée de la simulation est décrite par un langage de programmation standard de haut niveau : le nouveau MPEG-4 Structured Audio Orchestra Language (SAOL) ; en principe, il n'est pas exclu qu'aucune traduction soit nécessaire pour l'application, si celle-ci est déjà disponible dans ce format. En plus, la simulation à travers un langage de haut niveau permet de tracer le comportement de l'architecture simulée en fonction du temps de l'application même, résultat qui est fondamental quand sa charge de travail est fortement variable comme dans des scénarios interactifs. Dans ces cas la complexité a toujours été considérée comme une chose à deviner.

Le nouveau modèle virtuel pour l'analyse de la complexité a amené deux résultats pratiques principaux :

- la méthode de simulation proposée a été utilisée pour définir les niveaux de complexité pour l'outil « Structured Audio » de la norme MPEG-4, et par conséquent l'analyseur, indépendant de la plate forme, est devenu le logiciel de référence pour le test de Conformance de MPEG-4 SA.
- un jeu d'instruction virtuel a été conçu ; il a permis l'implémentation d'un décodeur efficace pour SA : SAINT, basé sur un noyau virtuel de DSP interprété. La flexibilité du DSP virtuel de SAINT a permis un transfert rapide de son moteur d'exécution vers un DSP superscalaire VLIW, ce qui a fait de lui un des blocs composants de ThreeDSPACE, système pour la diffusion avancée de scènes Audio 3-D basées sur des descriptions MPEG-4.

L'outil complet pour la simulation d'architecture, qui contient un simulateur de mémoire cache, a montré des résultats prometteurs, obtenant des estimations du temps d'exécution de SAINT avec une approximation de 10% en moyenne pour des processeurs general purpose et de 20% pour un processeur superscalaire VLIW très complexe. Les programmes estimés ont parfois des variations d'un facteur 10 dans leur complexité le long de l'axe du temps.

En général, les résultats expérimentaux peuvent être considérés comparables à ceux des plus récents simulateurs décrits dans la littérature. Une limite du langage utilisé pour la modélisation des applications est que sa généralité est limitée à des calculs à une dimension en virgule flottante ; le plus grand avantage est que la simulation en fonction du temps de la performance est très facile et donne des résultats dépendants du temps qui sont fondamentaux pour l'optimisation des applications en temps-réel. Avec l'outil développé dans ce travail de doctorat des programmes complexes peuvent être modélisés rapidement grâce aux bibliothèques spécifiques de SA et aussi analysés rapidement ; de plus, l'outil peut être facilement étendu pour devenir un outil de génération automatique et de configuration des blocs de base principaux du système qui réalise les applications considérées.

En conclusion, le modèle proposé est pensé comme une approche alternative pour coordonner deux côtés : un des objectifs principaux de ce travail à été de concevoir une méthode d'analyse systématique qui puisse être utile en même temps au programmeur de logiciels et à l'ingénieur des systèmes hardware. Le premier peut bénéficier d'un outil de haut niveau reconfigurable et spécifique capable de analyser le profil des programmes de façon simple et indépendante de la plate forme et de simuler facilement, avec des marges d'erreur, le comportement d'une plate forme existante ou virtuelle ; le deuxième peut exploiter des estimations de complexité en forme abstraite, avec la possibilité d'étudier les applications envisagées dans ses divers aspects (opérations, utilisation de la mémoire, flux de données entre les différents blocs du programme) et avec la potentialité d'étendre ces résultats vers la génération automatique d'architectures de système.

ABSTRACT

Research in multimedia for consumer electronics is dominated by the problem of incredibly short times-to-market, that means fast complexity estimations and fast design of new architectures. On one side more and more sophisticated and flexible applications are rapidly developed, on the other side the exponential growth in IC computational power seems to be hardly capable to keep pace with requirements for real-time applications, since their complexity is exponentially growing as well.

The processor's performance, often slowed down by bottlenecks in memories and buses, is further reduced by the time wasted in communication among the several application layers. From this problem comes the conception of integrated development frameworks for simulation and design.

Tools for simulation and analysis of architectures have appeared from academic and industrial research laboratories above all in the last decade. Many of them are conceived to provide low-level exact simulation of the supported devices, at the price of heavy slowdowns in simulation times and huge sizes of traced data reports. Some others, in the last years, introduced some degrees of approximation in simulations, in order to speed up execution time and to increase the flexibility of the tools to support multi-processors. The resulting more or less abstract models are anyway not suitable to analyze real multimedia-oriented applications, where programs are usually available in some languages including dedicated libraries and meaningful results are only those measured in function of time.

On another level, tools for hardware/software codesign or for block-based system design provide more useful results, but to be effective they must rely on rigid cores that may allow only a few degrees of reconfigurability; some very recent tools are conceived to design complex systems by modeling blocks through a high level description language.

Conversely, the virtual model and related tool proposed in this dissertation have their roots in an approach to the analysis of complexity that aims to be, as far as possible, platform independent. The method is based on the concepts of abstract classes of operations and simulation in function of the performance time.

The work described in this dissertation finds its application field in the world of multimedia, more precisely in multimedia-oriented Audio applications. Media applications are commonly programmed by imperative or object-oriented languages, which are composed by many different statements, operators and above all standard libraries. A careful profiling of typical applications permits to detect fundamental operations and functions and to define a virtual instruction set, grouping more or less similar operators and breaking functions into basic building blocks. The resulting virtual instruction set does not correspond to any actual one but it has as property to be easily mapped on a large number of existing ones. The simulation of an architecture requires then the availability of measures, benchmarks or estimations of at least one member of each abstract class. The number of classes, i.e. the complexity vector, can be adapted in length and detail to the needed degree of precision and to the available set of actual measures and/or benchmarks.

The input to the simulation is described by a high-level standardized programming language, the new MPEG-4 Structured Audio Orchestra Language (SAOL); in principle, it may also be the case that the application does not need any translation, if it is already available in this format. Moreover, simulation through a high-level language permits to trace the behavior of the target architecture in function of the internal time of the application itself, result that is fundamental when the related workload is highly variable as in downloadable and/or interactive scenarios. In such cases complexity has always been considered a guess.

The new virtual model for analyses of complexity led to two main practical results:

- the proposed method of simulation has been used to define complexity levels for Structured Audio in the MPEG-4 Standard, and consequently the platform independent analyzer has become the MPEG-4 reference software for MPEG-4 SA Conformance test.
- a virtual instruction set has been conceived that has permitted the implementation of an efficient Structured Audio decoder, SAINT, based on a virtual interpreted DSP core. The flexibility of the SAINT virtual DSP approach has permitted a fast porting of its execution engine on a superscalar VLIW DSP, making it one of the building blocks of the ThreeDSPACE system, a framework for advanced rendering of 3-D Audio scenes based on MPEG-4 descriptions.

The complete tool for simulation of architectures, including a cache simulator, has shown promising results, achieving estimations of the execution time of SAINT with an approximation of the 10% in the mean for a general purpose processor, and of the 20% for a very complex superscalar VLIW processor. Estimated programs have sometimes dynamic excursions of a factor 10 in their complexity along the time axis.

In general, the experimental results can be considered in line with those of the most recent, state-of-the-art simulators presented in literature. A limitation of the

language adopted to model the applications is that its generality is limited to one-dimensional floating-point computation; the most relevant advantage is that simulation in function of the performance time is straightforward and provides the time-dependent results that are fundamental for the optimization of real-time applications. With the tool developed in this PhD work, complex programs can be quickly modeled thanks to SA specific libraries and also quickly analyzed; moreover, the tool can be easily extended to become a tool for automatic generation and configuration of the main building blocks of a system running the application, or the class of applications, under consideration.

The proposed model is finally intended as an alternative approach to coordinate two sides; a principal goal of this work was to conceive and specify a systematic analysis method that can be useful to both the software programmer and to the hardware system engineer. The former can benefit of a reconfigurable and dedicated high-level software tool able to profile programs in a simple and platform independent manner and to easily simulate, with some margins of error, the behavior of a specific platform, existing or virtual; the latter can exploit complexity estimations in an abstract format, with the possibility to study the target application in its several aspects (operations, memory usage, data flows among the different program blocks) and the potentiality to extend these results to an automatic generation of high-level system architectures.

CHAPTER 1. INTRODUCTION

1. Contemporary Audio and Multimedia Systems

It is undeniable that the last three decades of the 20th century have witnessed one of the most impressive revolutions in the history of mankind, and this revolution is essentially due to computers. In fact, writers, journalists, researchers and economists define more and more often our age as the "Computer Age".

It is well known that Integrated Circuits (IC) and silicon technology has been experiencing a period of exponential progress (the famous *Moore's law*); in a still recent past hardware designers were challenging their knowledge to build digital signal processors powerful enough to meet the requirements of simple voice or musical applications; nowadays programmable and even general purpose devices are available that are able to decode video sequences in real-time.

The fast shrinking of transistor size in IC production influenced in a dramatic way all basic components of a computer architecture, and above all processors (in functionality and speed) and memories (in size). Many different, relatively low-cost computer systems are today available, characterized by impressive claimed performance; even if many factors, buses and memory speed in first place, did not experiment a similar improvement, peak performances for PCs and workstations reached the target of billions of operations per second and this trend is expected to continue [1]. The increased capability in signal processing and manipulation of the information encouraged the migration of most applications from an analog to a digital or mixed analog-digital format. Ease in editing data and lack of degradation in typical conditions of transmission and/or storage have been fundamental for this migration, of which the Audio world is being one of the most relevant witnesses.

It is under everybody's very eyes that music, telephony and sound processing in general, are being revolutioned in the last times. Even theaters and interpreters of classical music are changing their way of working, to exploit the benefits of the easy manipulation of data that computing devices and information technology provide them [2]. Avant-garde and popular musicians, naturally predisposed to innovation, moved in a career time from the first analog synthesizers able to produce new timbres and sounds, to digital programmable synthesizers and sequencers, until software tools capable of replacing all in once sound generation, digital mixing and post-production and of obtaining all of them from a single environment for "content creation". The last step in this evolution is represented by programming languages specific for digital software sound synthesis (SWSS) and software digital signal processing: sounds can be stored and transmitted by their structured description instead than by their physical signal.

During the last four-five years, over which the work described in this dissertation has spread, three main concepts, out of the world of computer research, invaded the market of consumer electronics: multimedia, real-time, (communication and) the

Internet. Practical support of these three concepts has been considered a fundamental issue throughout the whole work; they have been mapped to the Audio domain to continuously redefine the research scope where to exploit the described methods and results through the concept of suitable tools.

It is perhaps interesting to remember, to sustain the above position, that until a few years ago computing devices were conceived and designed in a majority of cases to work in an isolated environment, in offline machines or special-purpose dedicated boxes such as, for example, calculators, electronic instruments and processing boards. Today almost no digital device is acceptable if it does not have some support for inter-processor communication and/or standard communication protocols to permit interactive data exchange outside the host machine itself.

Recently research focused on power consumption too. Many devices are now available that requires an energy several order of magnitude lower than in the past for a similar purpose, and this made possible the birth of a new computing domain: personal mobile computing. It is expected that in the near future personal computing and communication devices will be mainly portable and battery operated, they will support multimedia functions and will be connected through wireless infrastructures [3]; consequently, mobile computing issues have also not been forgotten in the exploitation plan of this work.

2. Scope and Goals of the Work

In the described exponentially growing context, it is a continuous challenge for the designer (and even for the normal user) to keep pace with the several innovations to find the best solution for his needs; it is not easy for an engineer to cope at the same time with enough generality in his approach and with specialization in his own field. The systems to be designed increase continuously in complexity in both hardware and, consequently, software; computing and embedded systems can be composed of several processing units in fast communication among them through high speed buses, and moreover IC technology already permits to implement the complete system on a single chip (System on a Chip, SoC) [1]; the single author or the small group has been replaced by big research teams, since hundreds of engineers are necessary to design a new processor from scratch [3]. Everybody in the team deals with his "small" piece and sometimes he even does not have a global vision of the project on which he is working and that is growing in parallel in its different parts. As a consequence, there is a strong natural tendency, from hardware design to programming science, to conceive an application in "layers"; the development and diffusion of sophisticated CAD tools, compilers and simulators makes this feasible.

Experience demonstrates that this approach often brings relevant drawbacks in terms of performance and stability of the whole system. Efficient communication is required among the different layers, and the higher is the level of abstraction in describing applications the slower will be the reaction of the entire system.

A classical example comes from programming languages. Assembly language is recognized as the best way to obtain the optimal performance from a programmable device; it is nearly abandoned because of the complexity of instruction sets, due to

the parallelism of several operating units in the devices of the last generation. High level programming languages are now predominant, and further levels of abstraction has been introduced by object-oriented languages and high-level description languages [4], with remarkable practical slowdown (a meaningful and well-known example of this slowdown is given by e.g. protocol interfaces for operating systems that are opaque to the hardware layer).

2.1. Difficulties in the Design of Complex Media Applications and Systems

In such a situation, it is extremely difficult for the application designer or the content provider to efficiently choose the cost-effective target platform, because the performance of a device depends often on application protocol interfaces (APIs), on the operating system, on the low level software (drivers, low-level protocols, etc.) and on the characteristics of the hardware architecture. On the other side the system designer is often not able to correctly evaluate the workload that will be required by content and application providers for typical and worst-case situations, because it is not easy to evaluate, in a reasonable time, complexity for a wide range of programs, processing requirements and available hardware options (custom, semi-custom, integrated board, completely programmable device).

To efficiently match hardware and software layers of such applications it is not possible to separate anymore the concept and design of the different parts of a complete system, or at least to neglect the precise knowledge of all the key issues involved in the complete implementation; at this point it is fundamental to have a consistent estimation of the overall complexity. To improve and enhance toward this direction the first generation of CAD tools for VLSI design, solutions to these problems came from the remarkable development of research fields such as architecture simulation, instruction-set simulation and hardware/software codesign. The latter approach [5] is based on the contemporary design and test of hardware and software, through an optimal partition of the application over the two. This design methodology assumes that the whole system is being implemented at the same time, while in many cases hardware is available and only software must be developed, or on the other hand a high level, software oriented standard reference description is given, and specialized hardware must be just chosen or reconfigured. To face these problems more pragmatically, another, more recent trend, that targets consumer electronics and communication, is instead based on the idea to provide pre-manufactured structures to designers, which can map software on those structures, in a kind of block-based system design [6] [7]. This methodology is also better suited to small-to-medium companies that cannot afford the development of complete systems from scratch.

2.2. Goals of the Work

The work presented in this dissertation is nearer to this second, block-based trend in what concerns design of multimedia systems; but at the same time it is intended as an alternative approach to coordinate the two sides, the software application designer and the high-level hardware designer. The main goal of the work was to

conceive and specify a model for systematic analysis of Audio and Multimedia algorithms and real-time programs, the use of which could be useful in its results to both the programmer of software applications and to the system engineer. The first can benefit of having a reconfigurable and dedicated low-level software tool able to profile programs in a simple and platform independent manner and to easily simulate, with some margins of error, the behavior of a specific platform, existing or theoretical. The second can find useful the approach of having complexity estimations in a virtual format, with the possibility to study the target application in its several aspects (operations, memory usage, data flows among the different program blocks) and the potentiality to exploit these results for an automatic design of system architectures. It may also be the case that all this can even be done without any translation or rewriting of the application code, since the modeling language, on which the whole analysis and simulation is based, is a high level standardized programming language. The proposed model does not imply that the application providers must work concurrently to the chip developers, even if this can be more profitable; in fact, the model is virtual and flexible and can be configured to different platforms and solutions.

The concept of the proposed method and corresponding tool was inserted into the activity for the standardization process of MPEG phase 4 (MPEG-4); this gave at the same time two opportunities: 1) to define and implement a method having as reference the state-of-the-art in multimedia programming and coding, 2) to contribute with the same method at the estimation of complexity of the concerned parts of the standard, namely Structured Audio and AudioBIFS (see chapter 3). In fact, the development went on even further and the proposed tool has been subsequently used, in a subset of its functionality, as official tool for MPEG-4 Structured Audio Conformance¹ test.

The major achievement of this activity, which lasted for a period of three years covering two different projects, is that it has demonstrated how a high-level (at a higher level than machine instructions) systematic analysis of some classes of algorithms and programs can yield at the same time an efficient implementation of the algorithms and programs themselves and a flexible simulation environment for generic programs belonging to the same classes. This environment is reconfigurable, in order to support extensions and new features, and can be used to provide results for the design of new architectures, either software or hardware or mixed software/hardware. This is perfectly in line with the most recent trends in information technology and silicon technology, especially in those fields of application where fast design reshaping is necessary to meet extremely short times to market.

¹ Conformance is that part of a standard that takes care of defining test tools and procedures to verify the quality of service of a specific decoder that is claiming compatibility with the standard itself.

2.3. Contributions

The work described in this dissertation ranges over different areas of research: it is not possible to limit it into a single one. The major contributions are in the following three fields:

- software engineering for multimedia: the complexity analysis of typical programs in a high-level language permits to implement optimized compilers and interpreters. A new virtual DSP, characterized by a vectorial and specialized instruction set, is presented as main result. Its performances and portability allow to easily embed its core into more complex environments.
- computer simulation and design: the MPEG-4 decoder developed as application of the virtual DSP has been enhanced to perform as a high-level, macro-oriented, very fast computer simulator, with the possibility to use results to immediately and automatically configure block-based design of architectures and their sub-block components.
- programming with special emphasis on computer music and digital signal processing: a new and systematical approach to complexity estimation of programming languages is described in this work, based on the high-level language itself and on macro-oriented criteria.

Besides that, many other minor contributions are related to the following topics:

- real-time software digital signal processing
- 3-D Audio algorithms
- design of compilers and interpreters
- virtual machines
- international standardization process
- conformance test
- data coding and compression

3. Organization of the Dissertation

This dissertation is mainly divided into three parts: 1) an introductory overview of the state-of-the-art in computer simulation and multimedia standard coding; 2) a theoretical part where the virtual model for analyses of complexity is described; as an MPEG-related application case, the SAINT (Structured Audio INTerpreter) decoder is presented as a part of the ThreeDSPACE project; 3) a final part where the model for analysis and simulation is extended from instructions only to a more complete and detailed computer architecture simulation.

The first part is divided into two chapters. Chapter 2 is an overview of the state-of-the-art in computer design and simulation, with an emphasis to those devices particularly used for multimedia, i.e. general purpose processors and DSPs. Chapter 3 is an introduction to some fundamental concepts of the multimedia world that will be necessary for the comprehension of the rest of the work; in particular, basic concepts will be introduced of structured audio coding, of software Audio synthesis and processing languages and of the new features of the MPEG-4 Audio International Standard.

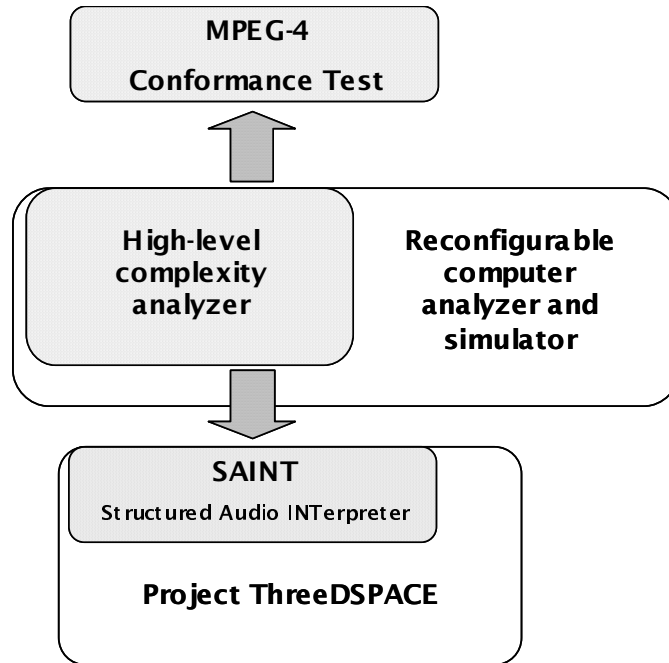


Figure 1. Relationship among the different parts of the work described in this dissertation

With chapter 4 the second part of the dissertation begins; this first chapter describes the new virtual model that has been conceived for the analysis of programs and applications, based on the definition of abstract classes of operations and platform-independent simulations. The analysis is oriented towards real-time multimedia applications and consequently it is characterized by a strong relationship with time. As a case study, the Conformance test procedure for MPEG-4 Structured Audio will be described. Chapter 5 will show how such a systematical approach to the analysis of a language can lead to a very efficient implementation of compilers and interpreters to better communicate with lower layers of the system. The concept and design of SAINT will be described; SAINT is based on a virtual DSP with a specific vectorial instruction set. Finally, Chapter 6 will conclude the second part with a presentation of the project ThreeDSPACE, which largely benefits of the SAINT approach to MPEG-4 decoding and adds to it a new powerful technology for 3-D Audio rendering.

Chapter 7 constitutes the third part; first, it is shown how the model for the analysis of complexity has been extended to a macro-oriented approach to computer simulation. Secondly, experimental results are reported that demonstrate a good accuracy in simulation, despite the approximating assumptions of the method. In the end, potential application to different programming languages is discussed, together with the possibility to enhance the proposed tool to a framework for automatic high-level design of architectures.

Chapter 8 concludes the report with hints to future work and projects based on SAINT and the derived simulator.

CHAPTER 2. PROCESSING UNITS FOR MULTIMEDIA: SIMULATION AND DESIGN

This chapter is intended as a brief survey of the state-of-the-art in simulation and design of processors. More precisely, since this dissertation proposes a simulation model and related methods mainly for the multimedia area, the overview will be limited to those devices commonly used in this field, namely microprocessors and DSPs, related with media consumer and professional electronics. After a short introduction on current design practice for hardware devices, some considerations will be also made about complexity and requirements for software. In fact, it is not correct to divide analyses of hardware from those of software, since it is clear that the two evolve dependently; on the other hand, an effort in this sense is at the same time useful, to separate and recognize the many different contributions to the overall application: these two sections will be indeed quite related to each other. The most relevant part of the chapter is then dedicated to two surveys of, respectively, software simulation for complex computer systems and general directions in design and verification of very large ICs, especially those known as Systems on a Chip (SoC). The main techniques and tools will be compared and shortly discussed.

1. Media Processing: Trends of General Purpose Processors and DSPs

It is not difficult to recognize the field of media-oriented electronic devices for consumer and professional markets as one of the most hectic areas of research nowadays. It is enough to navigate with a browser on the Internet or to read a computer magazine to notice almost every week that new processors with improved potentiality are disclosed by vendors, that new technologies are announced by many different research groups, well-established or still in search of funding, that enhanced standard protocols and languages for data transfer and connection to fast peripherals are proposed by consortia of the main companies of the sector. In spite of that, real innovations are not so much frequent, most often the enhancement only comes from increasingly sophisticated silicon plants, whose equipments are able to improve at a constant and programmed pace their technology.

In this ferment, some general trends can be nevertheless identified: they characterize many of the last generation devices for media applications, and they will probably characterize the evolution of the next decade [8].

1.1. The Current Generation of Processing Units

During the '90s, many developers realized that, as they were trying to completely port in software functionality previously left to obsolete versions of accelerator cards, the number of tasks to be committed to computers increased exponentially; it was no more possible to imagine inside computing machines "the" completely

general-purpose processor able to efficiently support the characteristics of so many kinds of programs, often coming out from dedicated programming languages and/or compilers. This is why they began soon to provide specific devices able to foster processing routines required by hit programs, games and network tools, such as advanced graphical interfaces for simulators and virtual reality, supports for interaction, sound and image decoders and processors, handlers for network protocols and information packets. In a short time the major chip vendors disclosed enhanced versions of their top-line processors with the so-called "multimedia extensions" [9][10], with alternate results and success. The main characteristics of those extensions are introduced below.

1.1.1. Data-Level Parallelism

It is well-known that multimedia and coding algorithms are characterized by DSP-like requirements; a few, dedicated, small loops are cyclically repeated over streaming or stored data. In such a situation it appears much easier to rely on the massive data-level parallelism than to persist on fine-coarse instruction level parallelism that years of research proved hard to detect and exploit [11]. Proposed extensions for microprocessors are normally based on typical solutions of parallel computing like SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) architectures; V-LIW (Very Long Instruction Word [11],[12]) can be considered as a single processor evolution of MIMD; as technology improves, the latter is becoming the most followed approach to parallelism (see [14] and [15] among many others as examples).

1.1.2. Data Transfer Bandwidth

A simple increase in processing power is not enough if it is not supported by an adequate data fetching from external devices to processing units. High transfer bandwidths are required in multimedia systems, for both stored (memories) and streaming (network, I/O ports) data. Extended instruction sets were enhanced soon by designers to provide a better data management, particularly to increase memory access and communication bandwidth [13].

1.1.3. On-chip Cache Memories

A third characteristic that is imposed by media applications is a dramatic increase in size of fast on-chip cache memories; technology permits to integrate them at exponentially rising size. This necessity for caches comes from two main factors: first of all the aforementioned necessity to increase memory throughput and data exchange among temporary buffers; secondly the obligation to effectively store large buffers required by e.g. coding standards such as MPEG-1 and MPEG-2 or graphical processing.

1.2. Multimedia Processors

In parallel to this tendency to include DSP-like features in general purpose RISC/CISC processors, some companies explored the opposite direction of designing new specialized DSP architectures for multimedia, i.e. DSP/CPU processing units

surrounded by hardware modules for special tasks (particularly I/O formatting and video processing), with fast connections and remarkable on-chip data caches. The Philips TriMedia processor is probably one of the first examples of System on a Chip, conceived and optimized for MPEG-2 video decoding but at the same time able to support other media functionality like sound processing, sound synthesis or video processing. TriMedia will be better analyzed in Chapter 6.

Indeed, only this chip survived of the several of the same kind that were announced in the last years, revealing that, without a smashing-hit application requiring a low-price device, the relatively high-cost solution of the general-purpose processor has been so far the most appreciated by the market. Moreover, it is objectively true that progress in solutions and performance for processors have been much more striking than for DSPs.

It does not appear correct, anyway, to say that design of specialized media DSP will become extinct, as many foresee [8]; it is more correct to say that embedding a DSP into a generic RISC processor architecture has revealed to be more effective than its dual solution. In addition, at least for some time still, this is true only when there is not any concern in terms of power consumption and/or very low price target.

In fact, as already hinted in the first chapter, a new factor arrived to further stimulate technology, i.e. the rising of the area called personal mobile computing, pushed by enhancements in both portable devices and wireless communications. Pioneer solutions in low-power architectures like, among others, StrongARM [17] and TinyRISC [18] have quickly become the reference for a new generation of devices, where peak performance is no more the predominant factor and instead low-price and low-power are quite important; while in a desktop computer, in presence of e.g. a conventional CRT or liquid-crystal color display, proportional price and power consumption offer some degrees of freedom to designers, this is no more true for small mobile devices, where lifetime is a fundamental feature.

1.3. Future Trends of Processing Units

Analyzing and extrapolating all these requirements and results, many researchers are trying to define future trends of microprocessors; it is evident from related literature that it is not possible to isolate a single, defined direction in architecture design. As probably it does not exist a general-purpose application for a general-purpose processor, the same it is impossible to define a general multimedia application for an all-in-one enhanced multimedia processor (see next section). Rather, some different architectural solutions recently proposed in literature for the next years are reported in the following points; they are mainly conceived to satisfy fundamental characteristics introduced in subsections 1.1.1 to 1.1.3.

- advanced superscalar processors: in these devices the current tendency to parallelism will be pushed further until reaching parallel execution of 16 instructions per clock-cycle or more; many functional units will run in parallel, supported by enormous memories (70% of the chip size or more, for cache and other register memories) providing the necessary bandwidth; aggressive branch predictors and predication techniques will be also necessary to exploit such a parallelism [19].

- vector IRAM processors: these devices will add to vector processing large, on-chip DRAM banks, which provide the vector units with enough bandwidth. The strong point with this approach is that DRAM can store much more data than conventional cache; being it on-chip can provide a wide connection at low power consumption, supplying by that to inferior memory speed [22].
- single-chip multiprocessors: in these devices, some distinct processors (4 to 16) will run in parallel one or multiple independent tasks. Every processor will stay simpler and single-thread, in a try to exploit parallelism at a very high level, i.e. the thread level. The key issue will be here fast on-chip communications among the many processors, which will require relatively low design effort [20] [21].
- raw machines: these devices implement highly parallel architectures, potentially with hundreds of "tiles", i.e. very simple processors with some reconfigurable logic. Execution and communication among tiles, that is the overall functionality will be controlled almost entirely in software. In addition, like for some simple V-LIW architectures, all instruction scheduling and optimization tasks will be left to software compilers [23].

2. Multimedia Applications: Complexity and Portability

2.1. *Complexity Issues in the Development of Software*

The fast growing in transistor density and performances of ICs made, and still make, programmers believe than sooner or later "everything" may be implemented by writing efficient software and exploiting more and more efficient compilers. It is undeniable that this is in a certain sense true, because today a well-optimized code can make a good quality video decoder run in real-time on a PC or workstation, when only a decade ago this was achievable only by a dedicated hardware. Nevertheless, at the same time it is also a fact that many factors participate to make this task much more difficult than it seems; some of these factors are presented in the points below.

- Complexity of applications is growing almost as fast as the increase in processing power of processors. There is a natural tendency to imagine more complex applications as soon as enough processing power is foreseen for the near future.
- Increasing complexity in software applications generates the already mentioned phenomenon of abstraction layers: a multimedia application is normally written in some high-level language, it relies on lower-level drivers and operating systems-related APIs, it provides interaction and is often configurable or programmable by some higher level scripting language. Efficient communication among all these layers is required.
- Increasing complexity of development tools (for instance, two popular C++ compilers for PC like Borland and MS Visual provides about 200 MB of libraries) together with increasing abstraction and short time to market concur to force the programmer to write software that can be far from being optimized, at least in its first releases.

- The concurrent presence in the same processor of different processes, above all if network-related, consistently congest the sharing of resources like RAM memory and system buses, which are known as being among the bottlenecks of modern systems because of their inferior speed.
- The rise and growth of personal mobile computing and the consequent security needs for network transactions renew the search for optimized, specific hardware solutions, in spite of processors that are able of billions of operations but at a consumption of tens of watts and running at best with non-secure systems.

2.2. *Media Software and Portability*

In addition to problems of complexity like those reported in the previous section, the contemporary availability on the market of different operating systems and processors, and above all the very high number of multimedia-supporting integrated boards and network boards, creates serious problems of compatibility and portability of a software toolset on the different configurations; a tool being optimized for, say, a desktop personal computer is far from being such even for a UNIX workstation, not to say of e.g. a video-game station. The problem became even more serious once the Internet made possible the download of small applications directly from a web site.

A solution that rapidly gained the favor of the developers is JAVA [24]. JAVA is an object-oriented language conceived for portability, security and compactness. A JAVA virtual machine, which is a software, interpreted one, is superimposed to the machine supposed to physically run the application, so that the virtual "executable" can be kept the same for every platform. What makes the success of JAVA is its robustness (above all automatic garbage collection), the security provided by dedicated methods and by the absence of pointers and the status of freeware under which Sun Microsystems distributes the specification and the Software Development Kit for several platforms. On the other hand, the interpreted virtual machine introduces a considerable drawback in terms of performance, unless specific optimizations are designed, as it happens for the personal computer platforms [4].

A recent enhancement to the JAVA technology comes from the JINI technology, always by Sun Microsystems [25]. JINI is a further effort on the roadmap to unify communication among different distributed devices; its goal is to provide a common infrastructure able to indistinctly connect to the system any kind of peripheral or component in a transparent way.

Many other efforts are being done for platform independency, but most of them are attempts to force proprietary de-facto standards for a single type of machine. Another way to afford the problem of compatibility leaving instead way to competition is that pursued by standardization organizations. This issue in the field of multimedia and audio in particular will be discussed in the next chapter.

2.3. *Remarks on Hardware/Software Partitioning*

Some conclusions are finally possible, as all the considerations about software are interlaced with remarks made in the previous section about hardware.

First of all it appears that, if theoretically many things may be implemented in pure software on powerful processors, this is still not practical in many cases because the concurrent presence of many threads and consequent resource sharing can slow down several times performance of fast devices. Moreover the communication between different pieces of software is not always optimal, because of the different hardware configurations that could be available and of the continuous merging of programs with third party libraries and tools: this leads often to a certain "weakness" of software in comparison to more expensive but more robust, custom hardware solutions.

In second place, as it will be better shown in the following, the production of hardware devices has always come with the idea of long times to market, while modern techniques based on CAD tools and powerful simulators can help to create well-evolved design flows that permit to yield new releases of processors in a few months. This does not mean that producing hardware is becoming cheap, but for those who can afford the price, an hardware solution in a custom or semi-custom design can lead to good results in terms of both time-to-market and flexibility. It is of course necessary a mass production to cover the costs.

Last, the problem of security for complete software solutions is much more serious than for hardware or mixed ones. It is understood that a secure approach in pure software is more expensive in terms of work and reduction of performances, and probably impossible to be completely solved without any kind of hardware support [26].

3. Simulation of Very Complex Devices for Multimedia Applications

3.1. *Simulation of Computing Devices*

As already stressed in the introduction, computing devices and systems have reached a complexity such that, considering the shortening of time to market, it is not possible to design a new processor from scratch without a massive investment and a group of hundreds of motivated engineers [27]. This encourages the successive modification of existing, well-established design flows and architectures, rather than totally new and specific concepts.

In any case, the design of present and future parallel architectures requires more and more rapid simulation of target designs running realistic workloads; at the early stages of a project, workloads are normally available in software, written as a code for different platforms or with a high-level language to be compiled. At the same time, an HDL description of the architecture under design is often not available yet, since only general architecture and instruction set are normally defined at this stage. This is why it is very important and useful to extend software simulation of actual machines, essentially used for software optimization, to software simulation of machines that do not exist yet, or at least that differ in some aspects from the actual one used for simulation.

Computer simulation has a long history inside computer science, but only recently, exactly for those needs in fast prototyping of complex hardware systems, it has evolved towards a more extended functionality.

The main concept behind simulators is the same that is used for profilers, i.e. code instrumentation: once the program is available in a software programming language, a tool automatically augments code in correspondence of specific instructions, so that it is possible to monitor execution while the program executes. A simple profiler has counters only to count instructions and function calls that happen during execution, often using a subset of the whole code, statistically selected, in order to reduce the simulation overhead. In such a way, only statistical global reports are available at the end of the execution.

A tracer is instead a more advanced tool, much more useful in the analysis, design and tuning of both hardware and software systems: a tracer is essentially a recorder, always through code instrumentation, of detailed information about the behavior of a program. Traces can be generated of accessed memory addresses, register usage, instruction combinations, instruction counts, etc., according to the precise goals of the tracing. When the traced program is written in a way that cannot be directly executed by the machine running the code, then the tracer is also called a simulator.

Finally, an analyzer is a tool that consumes informations generated by a tracer and uses them to predict the behavior of a program on a specific system. The feedback provided by analyzers can finally be exploited for improvement of a wide class of designs: architectures, compilers, applications. For instance, many analyzers have been implemented for cache and RAM memory simulations; they starts from a trace of memory accesses provided by a suitable tracing tool [28]. The boundary between simulators and analyzers is not strict in literature, since both tools are often integrated in a single framework.

Many features can be optimized in a tracer/analyzer to make it useful [29]; among them it should be made language and compiler independent, it should be reasonably fast, and, if it must be used for design, it should be able to trace programs on machines that are not yet available. According to the specific task, it is important to find the optimal trade-off among these requirements.

3.1.1. Simulation Techniques

Many simulation tools are presented and described in the literature of the last decade. All of them are based on one or more of three simulation techniques: interpretation, static compilation, dynamic compilation.

Interpretation based simulation allocates in memory of the host machine (the one running simulation) a data structure representing the state of the target machine (the one to be simulated). A loop is then started, in which a sequence of fetch, decode, dispatch and execute is repeated for each instruction of the target machine code, updating at each iteration the state of registers and memory. Most commercially available simulators are interpretative. They are characterized by ease of implementation, but suffer of an often heavy slowdown, due to the loop to be executed for every target operation.

Compilation based approaches reduce the runtime overhead by cross-compiling target machine instructions to one or more host machine instructions, which update the simulated machine state. If the translation is done at compile time the overhead is completely eliminated and this is the case for static compilation simulators; instead, in the case of dynamic compilation translation is done at load time, and the overhead is reduced only by the repeated execution of loops of target machine code. In this second case, it is also possible to simulate dynamically linked code, while in static tools this is obviously not possible.

A common trade-off faced by all the simulation tools reviewed in literature is the one between the speed of the simulation and its accuracy. Many tools are able to modify the accuracy of the simulation in order to reduce to the simulation time to the minimum required for a certain goal. Another very important trade-off is the one between the portability of the language and the precision of the simulation. Receiving a high-level language input can give the best portability and best optimization, but the tool can be used only with programs written in that language, where some machine details could be not accessible. Assembly code is less portable but can provide more detailed information. Using instead executable code as input no longer requires access to the (possibly unavailable) source code, but resulting information may be obscure, being in fact without any kind of symbolic translation. The main characteristics of some interesting tools are reported in Table 1.

Table 1 Main features of some computer and system simulators presented in literature

Simulator	Level	Input	Method - Notes	Slowdown
WWTII	user	exe	static cc + interpreter	25-166
SHADE	user	exe	dynamic cross-compiler	8-15
Embra	user	asm	dynamic cross-compiler	3-20
SimOS	syst	asm	dynamic cross-compiler + interpreter	2-500
SimICS	syst	asm	interpreter	25-75
Mermaid	user	C-aug	abstract machine	60-750
Zhu/Gajski	user	hll	cross-compiler + vm + static mapping	1.1-2.5

3.2. Overview of Some Instruction Set and Computer Simulators

Oldest simulators in literature are essentially instruction-set simulators, i.e. they support simulation of instruction sets different from that of the host machine. A single process at a time on uniprocessor or parallel machines is normally supported, and then they are not able to simulate interaction between the process and the operating system. Input to the tool is most often an executable.

Wisconsin Wind Tunnel II [31] is a fast simulator that is based on the possibility to directly execute code of the host machine and to simulate by an interpreted approach communication among the possibly several processors; WWT aims at simulating a multiprocessor executable on a uniprocessor, and in this sense it is not reconfigurable without a translation of the whole implementation. WWT introduces anyway consistent research on the optimal way to introduce augmented code for optimal profiling [32].

Some limitations in WWT are removed by SHADE [33],[34] and by Embra [35], which are true instruction-set simulators based on dynamic cross-compilation. SHADE in particular is able to cache cross-compiled code for overhead amortization; moreover, it allows changing dynamically traced information, according to the needs of the analyzer that are provided through special-purpose code. In both SHADE and Embra target and host instruction sets can differ; reported experiments are anyway among devices of the same family, or for which instructions can be easily translated from one to another.

3.2.1. *SimOS and SimICS*

SimOS [36],[37] and SimICS [38] are two very similar tools in conception and goals. They aim at extending the simulation of a single process to the simulation of a complete system, including multiple processes and the operating system. In both of them a commercial operating system can be booted on the simulated hardware, so that it is possible to investigate realistic workloads. They have the ability to dynamically change levels of detail for the simulation, allowing slowing down the simulation with a detailed tracing only when it is desired. The two tools are targeted at a software analysis and its interaction with the key hardware components. They use an interpreted approach, which better permits to model different kinds of processors and hardware configurations; in this sense SimOS is more flexible, thanks to its configurable semantics for the classification of hardware events. SimICS is more rigid on supported hardware, and then more performing.

3.2.2. *Mermaid*

Mermaid [39] is a first meaningful example of simulator conceived to support levels of abstractions, i.e. simulations take place at a level of abstract machine instructions. This approach results in a higher simulation performance at the cost of a small loss in accuracy. The Mermaid simulator is driven by abstract instructions, called "operations", representing processor activity, memory I/O and communication; of course, in this case low-level simulation of e.g. the processor pipelines is not possible, due to the lack of register specification in the operations. Two models are used, one for single node computation and another for multinode communication; each node defines its own set of operations. Using this approach is not possible to interpret directly machine instructions, and then the application must be explicitly modeled, i.e. programmed.

In the Mermaid project, to guarantee architecture independence at the application level, it was decided to directly augment C code in which applications are often programmed. An annotation can be traced that follows the memory of the program, the computational behavior and communications.

This method permits to stay at the same time independent from the target machine, which is an abstract "c-machine" (c language is its instruction set) and from the host machine, with the only constraint to have a system for which it does exist an available source of a C compiler that allows instrumentation of the code.

The major limitation of this method, which provides simulation accuracy of 2-5 % in the mean and 4-30 % in worst case, is that it cannot be used to simulate another

instruction set than C, unless a cross-compiler from this other set to C is used. However, a workload can be estimated for several different architectures at a very low cost.

3.2.3. *Zhu/Gajski*

A new ultra-fast instruction set simulator has been recently proposed by Zhu and Gajski [40]; this simulator is designed with two main goals in mind: simulation speed and retargetability, i.e. how easily the tool can be extended to handle new target machines and new host platforms.

Simulation speed is obtained using the fastest method, i.e. static cross-compilation of the target machine code. At the same time, retargetability is obtained cross-compiling to a RISC virtual machine, which has predefined instruction set and an unlimited number of virtual registers. The method requires then implementing the virtual instruction set on the host platform, and only this part need to be rebuilt in order to simulate on another host. The complete tool makes use of a retargetable compiler that generates virtual machine instructions; then a code generation interface, implemented on the host, transform each virtual instruction in a form that can be compiled into host machine instructions. In this case, slowdown is very limited, from 10% to 250% of the real execution time.

The tool aims mainly at codesign (see next section) and embedded systems; in this sense, the limitation on dynamic libraries is not so heavy to tolerate. Another limitation is that of the retargetable compiler, in the sense that it requires a parser for different target instruction sets and virtual sets.

4. Modern Trends of Design for Embedded Systems and Systems on a Chip

The field of computer simulation is an essential one to be able to evaluate modern systems without having to go down to real hardware. At the same time, its potential is limited to analysis, and it requires afterwards a precise specification and a complete design flow to arrive at the software and hardware implementation, based on such an analysis.

Increasingly fast times to market in some consumer electronics areas requires an even more aggressive design flow; moreover complexity of the systems under design is often so high that it could be useful to go through the complete design and simulation at a high level before any integration begins, in order to allow fast and cheap iterative refinements of the hardware and software partitioning. This explains the interest in fostering automation in the design process, until the capability to describe and simulate a complete system with a single integrated tool. In the following subsections, an overview will be given of some major design trends. The work related to this dissertation does not deal directly with VLSI design until its final potential extension, then this overview will be kept, for convenience, brief and forcedly incomplete. But indeed it is important to insert this as a background, to better understand the possibility of practical exploitations beyond the theoretical analysis.

4.1. Hardware/Software Codesign

4.1.1. Fundamentals of Hardware/Software Codesign

Software/Hardware codesign can be defined as the simultaneous design of both hardware and software to implement a desired functionality. Efficient codesign goes together with coverification, which is the simultaneous verification of both software and hardware and in what extent it fits into the desired function. Codesign and coverification have been recently inserted into the normal product development especially when products incorporate embedded systems or when large and complex Systems on a Chip are involved [42], [44].

Traditionally these systems were developed by decomposing an initial specification into hardware and software components, which were then developed separately. Integration was deferred until late in the design process. This early separation of concerns could make it difficult to explore the design space and could lead to problems when integration was finally attempted. Despite these drawbacks, this process has been used because of the gap in skills and approaches between the traditional worlds of software and hardware design.

By using simulation models, it is instead possible to find conflicts between top-down constraints, which come from design requirements and bottom-up constraints, which come from physical data. Basic codesign steps are described in Figure 2.

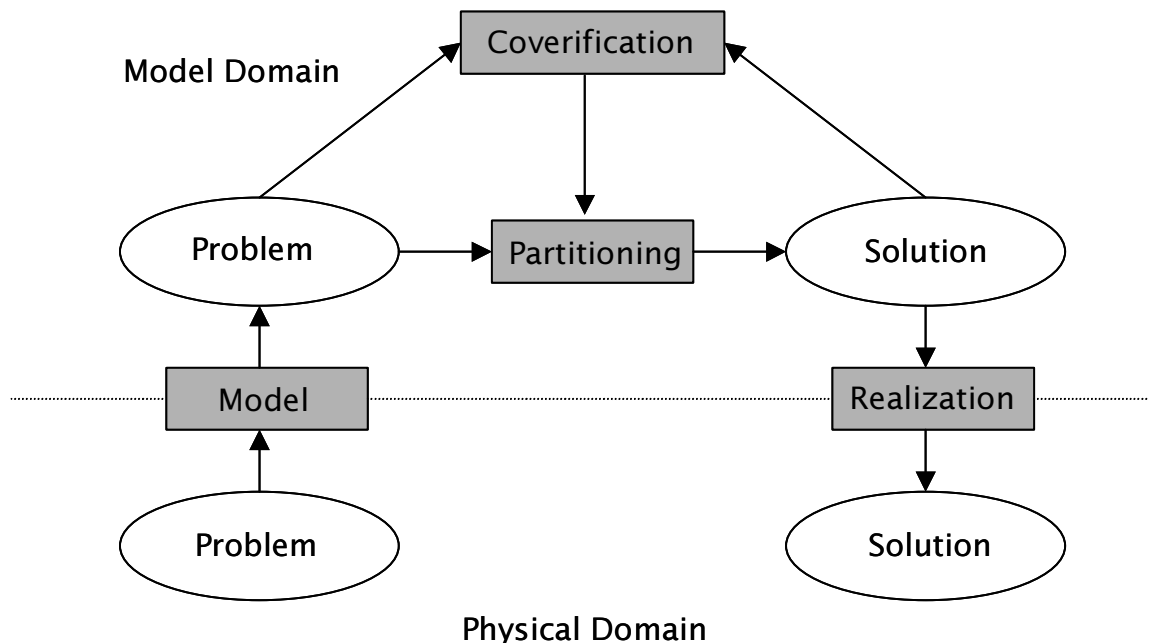


Figure 2 Basic steps for the codesign approach

It is often the case that hardware is available, so this can not be changed by software/hardware codesign. Only the software can be changed, and it should be fitted to this physical data. Therefore, a certain modeling strategy is necessary to cover the existing hardware. This modeling is not easy and it will never be perfectly adequate because the reality is too complex to find a perfect model.

It seems then easier to design both hardware and software, because it is often easier to design two things that have to work together, than design one thing and fit it around another. However, if both hardware and software have to be designed, powerful verification is essential because it is necessary to design two different "products" that interact with each other and nothing is "physical" on both "products". Of course, different techniques have been developed to verify combined hardware-software systems, but each of them has its own limitations, as it will be shown below.

Because there is no widely accepted methodology or tool available to drive designers to create a functional specification, manners mostly ad-hoc are used, heavily relying on informal and manual techniques and exploring only few possibilities. The main concern in such a methodology is precisely the specification of the system's functionality and the exploration of the system-level implementations [42].

When the steps in Figure 2 are successfully completed, embedded-system design methodology from specification to manufacturing is roughly defined. This hierarchical modeling methodology can provide a high speed in productivity, preserving consistency through all levels and thus avoiding unnecessary iterations, which makes the process more efficient and faster.

To describe a system's functionality, the functionality itself must be decomposed and relationships between the pieces should be described. There are many models to describe functionality of the system, and among them [5]:

1. Dataflow graph. A dataflow graph decomposes functionality into data-transforming activities and the dataflow between these activities.
2. Finite-State Machine (FSM). By this model, the system is represented as a set of states and a set of arcs that indicate transition of the system from one state to another because of certain occurring events.
3. Communicating Sequential Processes (CSP). This model decomposes the system into a set of concurrently executing processes, processes that execute program instructions sequentially.
4. Program-State Machine (PSM). This model combines FSM and CSP by permitting each state of a concurrent FSM to contain actions, described by program instructions.

Of course, each model has its own advantages and disadvantages, and the choice of the correct model introduce an additional delicate task into the process.

To specify functionality, several languages are commonly used by designers. VHDL and Verilog are very popular standards because of the easy description of a CSP model through their process and sequential-statement constructs. Of course, other languages are used as well, but normally they do not directly supports state transitions.

Codesign is still a very new field and researchers in this area have rapidly evolving interests. Work is in progress aiming at introducing more sophisticated algorithms and features on top of a basic framework as discussed above. Most of the implementation effort is devoted to cost/performance evaluation. Higher level of automation in optimization, direct user selection, analysis of data flow connectivity

and resource-analysis is currently researched. On the other hand, while some efforts can be made for partitioning algorithms and for their theoretical definition [42], in most available tools (see next subsection) this delicate part is left to the verification of a sequence of reasonable solutions. At this point, it is clear the practical limitation of the theoretical codesign approach; in most cases it is reduced in practice to a mix of good sense and powerful coverification.

4.1.2. Hardware/Software Codesign Tools

Among the academic research projects, the most important is probably Polis [43], [44]; it is a hardware/software codesign package by the UC Berkeley CAD group that uses Ptolemy as a framework. It aims at implementing unbiased specification and efficient automated synthesis for control intensive, reactive real-time systems. The Polis based design system is centered on a single Finite State Machine-like representation, called a Co-design Finite State Machine (CFSM).

Each element of a network of CFSMs describes a component of the system to be modeled. CFSMs are also a synthesizable and verifiable model, because many existing theories and tools for the FSM model can be easily adapted for CFSM [43]. In Polis, designers write their specifications in a high level language (e.g., ESTEREL, graphical FSMs, subsets of Verilog or VHDL) that can be directly translated into CFSMs. Any high level language with precise semantics based on extended FSMs can be used to model individual CFSMs. Polis addresses high-level language translation, formal verification, cosimulation, manual partitioning, and synthesis of hardware and software including the interfaces.

Among the commercial tools, for which it is always hard to find objective documentation, a well-known one is N2C by CoWare [46], a start-up from IMEC. N2C uses extended (to support timing information) C/C++ code as modeling language, thing that often speed up the functionality description; it is biased towards support of high level re-use of modules and it is able to generate VHDL/Verilog blocks.

In N2C communication models between hardware and software can be specified. After manual partitioning, device drivers and glue logic are generated to support the communication model. The design can be verified with untimed models, instruction accurate models, or bus cycle accurate models. It relies on an instruction set simulator and a hardware simulator. Its primary strength is support for Intellectual Property cores.

All the principal vendors of CAD tools provide a more or less codesign-oriented tool. For instance, one of the most comprehensive tools as for support of embedded cores for simulation is COSSAP from Synopsys [45]. COSSAP is a DSP design tool suite. It is used for algorithmic specification and verification to evaluate system concepts and partitioning. Implementation can be in C or HDL code. This tool set provides virtual processor models to execute software and is linked to a hardware simulation. It is advertised as a co-verification tool.

Finally, advanced tools are also provided by Cadence. The Alta tools (BONeS designer, plus other utilities) are targeted to high-level design aiming at SoC implementations. BONeS is a tool for design and analysis of system architectures, networks, and protocols. It is completed by a block-oriented design, simulation, and

implementation environment for electronic systems. These two high-level design tools offer a variety of library elements, protocol models, and utilities to assist in a system level design and evaluation.

4.2. Block-based System Design

Hardware/software codesign is a powerful mean if a designer wants to meet short times for a complex design. Anyway, at least scanning tools that are available on the market, this approach presents some limitations. Among them, as evident from the previous subsection, the difficulty to choose a suitable abstract model and the almost impossible task to define reliable partition algorithm to automate this fundamental task.

Another limitation that can be often hard to face for a designer is that most of the codesign tools are based, for coverification, on traditional RISC CPU core suppliers. These cores are normally "hard" cores, in the sense that they are supplied as they are, already optimized through RTL and subsequent place and route custom design. The design often is reduced to the definition of a system bus, in many cases with a standardized on-chip bus interface that can be directly used with embedded cores, to which the several processing units, memories and I/O blocks are attached, with some additional glue logic to control the whole system. The resulting design is verified by the relating software and an iteration process is repeated for progressive refinement.

In many cases, this procedure is neither satisfactory nor acceptable; such manufacturing restrictions have pushed both core and ASIC suppliers to develop "soft" processor versions. These versions are often supplied in the form of an RTL/HDL file that can be integrated into the rest of the SoC design. Of course, there are drawbacks in this approach too. Perhaps the worst is the issue of functionality once the core is synthesized and laid out: when the core is synthesized with the rest of the custom logic functionality, performances are not guaranteed anymore.

A new trend is now developing that may be very interesting for designers in search of performance and optimization; this new technology stream is currently introduced by ARC Cores, Tensilica and Improv Systems [7], [48]. These companies have developed tool suites and processor architectures that can easily be modified in their main features and component blocks almost by pointing and clicking on a graphical user interface. This new trend lies in between custom ASIC design and the adoption of generally reconfigurable hardware.

4.2.1. ARC Cores

ARC's solution is a basic 32-bit RISC processor architecture with a four-stage pipeline and a memory controller. Designers can add caches or a math coprocessor to the architecture; they also can modify the architecture by increasing or decreasing the number of register files, expanding or reducing the bus widths, adding new instructions, or deleting some commands. The processor includes basic instructions with additional variations. With them, designers can create application-specific instruction extensions. DSP hardware extensions include also a multiply-

accumulate block, X-Y memories, saturating add/subtract functions, and instruction-cache options.

Core performance totally depends on the process used to fabricate the chip and the complexity of the final SoC solution. The tool suite includes a software library of DSP functions, with a debug capability to handle multiple ARC cores in the same system.

4.2.2. Extensa

A high-performance 32-bit RISC processor also can be configured and reconfigured by the Extensa design system from Tensilica [48]. Available as a synthesizable core, this processor also is customizable by a graphical design tool suite. Targeted at deep-submicron processes, the Extensa processor contains about 25,000 gates, and employs a five-stage pipeline.

The base architecture consumes a very small section of the SoC. Even though it is more rigid in its ability to alter features, the processor does allow the addition of caches. It also supports on-chip coprocessors, a multiplier-accumulator, and many other I/O solutions.

4.2.3. Jazz PSA

Another slightly different but still interesting solution is proposed by Improv Systems with the Jazz Programmable System Architecture (PSA) [49]. A PSA chip can be configured after manufacture; it incorporates an VLIW technology core coupled with multiengine concurrency. The architecture is programmed or configured by defining its functionality in JAVA, using an Improv-defined structured JAVA methodology. Once more the main goal is to provide a pre-manufactured structure (multiple connected processing engines) and then map the software into that structure. A system developer can define the functionality and constraints of his application using the Improv's JAVA framework and then may use the provided compiler to convert this to instructions that can be loaded into the PSA chip.

Improv's compiler combines techniques from object-oriented systems analysis, high-level language compilers and hardware behavioral synthesis into a single executable program. Once the instructions are on the chip, the chip operates as an application-specific standard product (ASSP). In this way it is possible to combine programmability like in an embedded processor with FPGA-like reconfigurability.

CHAPTER 3. MULTIMEDIA APPLICATIONS: STANDARDS FOR THE TRANSMISSION OF INTERACTIVE SCENARIOS

After having reviewed the main trends in simulation and design of media-oriented processing units, it is now the moment to investigate in some more details the multimedia requirements; in particular the attention will be devoted to the MPEG-4 International Standard, expected to become the reference coding framework in this field, and then to one of its most interesting and innovative parts, the Structured Audio (SA) toolset and its Structured Audio Orchestra Language, a powerful mean to describe synthesis and processing algorithms.

1. The MPEG-4 International Standard

1.1. *General Overview*

The MPEG-4 International Standards has been released, in its first version, in 1998. A second, enhanced version has been released in 2000 and further versions for some of its parts are still under development.

The MPEG-4 standard provides a set of technologies to satisfy the needs of authors, service providers and end users [51].

For authors, MPEG-4 enables the production of content that has a good reusability and greater flexibility than possible with other individual technologies such as digital television, animated graphics, World Wide Web (WWW) pages and their extensions. In addition, it is possible to better manage and protect content owner rights through a flexible IPMP (Intellectual Property Management and Protection) interface.

For network service providers MPEG-4 offer transparent information, which can be interpreted and translated into the appropriate native messages of each network with the help of the related standards. The exact mapping for these translations are not in the scope of MPEG-4 and are left to be defined by network providers. Anyway, signaling of the required Quality of Service (QoS) information in the stream will enable transport optimization in heterogeneous networks.

For end users, MPEG-4 provides a large range of functionality, which could potentially be accessed on a single compact terminal with different levels of interaction with content, but always within the limits set by the author. MPEG-4 applications includes, among others, real-time communications, digital broadcasting and mobile multimedia.

1.1.1. What MPEG-4 Standardizes

The main goal for the development of MPEG-4 was to avoid the rise and spreading of a multitude of proprietary, non-interworking formats and players for multimedia content. To achieve this result, MPEG-4 provides standardized methods to [52]:

1. represent units of audio, visual or audiovisual content, called "audio/visual objects", or *AVOs*. These *AVOs* can be of natural or synthetic origin; this means that they could be recorded by a camera or by a microphone, or generated with a computer; MPEG-4 is then the first international standard introducing the concept of "synthetic and natural hybrid coding" (SNHC).
2. compose these objects together to create *compound* audiovisual objects that form audiovisual scenes;
3. multiplex and synchronize the data associated with *AVOs*, so that they can be transported over network channels providing an appropriate QoS for the requirements of the specific *AVOs*;
4. interact with the audiovisual scene generated at the receiver's end, according to limitations set by the content provider.

Audiovisual scenes are composed of several *AVOs*, organized in a hierarchical fashion. At the leaves of the hierarchy tree there are the so-called primitive *AVOs*, such as a 2-dimensional fixed background, the picture of a talking person (without the background), the voice associated with that person, etc.

MPEG standardizes a number of such primitive *AVOs*, capable of representing both natural and synthetic content types, which can be either 2- or 3-dimensional. In addition to the *AVOs* mentioned above and shown in Figure 3, MPEG-4 defines the coded representation of objects such as

- text and graphics;
- talking heads and associated text to be used at the receiver's end to synthesize the speech and animate the head;
- animated human bodies.

1.1.2. Coding of Audio Visual Scenes

In their coded form, these objects are represented as efficiently as possible. This means that the bits used for coding these objects are no more than necessary to support the desired quality and functionality. Examples of such quality and functionality are error robustness, allowing extraction and editing of an object, or having an object available in a scaleable form. It is important to note that, even in their coded form, objects (aural or visual) can be represented and manipulated independently of their surroundings or background.

1.1.3. Scene Composition and VRML

Figure 3 gives an example that highlights the way in which an audiovisual scene in MPEG-4 is composed of individual objects. The figure contains compound *AVOs* that group elementary *AVOs* together. As an example: the visual object corresponding to the talking person and the corresponding voice are "composed" together to form a new compound *AVO*, containing both the aural and visual components of a talking

person. Such grouping capability allows authors to construct complex scenes, and enables users to manipulate meaningful (sets of) objects.

More generally, MPEG-4 provides a standardized way to compose a scene, allowing for example to apply streamed data to AVOs, in order to modify their attributes or to change, interactively, the user's viewing and listening points anywhere in the scene inside the specified borders.

The scene composition borrows several concepts from another standard, the Virtual Reality Modeling Language or VRML [53] in terms of both its structure and the functionality of object composition nodes. Anyway there are also several new concepts and remarkable enhancements, such as a much more powerful sound description and the standardized way to encode description into an efficient binary coding. Some more details about MPEG-4 scenes will be given in subsection 1.3.

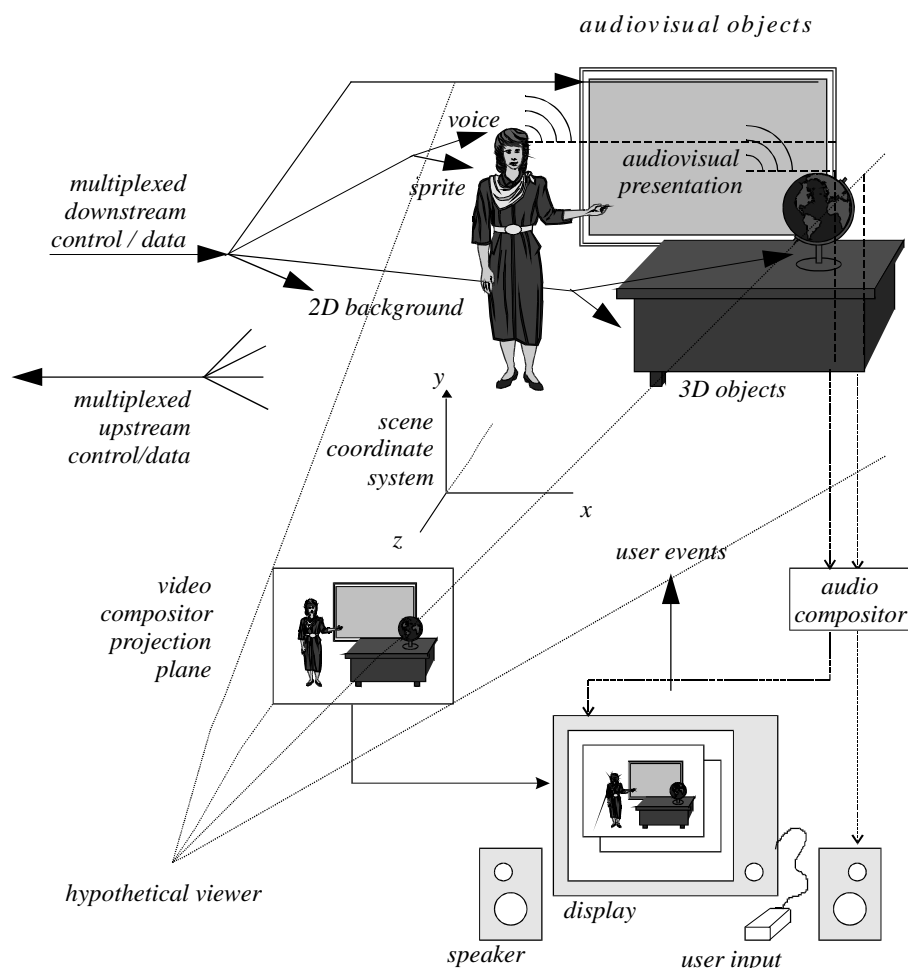


Figure 3 An example of an MPEG-4 Audio-Visual Scene (from the MPEG-4 standard)

For transmission, AVOs and other relevant information, like scene descriptions or IPMP, are conveyed through elementary streams multiplexed into an MPEG-4 stream. At the receiver side the process is reversed, the stream is demultiplexed and the

several elementary streams are sent to specific decoders. In general, the user observes a scene that is composed following the design of the scene's author. Depending on the degree of freedom allowed by the author, however, the user has the possibility to interact with the scene. Operations a user may be allowed to perform include [51]:

- change the viewing/listening point of the scene, e.g. by navigation through a scene;
- drag objects in the scene to a different position;
- trigger a cascade of events by clicking on a specific object, e.g. starting or stopping a video stream;
- select the desired language when multiple language tracks are available;
- more complex kinds of behavior can also be triggered, e.g. a virtual phone rings, the user answers and a communication link is established.

1.2. Audio Coding in MPEG-4

Audio in MPEG-4 is essentially composed by three groups of tools [54]:

1. Natural Audio, which contains a certain number of tools for conventional audio compression, i.e. tools based on lossy algorithms in the domain of either time or frequency. In particular MPEG-4 supports an enhanced version of MPEG-2 AAC (Advanced Audio Coding), similar to the popular MPEG-1 Layer 3 (MP3) standard, and Code-Excited Linear Prediction (CELP) for voice coding.

2. Synthetic Audio, where audio samples are generated directly at the decoder side, without any kind of natural recording and compression at the encoder; two tools belong to this family, Structured Audio and the Text to Speech Interface.

3. Systems AudioBIFS, which represents the upper level of the audio encoding and allows, through a high level language, the description of audio scenes based on the composition of different audio objects, as explained in the previous subsection. The first group of tools, on the time being, is the most popular and it is widely used in the audio community, since it is well known in its behavior and quality, and the implementation of the several different decoders does not represent a hard task for the last generation of processors and DSP units.

The second group of tools represents instead a more modern approach to audio encoding, and until some time ago tools of this type were not very much considered outside the world of academic research laboratories. Structured Audio in particular will be described in details in the second half of this chapter and will constitute the high level programming language supporting the virtual modeling method proposed in this dissertation.

1.3. The MPEG-4 Systems Layer: BIFS Scene Description

The Systems layer of MPEG-4 specifies functionality for the communication of interactive audio-visual scenes. More specifically three kinds of information are in the scope of this layer [55]:

1. the system level description related to the coded AVOs, e.g. scales, references, origins etc.

2. the coded representation of spatio-temporal positioning of AVOs and the behavior in response to interaction
3. the coded representation of information related to the management of data streams, like synchronization, identification, association of stream content, etc.

Among these parts, the most relevant is the scene description, which is coded by a mark-up language called Binary Format for Scenes, or BIFS. BIFS enables the concise transmission of audiovisual scenes "composited" from several pieces of content such as video clips, computer graphics, recorded sound and parametric sound synthesis. A scene description follows a hierarchical structure that can be represented as a graph. Nodes of the graph form audio-visual objects, as illustrated in Figure 4, which corresponds to the content of Figure 3.

The structure is not necessarily static; nodes may be added, deleted or modified.

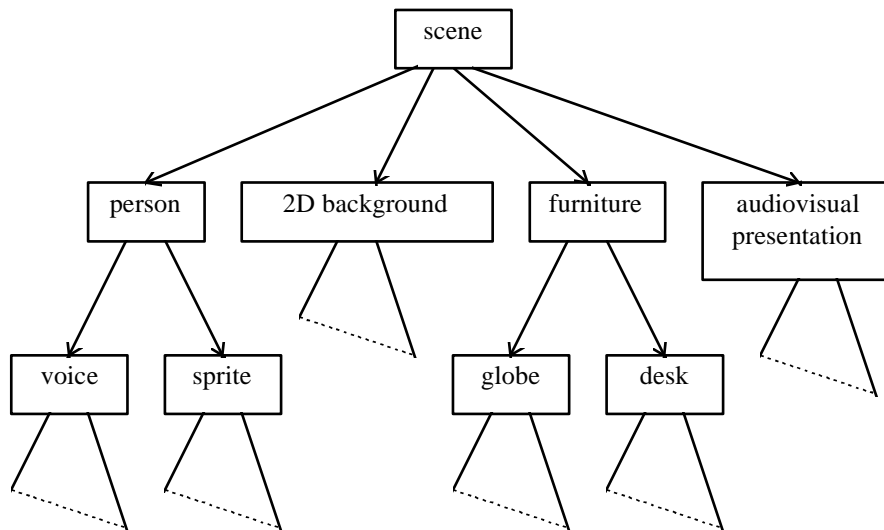


Figure 4 Logical structure of the scene represented in Figure 3.

1.3.1. AudioBIFS

The part of BIFS controlling the composition of sound scenes is called AudioBIFS. AudioBIFS provides audio extended capabilities, which are much more powerful and efficient than those present in similar multimedia languages such as the already mentioned VRML; it provides mechanisms to perform mixing, delay, 3-D spatial processing or other additional effects, to enhance the audio experience delivered by high-quality coding tools such as AAC. Some of the processing routines present in a typical composition algorithm require a considerable amount of DSP-like processing power, and this could severely affect the performance of a general purpose processor that is managing the decoding process of a complete audio-visual interactive scene [56], [57].

An example of audio scene is presented in Figure 5. In particular, one of the audio nodes (AudioFX) requires the implementation of a Structured Audio decoder and is

used for algorithm downloading. Audio nodes are conceived with a different philosophy, more coherent with audio applications than the visual approach of VRML: they are organized as a processing subtree, where information streams from AudioSources through processing and mixing nodes until the Sound (or Sound2D) node, where they finally receive the connotation of objects and are placed in the scene.

In addition to the nodes defined in MPEG-4 version 1, the second version of the standard provides extended features, particularly concerning environmental spatialization and acoustic properties of materials [59],[60]. These new features will permit a complete and satisfactory description of the most advanced acoustic scenes [58].

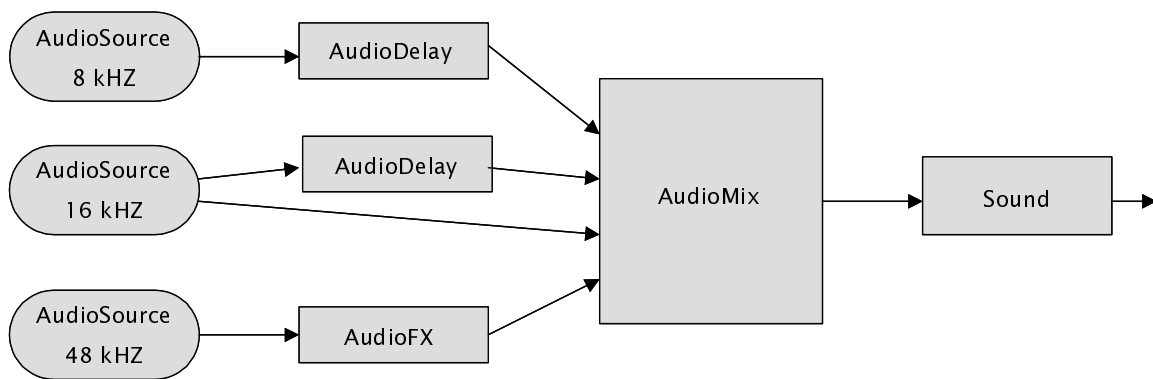


Figure 5 Example of Audio scene description using the BIFS nodes

2. Structured Audio Coding and MPEG-4

2.1. Structured Representations of Audio Content

Structured audio representations are description formats that are made up of semantic information about the sounds they represent and that make use of high-level or algorithmic models. Examples of structured audio representations that are known in the literature are musical-instrument digital interface (MIDI, [61],[62]), music synthesis languages, linear-prediction model of speech, etc.

A structured audio representation has a number of features by which it can be identified. The most straightforward characteristic is that structured representations encode a signal, a sound, in relation to a model [63],[64],[65]; this means that they make assumptions about the nature of the sound being represented and use these assumptions in defining a parameter space. In general, the lower the dimensionality of the feature space, the more structured the representation is. In a very-low-dimensional representation, in fact, each coordinate must have relatively higher meaning than in a higher-dimensional space. In a structured representation the parameters are individually interpretable as representing certain high-level features about the sound, and their manipulation gives simple control over physical and perceptual aspects of the sound.

2.1.1. *Structured Audio as Coding Mean*

For many domains, there is great applicability for sound representations that only encode partial information about a sound or that cannot be easily encoded from a sound waveform. This raises the problem of whether the format is, in fact, *encodable* (i.e. how easy it is to extract the representation directly from an audio waveform) and whether it will be *decodable* (i.e. whether the representation can be turned back into sound without additional information) at the receiver's side.

Structured audio methods are useful for sound compression and transmission because they exploit signal redundancies that other coding methods cannot exploit. Existing sound compression methods work in one of two ways.

- Lossless coders remove entropic or information-theoretical redundancy from a digitally sampled signal. This redundancy arises from the fact that successive samples are not statistically independent and that some sample values occur more often than others do. Sounds coded in this manner may be exactly reconstructed.
- Perceptual coders remove perceptual redundancy from a signal, that is, redundancy created by overspecifying the sound format with regard to the human perceptual system and including details that cannot be perceived. When the perceptually unneeded information is removed, the result is a lossy coding scheme; this means that the original waveform cannot be reconstructed exactly from the coding. However, the reconstructed waveform will sound like the original to a human listener [66].

Most sound signals also contain the so-called structural redundancy, which arises in several ways. First, many notes or sound events in a soundtrack sound the same or nearly the same; in waveform coding, these events will simply be represented multiple times. Using timbral models, it is possible to concisely describe the similarities and differences between different notes characterized by the "same" timbre. Many soundtracks contain sections of repeated material; for instance, drumbeats in popular music, and exact recapitulations in classical music. If these repetitions can be represented symbolically, it is possible to reclaim a great deal of redundant information.

Last, many sounds are more simply represented as processes than as waveforms; for example, consider speech sounds processed to add artificial reverberation. It could be effective the usage of a highly efficient speech coder (like CELP) to transmit the flat, unreverberated speech, and parameters to (or an algorithmic description of) a reverberation algorithm; the effect postprocessing is then performed after the speech is decoded. This is a more efficient transmission method than using a general-purpose coding scheme to transmit the reverberated waveform.

2.1.2. *Sound Synthesis*

The tasks of encoding and decoding are only one area within the study of structured sound representations. For other domains, such as interactive music systems, content-based retrieval and virtual environments, the capability to construct synthetic sounds from scratch or from a semantic description of its timbral properties is become more and more important.

A wide variety of sound synthesis algorithms is currently available either commercially or in the literature [63]. Each one of them exhibits some peculiar characteristics that could make it preferable to others, depending on goals and needs. As stressed earlier, technological progress has made enormous steps forward in the past few years as far as the computational power that can be made available at low cost is concerned. At the same time, sound synthesis methods have become more and more computationally efficient. As a consequence, musicians can nowadays access a wide collection of synthesis techniques (all available at low cost in their full functionality), and concentrate on their timbral properties.

Each sound synthesis algorithm can be thought of as a digital model for the sound itself [65]. Though this observation may seem quite obvious, its meaning for sound synthesis is not so straightforward. As a matter of fact, modeling sounds is much more than just generating them, as a digital model can be used for representing and generating a whole class of sounds, depending on the choice of control parameters. The idea of associating a class of sounds to a digital sound model is in complete accordance with the way in which musicians classify natural musical instruments according to their sound generation mechanism. For example, strings and woodwinds are normally seen as timbral classes of acoustic instruments characterized by their sound generation mechanism. It should be quite intuitive that the degree of compactness of a class of sounds is determined, on one hand, by the sensitivity of the digital model to parameter variations and, on the other hand, on the amount of control that is necessary for obtaining a certain desired sound.

2.1.3. SWSS Languages

A particular and well-established method for structured audio descriptions is the exploitation of specific programming languages.

Languages for Software Sound Synthesis (SWSS Languages) deep their roots in the academic research of the '60s. Among several efforts and different techniques, Music V-style tools [67] are recognized as the most effective and accepted worldwide. In these tools a musical representation is programmed in two parts, using two different languages: the instrument² definition describes the connections of signal generators and/or processors for the sounds to be used, and the event (note) list is the score, described in terms of parameters sent to the instrument. A programmer or a content creator makes a "patch" among modules such as oscillators, amplifiers, mixers and control function generators (called unit generators in SWSS languages), then sends trigger and control data to the patch to make or control sound.

At the very beginning these languages were based uniquely on unit generators. At a later stage many of them became based on syntax from general purpose programming languages, allowing the modeling of more general and flexible algorithms by expressions and conditional branches; it is clear that the most widely

² In SWSS languages an instrument is normally intended in a wide sense, including in the term blocks of code for the description of both synthesis algorithms and processing algorithms.

used template has been the C programming language [68], even if other languages like LISP and Pascal inspired other efficient solutions [69].

2.2. *Structured Audio in MPEG-4*

MPEG-4 Structured Audio (SA) can be viewed as a collection of tools which enables the description of audio synthesis and audio processing algorithms, exactly as it happens in Music V-styled tools. Actually it is an evolution of the MIT's Csound language [72], redesigned for specific needs of sound coding and transmission and further improved by the several months of discussion and work inside the MPEG Audio subgroup. SA, differently from all other MPEG-4 components, does not standardize a particular set of synthesis or processing methods, but a way for describing such methods. This means that any current or future sound-synthesis or sound-processing algorithm can be described in the MPEG-4 Structured Audio format [69]. The generality of SA will be further analyzed in the next section 2.3.

A Structured Audio bitstream allows the transmission and decoding of synthetic sound effects and music using several techniques, potentially all the known ones. Using SA, high-quality sound can be created at extremely low bandwidth, and this is again a major difference in comparison to all other coding methods, where the bitrate is normally proportional to Audio quality. Quality in SA only rely on the goodness of the description model, independently from the bitrate. Typical synthetic music may be coded in this format at bit rates ranging from 0 kbps (no continuous cost) to 2 or 3 kbps for extremely subtle coding of expressive performance, using multiple instruments. This is true once the correct algorithm describing synthesis and/or processing, and possibly the wavetables necessary to the specific sound model, have been downloaded in the bitstream header and correctly interpreted by the SA decoder: this may require a few kilobytes to a few megabytes of information.

2.2.1. *Major Elements of the MPEG-4 Structured Audio Toolset*

There are five major elements in the Structured Audio toolset [70],[71]. They are graphically summarized in Figure 6.

The Structured Audio Orchestra Language, or SAOL. SAOL is a SWSS and digital-signal processing language that permits the description of arbitrary synthesis and processing algorithms as part of the content bitstream of MPEG. The syntax and semantics are the only parts of SAOL standardized in MPEG-4 (i.e., no algorithms), together with a built-in core library. The syntax of SAOL can be considered as an extended subset of the C programming language. As it will be explained in details later, the main differences from C lie in the unique data format (32-bit floating-point format), in the lack of pointers and in the possibility to declare only one-dimensional data structures. These apparently very strong limitations cope very well with the purpose of SAOL, since they correspond to peculiarities of the audio data.

A basic extended feature of the SAOL language is the presence of control variables (ksig) and audio variables (asig): each of them is "executed" (i.e. updated) at its specific frequency, called respectively *krate* and *srate* in the two cases. These two frequencies are programmable in each single program. Variables in the global block

can be shared among all the instances of every single instrument, while local variables are private to each instantiation.

SAOL is the programming language that describes the *orchestra*. The orchestra is the main concept behind the SA performance: it is a collection of instruments and connection buses. Instruments are the real algorithms, describing physical elements for sound synthesis as well as effect procedures for sound processing, in the same way in which they can be described in C. Buses are buffer lines used to connect the several instruments among them, allowing the easy link of complex synthesis-processing chains. Buses are instantiated through special global statements (or calls) of the SAOL language.

Unlike its predecessor CSound [72], SA has a sample-by-sample (from now on 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_i with

$$B_i = \text{srate/krate} \quad (1)$$

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.

Further details about SAOL will be presented in Chapter 4.

The Structured Audio Score Language, or SASL. SASL is a simple score and control language, which can be used to describe the manner in which sound-generation algorithms described in SAOL (the orchestra) are configured to produce sound, or the manner in which control parameters of specific processing algorithms evolve in time to produce particular effects.

SASL contains five types of statements:

1. *Instr* statements (instrument) are used to instantiate voices, i.e. instances of an instrument, during synthesis processes and to specify their main parameters, like pitch, start time, duration, and other variables specific to each single instrument.
2. *Control* statements are used to change the values of the control variables in the SAOL algorithms. Every control variable in each particular instrument of the orchestra, as well as global control variables, can be modified in real time, allowing a continuous update of the synthesis and processing algorithms.
3. *Table* statements are used to instantiate new tables to be used by a specific SAOL orchestra or to destroy tables that are no more useful for the current process. This statement is useful for a correct managing of the memory space, since this often represents a critical issue for multimedia systems.
4. *Tempo* statements are used to control the speed of a SA performance. The tempo variable controls the ratio between the score times describing streaming events and the absolute reference time described in seconds.
5. *End* statement states the orchestra time at which the performance is over and the complete orchestra space can be freed.

The Structured Audio Sample Bank Format, or SASBF. The Sample Bank format allows for the transmission of: 1) banks of audio samples to be used in synthesis algorithms based on wavetables, and 2) the description of simple processing algorithms to use with them. The utility of this tool is evident, since wavetable synthesis is the most popular synthesis technique in many present musical synthesizers and in the majority of multimedia audio boards.

Normative reference to the MIDI standard [62], standardized externally to MPEG-4 by the MIDI Manufacturers Association. MIDI is an alternate mean of structural control, which can be used in conjunction with or instead of SASL. Although less powerful and flexible than SASL, MIDI support provides to MPEG important backward-compatibility with much existing content and authoring tools.

A normative scheduler description. The scheduler is the core run-time element of the Structured Audio decoding process. It maps structural sound control, specified in SASL or MIDI, to real-time events dispatched using the normative sound-generation algorithms. Each event received through the score file at the beginning of the downloaded bitstream, as well as each stream event received in real time from a far or local terminal, is decoded and scheduled in the event queue; at each control period (programmable through the *krate* variable) instruments are instantiated or terminated, control parameters are updated, and all the active instruments (or procedures) in the orchestra are executed in order to generate or process a buffer of output samples at the audio rate (i.e. the sampling rate in common natural decoders).

2.2.2. How SA is linked to MPEG-4 Scenes

In MPEG-4, SA is used as a sound-synthesis model generator, but also as a description method for sound effects and other post-production algorithms. The BIFS AudioFX node (see [55],[58]) allows the inclusion of signal-processing algorithms described in SAOL in the AudioBIFS scene, which are applied to the outputs of the sound nodes subsidiary to that node in the scene graph. This functionality fits well into the described instrument-bus methodology in Structured Audio orchestras.

Each node in a BIFS scene graph that contains SAOL code is then either an AudioSource node or an AudioFX node. If the former, there are no input sources to the SAOL orchestra; in this case the samples presented to the output buffer are generated through the synthesis algorithms used to model the specific instruments; these samples are possibly processed by additional effect instruments, until they reach the output bus and finally the output buffer. In this case, the Structured Audio object is seen from the outside in the same way of a normal natural audio object generated by each of the other MPEG-4 natural audio decoders.

In the case of an AudioFX node, the children nodes of the node provide several channels of input sound to the orchestra, which are stored in a special input bus of the SA decoder. From this bus, they may be sent to any instrument(s) desired and the audio data provided in such a way must be treated normally. The number of orchestra input channels is the sum of the numbers of channels of sound provided by each of the children, and thus it is not fixed a priori in any way; as for control

and audio frequencies, the number of input and output channels to and from each SA orchestra (and then each performance) is fully programmable by the content provider.

In conclusion SA provides the new MPEG-4 standard with a very flexible, audio-oriented description tool (see next section) at the price of having additional decoding complexity in comparison to normal audio coding schemes [54]. This can be a problem for some types of low-cost applications oriented towards the mass market, but at the same time it gives a great new possibility to audio coding; this complexity problem will be one of the topics discussed in details throughout the core of this dissertation.

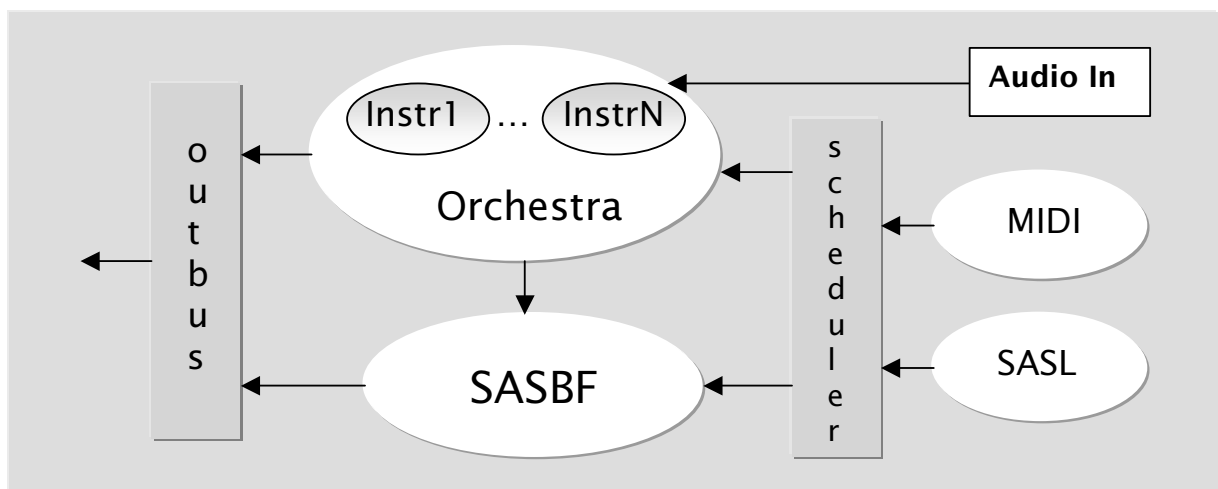


Figure 6 Major elements of the MPEG-4 Structured Audio toolset

2.3. Structured Audio as a General Coding Scheme

Until the inclusion of SA inside the MPEG-4 International Standard, it was not usual to refer to SWSS languages as to encoding/decoding techniques.

In fact, as consistently proved in [73], the term "coding" can be used to refer to any procedure that can be described through a model. For example, an audio codec such as MPEG-4 AAC [74] or MPEG-4 CELP can be considered as the combination of:

1. a sound-understanding algorithm that set the parameters in a representation space based on the analysis of the acoustic waveform
2. the transmission of these parameters
3. a sound synthesis algorithm that maps from the transmitted parameters to a new sound.

The understanding step can be called the encoding step, and the synthesis step the decoding step, but despite the terminology the underlying reality is one.

The same three steps can be recognized for a structured audio description, as it appears evident from the previous introduction. Simply, the understanding (encoding) side is less computationally demanding for e.g. MIDI than for AAC, but more general for AAC than for MIDI (much more waveforms can be represented, since the AAC encoder can be used on every sound, whereas the pitch tracking required by MIDI only applies to a subset of well-defined monophonic sources).

At this level of abstraction a classical coding scheme becomes a tracking / synthesis scheme, while on the other hand a structured audio description can be seen as a compression / decompression scheme. In both cases this assumption is based on the real fact that any kind of signal description requires a model and input parameters to the model. From this viewpoint it is not difficult to extend the reasoning to every kind of program: a mixing console is based on a model (the mixing matrix), it receives the parameters (coefficients and input samples) and it synthesizes the new output (the mixed channels).

Once this concept is assumed, it is clear that any kind of traditional audio (and multimedia in general) coding is a specification of a model, standardized or not, to which it is necessary to provide input parameters to generate the correct output. MPEG-4 Structured Audio is conceived a step above, in the sense that it provides a mean to "describe" coding models through the SAOL language and to transmit input parameters to the model through the SASL language. Due to the extreme novelty and to the unexplored complexity of such a tool in a coding environment, it is not imaginable to reduce every audio track to a structured audio description, even if the theoretical possibility does exist. SA includes in its core library of functions some functionality like filters, ffts, noise generators, sinusoidal operators that are at the base of many coding schemes. In [73] it is further demonstrated that a structured audio toolset such as MPEG-4 Structured Audio is a general coding scheme, in the sense that any coding and/or processing procedure can be described by this approach, and that the bitstream that must be delivered in this case is always the minimal required if the model is correctly described. In conclusion it is possible to state that SA is capable of delivering any sound model and that it is a universal minimal audio format.

2.3.1. SA as an Alternative to More General Languages

At last, it is interesting to spend a few words to understand if and how another general purpose programming language, such as C or JAVA, can be used for the same purpose and then if a dedicated SWSS toolset is necessary or not. In fact, C (or JAVA as well) technology is more advanced than the SA one, as technical considerations in the next chapters will clearly demonstrate. At the same time it is not wise to exclude a priori a new technology just because it is less mature than another. The main reason to choose this approach is twofold, considering the domain of real-time application for media communications.

First of all SA is a general programming language but with strong limitations in comparison to C or other languages. This does not make it suitable for a range of applications larger than the audio domain, but at the same time it permits to move to this new domain, modeling and coding, with an easier-to-use tool. It must not be forgotten that a coding scheme always requires, at least for real-time targets, an automatic generation of the parameters, and in this case also of the description model; then it is simpler to move to this domain with a partially dedicated environment, which moreover can be better exploited on a larger range of platforms including DSPs, as it will be shown later.

Secondly, SA comes with a standard internal scheduler, which is not completely flexible, being it tied to some peculiar execution steps, but it allows to rely on a solid base for real-time applications, since it is conceived specially for that. It will be described how a decoder cannot be classified as compliant with the standard if it is not able to run in real-time, obeying to the scheduler, programs below certain well-defined levels of complexity. The definition of this kind of guaranteed levels is one of the main issues and contributions of this work and it is absolutely not provided by any other programming language.

CHAPTER 4. TOWARDS A LANGUAGE-ORIENTED VIRTUAL MODEL FOR SIMULATION OF REAL-TIME APPLICATIONS

This chapter presents the virtual model on which simulation of real-time applications is based. It constitutes then one of the most important parts of the dissertation; here innovative directions for research are introduced.

In the first part, after a short review and critic of the issues previously described in Chapters 2 and 3, the concepts behind the development of the virtual model will be described. In the second part these concepts will be applied to the specific case of the Structured Audio Orchestra Language, which for several reasons constitutes, as it will be shown, a good candidate for the validation of the proposed approach.

In the end, the first application case, i.e. the standard procedure for the MPEG-4 SA Conformance Test, will be explained.

1. Multimedia Systems, Simulation and Complexity

In Chapter 2 some meaningful simulation techniques and tools have been reviewed and trends of system design have been introduced. The underlying basic goal of all the tools for architecture simulation is of course that of being able to preview the behavior of some architecture when selected applications run on it. When specified, experiments for the reviewed simulation tools were conducted using the SPEC'95 benchmark suite [75] (only recently replaced by the SPEC2k suite [77]). This suite contains a collection of typical, well-known algorithms that cover some aspects of the world of the so-called general-purpose programming; as an example the test bench suite of C-written programs is briefly summarized in Table 2. The suite of programs written in Fortran is naturally far from the multimedia world, including problems of applied mathematics, physics and chemistry.

Table 2 Summary of the SPEC'95 CINT95 benchmark suite; applications are written in C.

Program	Description
go	Artificial intelligence; plays the game of "Go"
m88ksim	Chip simulator; runs test program
gcc	GCC compiler; builds SPARC code
compress	Compresses and decompresses file
li	LISP interpreter
jpeg	Graphic compression and decompression
perl	Manipulates strings and prime numbers in Perl
vortex	A database program

It follows that all these tools have been conceived to provide simulation of typical environments composed by a workstation processor with known general-purpose RISC instruction set, where applications are supposed to be developed in high level languages such as C, C++ or JAVA. Only in the last two analyzed cases, Mermaid and Zhu/Gajski, methods are proposed that may be able to provide real abstraction from a specific platform and to virtualize the simulation environment.

Simulation frameworks more related to hardware design like those contained in commercial software/hardware co-verification tools are again dependent on specific low-level instructions, and in all cases simulation of complex logic designs for intensive tasks can be performed only exploiting simulation toolkits for the supported cores. This means that, before the purchase of a suitable development kit and corresponding design tool, it is necessary to select the core, or family of cores, that will be used for the final implementation. This is often hard to be done if the application itself is not available in advance in its complete technical details. Finally, these tools are not very useful to "guide" the designer through a wide range of implementation problems and potential solutions.

1.1. Standards and Design of Applications

Chapter 3 introduced the issue of standardization and particularly what is probably the greatest effort in this sense in the field of multimedia: MPEG-4. In terms of complexity evaluation and implementation issues, standards, and MPEG-4 in particular, offer some interesting points that must be carefully analyzed.

Standards arise from the necessity of interoperability among different agents. The only alternative to a standard is a "de-facto" standard, where a single agent occupies a dominant role in a certain field and it is then in the position to impose its own rules for communication among parts. Since for multimedia and real-time protocols this is certainly not the case, complexity evaluations for a selected or a currently available solution cannot be accepted as valid criteria to establish metrics for the standardized domain. This means that it is not possible to measure the associated complexity of an application by running it on a given processor, with a given operating system and using a common programming language, because in this way it is not possible at all to know a-priori, without owning the "conformant" system, if and how another completely different solution will be appropriate for the same task or not. In practice, a level of abstraction is needed to evaluate platforms and their corresponding applications.

A confirmation of this need is the fact that standards are often conceived to target some particular systems, but practice and experience have shown that suitable systems can change according to available technology, to power consumption requirements, to security requirements and to ease of use of the applications even before the final publication of the standard itself. A complexity estimation of the standard based on a specific system would break it as soon as the target platform is no more acceptable, while an abstract simulation criterion can always be mapped to an actual solution, which can be tested by this model for compliance.

1.2. MPEG-4 and the Design of Applications

If the focus is now moved to the MPEG-4 standard in particular, there are at least another couple of important factors that can be considered and that in some way make the currently adopted methods for development and test non optimal for the considered spectrum of applications.

First of all, the problem of complexity. If many of the coding standards of the past were characterized by a high encoding complexity (at the server side) and by a much lower decoding complexity (at the client side), in MPEG-4 this assumption is often reversed. Scene descriptions, synthetic aural and visual content, easily convey heavy workloads to the decoder system (estimations are in the range of GFlops for fully-functional and interactive scenarios [54]); this means that either expensive, high-performance general purpose (multi-)processors are applied to every consumer end terminal, or conversely some cheap, dedicated or semi-dedicated hardware accelerator needs to be included in the system. From what has been discussed in previous chapters, this can reasonably be done nowadays by developing appropriate systems on a chip; in any case at least some parts of the standard still welcome the development of custom or semi-custom solutions. This is in partial contrast with an a-priori selection of the instruction set and architecture, done without a preliminary and systematic analysis of the desired system.

Moreover, some parts of MPEG-4, namely scene description and Structured Audio, go up one level in normative specification, and standardize not a specific decoding algorithm to decode the bitstream, rather a general/scripting language or a programming language through which the algorithms themselves and the complete applications can be downloaded and made responsive to interaction at a high level to the end user. Automatically, the application is no more known a-priori, except for its basic building blocks, and different systems could react in completely unpredictable ways if they were not tested and configured on the basis of some fundamental test principles. A standard like MPEG-4, lying for tradition in a context aiming at providing the best technology and the best possible Quality of Service (QoS) cannot stand without a consistent Conformance test.

When these remarks are compared with common practice in analysis and design techniques, it appears evident that a simulation based on a well-defined instruction set (i.e. assembly language), or a design made assuming a specific core as starting point cannot be enough anymore to proceed to the search for an optimal solution: this could only be achieved by chance.

What is instead necessary is:

1. to start with a systematic analysis of the language description methods, then
2. to characterize the main "building blocks" of the potential application and finally
3. to proceed in parallel to the efficient, but low-level, further analysis of these building blocks and to the higher level, abstract modeling of applications based on these building blocks.

This is what have been done and will be described in the rest of the chapter.

2. Virtual Analysis of a Programming Language

Each programming language is composed by a collection of rules of syntax and semantics that defines a computation model or, in other words, an abstract machine [78]. Then, moving from this definition, it is not difficult to see a programming language as an abstraction of a physical computing device, and indeed there is no great conceptual difference between a sequence of assembly instructions and a program in a higher level software language. Indeed, a programmable device defines a computation model based exactly on its assembly language definition, i.e. its instructions and memory structures; on the other hand the C programming language defines an abstract machine that, according to the specific program, is composed by C instructions and operators and by allocated memory for data and instructions themselves.

Further, it is possible to define computation as the solving of a problem by means of a(n abstract) machine and a program as a description of the computation that makes use of instructions and storage space of the machine [78].

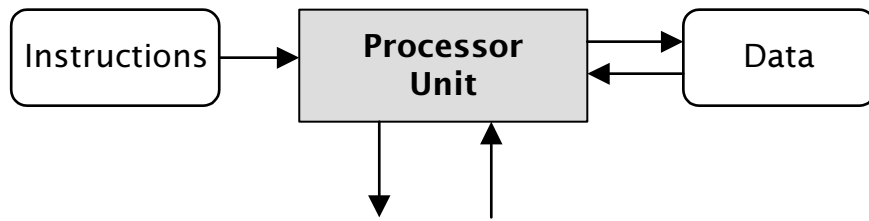
Given these definitions, and since the number of programming languages differing by functionality and use is nowadays huge, it is desirable to try a classification of the different languages moving from the programming technique, or programming paradigm [79]; this classification will permit to select which languages are suitable for analysis according to the method that will be described later, and their main characteristics that should be taken into account.

2.1. Programming Paradigms

A programming language can be said to support a programming paradigm, a programming style, if it provides facilities that makes it convenient to use that style. A language does not support a technique if it takes great effort or skill to write such programs [79]. Moreover, it is important to remark that it is not so much relevant what features a language possesses but that the features it does possess are sufficient to support the desired programming paradigm in the desired application areas. Programming languages can be divided, based on their structure, into four main paradigms [78]. They will be briefly described here according to the type of abstract machine they define, to the way in which the abstract machine evolves to execute a program and to the form that computation takes during the execution of the same program. Some examples will also be given to better identify each case. It is understood that often the border lines among languages and paradigms are not precise; some languages, like C, are so general and flexible that they could support every programming style, even if they are better suited for a specific one.

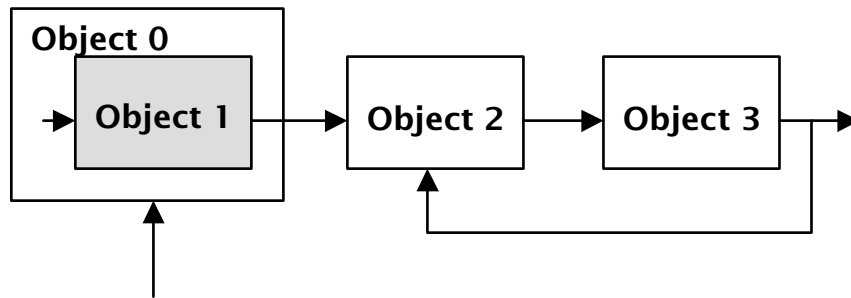
2.1.1. Imperative Programming

In the imperative programming paradigm the abstract machine is composed by a processor unit executing instructions and by a memory space where data are stored. During the execution of the program the stored data are modified by the processor unit. Computation is specified by the sequence of the several instructions that are necessary to execute the particular task. Examples of languages supporting the imperative paradigm are C, Pascal, Fortran and Basic.



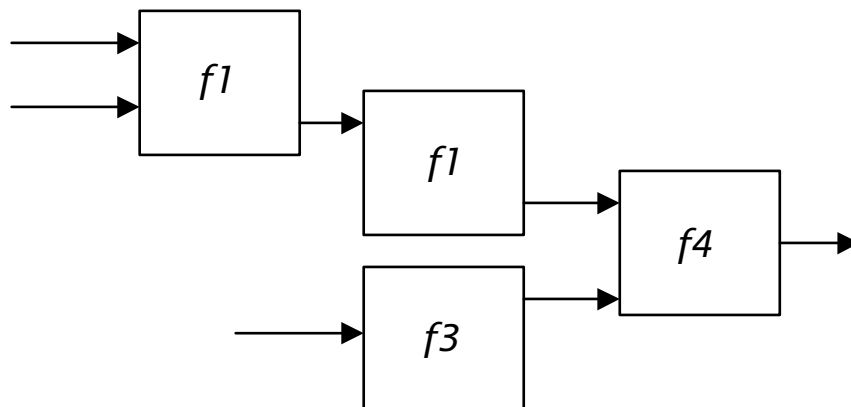
2.1.2. Object-Oriented Programming

In the object-oriented programming paradigm the abstract machine is composed by a collections of objects, which are defined according to the syntax of the language. During the execution the different objects composing the abstract machine interacts via message passing. Computation is specified by the sequence of interactions among objects that are required to execute the particular task. Examples of languages supporting the object-oriented paradigm are C++, Smalltalk and JAVA.



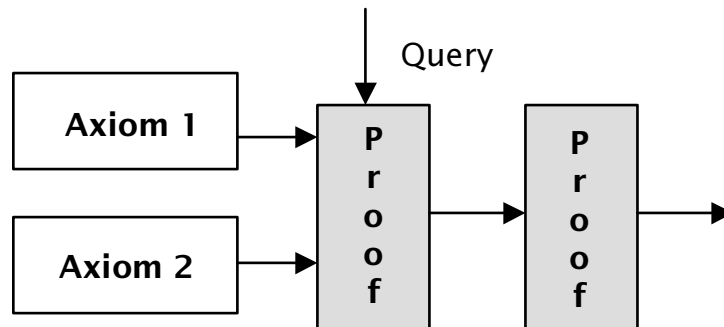
2.1.3. Functional Programming

In the functional programming paradigm the abstract machine is composed by a collection of functions. During the execution the several functions generate an output that is used to activate other functions. Computation is specified by a sequence of compositions of the different functions. Examples of languages supporting the functional paradigm are Lisp, Common Lisp and ML.



2.1.4. Logic Programming

In the logic programming paradigm the abstract machine is constituted by a theorem prover. During the execution a new deduction is derived from the collection of already proven theorems. Computation is specified by a sequence of deductions of new theorems. An example of language supporting the logic paradigm is Prolog.



2.1.5. About Parallelism

An additional paradigm, sometimes referred to as the concurrent paradigm, has not been considered explicitly. There is a strong impression that concurrent programming is something still too much bound to the specific hardware platform, since so far the enormous efforts devoted to this interesting branch of research have not produced results that could be reused in general form [12]; rather, either specific languages are conceived for specific problems, or general purpose languages are extended to deal with forms of concurrency. In any case, the concurrent paradigm is not considered an independent potential case for the kind of abstract analysis that is described here. It will be shown later how and to which degree parallelism will be supported by the proposed abstract simulator.

2.2. Paradigms and Multimedia Applications

Among the different paradigms surveyed in the previous subsections the most important ones for multimedia real-time applications are imperative and object-oriented. Functional programming has important application areas. Artificial intelligence programming is done in functional programming languages and the AI techniques migrate to real-world applications. Functional languages have been sometimes used for computer music and signal processing (in particular Common Lisp, see [69]), but its use in this domain remains occasional. Logic Programming is a very elegant style, but often not efficient and unsuitable for signal and state processing.

The object-oriented paradigm gained fast a great acceptance in the last years, also because the computational power made available by hardware devices became enough to support the further level of abstraction provided by this style of programming. In fact, in strict object-oriented programming, data and interactions among data are hidden to methods and made visible to objects only: this means that data and instructions are often accessed through redirections and then they

require more computation. But in most popular cases object-oriented languages evolved from imperative languages, and actual methods are in most cases implemented in an imperative fashion. In the end, as long as an object-oriented language has to be compiled for an imperative architecture, like those of common DSPs and processors, it is not difficult to see it as an extension of an imperative paradigm (and this is what many C++ compilers for instance do). In this sense an "abstract compiler" supports the object-oriented paradigm because it is cross-compileable into an imperative paradigm.

As a final remark, as far as efficiency is concerned, it is not surprising to note that the almost totality of hardware media devices are still conceived and realized in an imperative logic style, leaving to compilers the burden to solve more elegant situations.

2.3. *The Definition of the Virtual Model*

It has been discussed how imperative languages, and their object-oriented extensions, have traditionally been considered the best programming paradigm for multimedia and real-time applications. This consideration permits to reduce consistently the types and number and languages under consideration without losing in generality for the following analysis.

Programs written in standard imperative languages are normally characterized by the following features:

- Data Type definitions: if necessary, derivate data types are defined as composition of primitive, or native, data types.
- Memory allocation: variables belonging to primitive or derivate data types are declared, and this implies a static memory allocation at the beginning of the process. Memory can also be allocated in run-time, with special purpose functions, and this implies a dynamic memory allocation.
- Statements and expressions: the computation is specified by a sequence of statements. Among statements, assignment is the one that allows the modification of a stored variable with a new value calculated through an expression. An expression is a sequence of operations specified by operators, which consumes the specified variables to generate an appropriate combination of them.
- Library functions: standard languages are characterized by a more or less extended library of normative functions, to be provided for linking with each compiler and that can be used in standard form on every platform.

An imperative programming language is then extremely complex from the point of view of its simulation. The more a language is flexible, the more it is difficult to track all its possible features in a correct way. But indeed what is needed is to find a systematic approach to its simulation in order to be able to analyze a program in a fashion that could be valuable at the same time for the widest possible range of devices, either hardware or software or a mix of both. This means that the approach to design that will be investigated consists of starting from a platform-independent analysis of a language and of its typical applications and, only after that, of moving

down to the design of an architecture conceived to support at best the requirements of such specific application, or of a certain class of such applications.

In practice, the goal is to define a model, intermediate between the complex language and the hardware device, able to make abstraction of some of the language complexity and at the same time to take into account the peculiar characteristics of most existing hardware, so that an application written in the language can induce an optimal solution to the computational problem. Further, this implies the capability to define a supporting device or system architecture for an existing application, or class of applications, that are already specified and for whom it is not possible (or reasonable) to change their structure or algorithms. A graphical representation of this approach is presented in Figure 7.

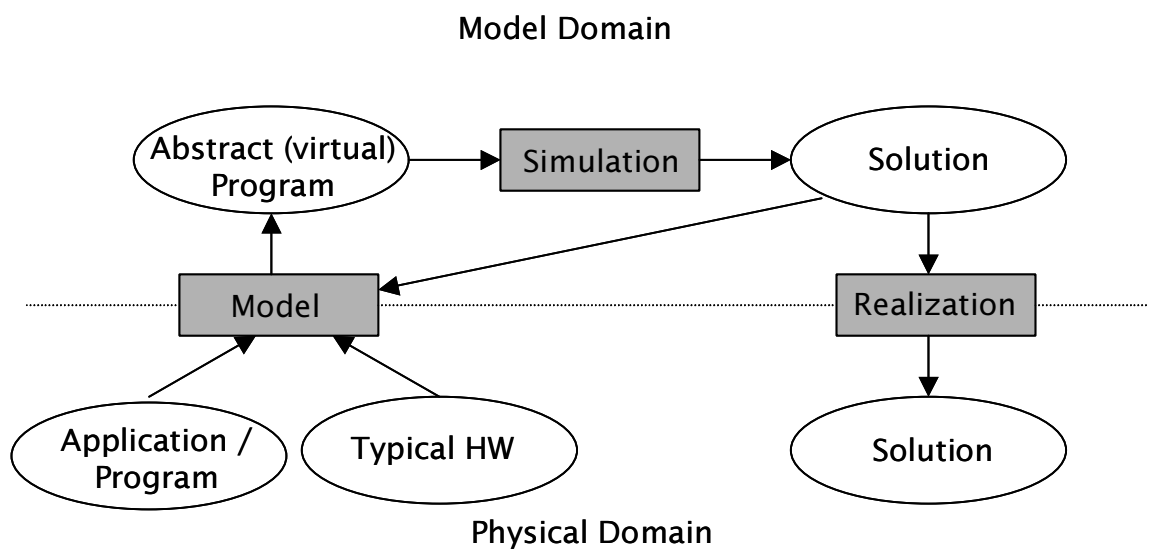


Figure 7 A graphical representation of the proposed approach. An abstract, or virtual, program is derived from the application and modeled through typical hardware reference architectures. Partial solutions can contribute to better refine the abstract model, in order to highlight fundamental features of the application under consideration.

This is partially in contrast to the design process underlying e.g. hardware/software co-design, where hardware and software are supposed to be made available at the same time and with the same flexibility of intervention on both of them; this is often not true and not possible in real cases. To better underline the main difference between the proposed approach and the commonly adopted one, Figure 2 is reproduced in Figure 8, where the interaction with existing cores has been introduced to better represent the practice under consideration.

It is necessary to find a way to abstract the programming language starting from its features and in a way that the resulting model is simpler than it is in reality: this way is the modelization process. Differently from hardware/software codesign, this process can be refined by iterations and simulations of the abstract program over the abstract machine, this latter being defined by the abstract language itself.

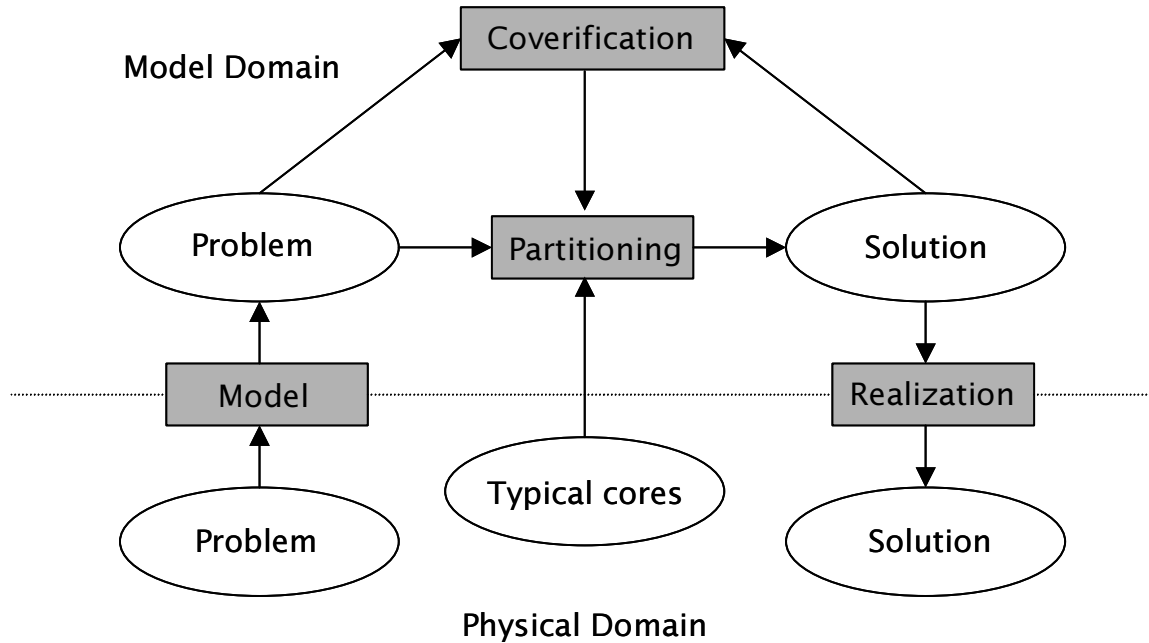


Figure 8 Graphical representation of a practical hardware/software codesign approach

2.4. **Functionality Abstraction in Imperative Languages**

The key point to understand the modelization process and the analysis principles proposed in this dissertation is to realize that the simulation of a real-time application running on a specific machine does not need, at the high level of the general architecture design, to provide a result of the application to be proved for a "bit-exact" correctness, but rather it needs to show if the architecture will support the complexity of the simulated workload or not. The mathematically correct result will be required and tested later, at the lower level of the device implementation.

For instance, let's consider the following two examples:

```
x = input * amp;
output = x * max_16bit_int;
```

and

```
x = input - amp;
output = x * max_16bit_int;
```

The first example, at the low level, normally requires the fetching of four operands, two multiplications and the storage of the two results; the second is similar, the only difference being the presence of a subtraction instead of the multiplication in the first line. At the higher level, the same two examples can be measured with a simpler metrics: two multiplications the first, one subtraction and one multiplication the second. Let's suppose now that all the typical hardware devices potentially suitable for the above programs only implement addition and subtraction in their instruction sets; in this case the two programs are not equivalent in complexity,

since the first one requires $(amp + max_16bit_int - 2)$ operations while the second only max_16bit_int operations. If, on the opposite case, all the devices also contain multiplication in the instruction set, and multiplication is executed at an equal time as addition and subtraction, it is possible to further abstract the metrics used in the simulation and to write the two programs as:

```
x = op(input, amp);  
output = op(x, max_16bit_int);
```

At this level of abstraction the two programs are equivalent in terms of complexity and they cost two "operations"; in this way the possibility to measure the output of the program is lost, but the programming language is simplified since three different operators have been reduced to a single abstract one; the output of the operator is undetermined but its execution time and memory location on which it operates are known.

It is finally possible to imagine that, in the first case, the designer decides to simulate for multiplications separately, i.e. instead of replacing them with the many necessary additions to obtain the result (decomposition of the multiplication as required by the actual devices) he can count the multiplication separately from the rest; if he verifies, as in the first example, that the majority or the totality of the operations come from multiplications he may decide, if necessary, to introduce an hardwired multiplication in the final solution.

This example is quite trivial, but it covers all the basic blocks represented in Figure 7. Many more details concerning the abstraction method will be given in the following section, where the real case of the SAOL language will be considered. It is now important to say a few words in general concerning the way in which the above described features of imperative languages will be treated.

2.4.1. Data Types

Derivate data types constitute a composition of native types. An operation involving a derivate type is reduced to the native operations necessary to manage it. For instance a structure containing two numbers can constitute a new type of the language; operations involving the fields of the structure are identical to operations with numbers (and in fact the compiler takes care to eliminate the abstract type to restore the native data and operation).

2.4.2. Memory Allocation

Every declared variable belonging to a certain type (native or derivate) require a statically or dynamically allocated memory space, according to the rules of the language and to the type itself. Every memory space that is dynamically allocated in run-time through allocation functions or constructors must be also considered in a similar manner. Tracking allocated memory is specially important for dynamic simulation in function of time, as it will be described in the next subsections.

2.4.3. *Statements and Expressions*

All statements and expressions must be considered, and possibly they must be grouped in homogeneous class of operations, as shown in the above example. It is important to clarify that it is obviously not possible to precisely abstract the several operators of a programming language in function of all the actual hardware devices, but rather a reasonable trade-off between generality and complexity of the abstract model shall be found. The problem of complexity vector will be briefly discussed in subsection 2.4.5.

2.4.4. *Library Functions*

The abstraction of library functions constitutes the most difficult part of the modelization process. Modern languages easily standardize hundreds of different functions, not to say thousands, and they often constitute a very heterogeneous set of functionality. Of course, most of the library functions are implemented using the native data types and native operators (standard functions, in fact, are nothing else than well-defined derivative operators), and this could make possible a decomposition of each of them to a certain amount of elementary instructions. This can be considered as a possible approach to the problem, but it is also the case that some functions are so complex that cannot be decomposed easily or in a general form in elementary sequences of statement. In this case another approach is to consider the function as an operator in itself that can reasonably require an estimated or measured amount of computation on the target device.

2.4.5. *Approximation in the Simulations*

It already appears from what presented so far that the proposed approach introduces some degrees of approximation in the simulation process. The modelization can be made more or less detailed according to the specific needs, but in any case to have a meaningful and usable abstract model it will be necessary to "group" in the same class operations that could slightly differ in practice. This will be true in particular for the standard function libraries.

To better understand the meaning of this approximation, it is possible to view the set of virtual operators as a *complexity vector* C_v , composed by n elements, where n is of course the number of virtual operators. This vector defines an n -valued function, which is depending on the simulation and on time, and define the complexity of the simulated application. If only one programming language is considered and only one hardware device, the modelization can be precise; the language compiler for the hardware device generates the assembly code, which can be simplified grouping together all the instructions with exactly the same requirements; this will constitute an abstract machine model and every feature of the language can be mapped on the model as the compiler does.

When the model has to be extended to different, or theoretical, hardware devices, the situation changes. Let's suppose, for example, that device A executes addition and multiplication in one clock cycle with no pipeline, while device B has two stages of pipeline and multiplication has a latency of two cycles (i.e. its result is available two cycles after its decoding), addition of one cycle. To make a simulation that is

independent from the two devices, it is necessary to add up addition and multiplication in two different element of the complexity vector, so that the result can be re-mapped correctly in the two cases. If all existing devices only implement addition and multiplication it makes sense to keep separate the two operations. If instead all existing devices further implement e.g. the square root operator with a latency in the order from 15 to 18 cycles, and the sinus operator with a latency of 18 to 22 cycles, again an entry should be made in the vector for each operator; at the same time it comes to evidence that two "classes" of operators are present, the arithmetic ones (1 or 2 cycles) and the complex ones (tens of cycles).

In a design task, one may proceed to a more abstract simulation, counting arithmetic and complex operations. Then, after having obtained the first results, he may choose to further refine analysis or to stop at a certain point, according to these results of the simulation. If, on the other hand, different applications must be simulated to evaluate their complexity and select suitable devices, a higher level, approximated simulation could be the best solution, because it permits to better eliminate some differences, for instance the fact that some architectures could not be able to execute a sinus in pipeline with the square root, while other can.

Let's consider finally the real case of e.g. RISC devices, with tens of different instructions in the set, and complex languages, with hundreds of different features (even if library functions are theoretically standard, their execution time depends on the libraries provided by the compiler). Let's add to this the fact that different programmers could optimize code in slightly different ways using different features, and it becomes evident that a certain approximation in the simulation results must *a-priori* be considered, even if the modelization were absolutely precise, otherwise the complexity of the simulation would explode.

As a reference, a similar kind of approximation is claimed in the Mermaid simulator.

2.4.6. Final Remarks

The proposed method of abstract analysis does not aim only at a language simulator or an instruction set simulator, rather it is intended as a starting point for a virtual simulator of real-time platforms. In normal, efficient programming, especially for signal processing, most of the calculation is carried on through calls to suitable library functions, made available by every compiler toolset as implementation of the standardized libraries of the language or as extensions for multimedia support. The proposed approach takes then into specific account library functions and their complexity, thing of which no mention is made for other tools presented in literature and based on the exploitation of an abstract model (Mermaid and Zhu/Gajski); for this reason the resulting tool may perhaps introduce slightly rougher and more imprecise performance than reported in [39] or [40].

In this sense, it is useful to remark that the reference goal has been to achieve a precision in simulation that can be enough for a correct estimation of the workload, but at the same time in a context where unpredictable interaction and streaming information have the possibility to partially modify some aspects of the considered processing.

3. Dynamic Analysis of a Program Execution

3.1. *General Remarks*

If virtual analysis of an imperative programming language can be carried on using abstract or virtual criteria, the same cannot be said for the analysis of dynamic requirements. An application supposed to run in real-time must forcedly match its execution time against the wall time that the performance is supposed to last, and the first shall necessarily be less than the second. But this is unfortunately far from being enough. In fact, real-time multimedia programs have in most cases a time-varying behavior in function of time, and this is due essentially to interaction and to the evolution of their control parameters in function of time. A very meaningful example will be shown in the next section for the case of musical synthesis. This means that an overall time estimation is not meaningful at all, since real-time constraints must be assured in every single time interval of the performance; the granularity of the time interval can be imposed by both the operating system and/or the reaction delay that can be tolerated for the specific application.

An analysis of the application in function of time requires then either a simulation of the complete operating system (as done in SimOS and SimICS), where the system's scheduler has an exact knowledge of time, or the possibility to track the performance time at the level of the programming language and of the program execution. The first approach is feasible, but it imposes the platform to be a general computing machine running an operating system, and it is then not suitable for a more general approach based on e.g. DSPs or custom solutions.

The second approach requires the "time" variable to be visible at the programming language level, or at the program execution level (in the case of an interpreter), which may require a very hard task of code rework. In fact, the time variable may have a generic name, it may even be masked at the system-call level; the application may or may not have its own scheduler, regardless of the fact whether it is supposed to run as a plug-in to another real-time framework like, for instance, MPEG-4.

In any case, the detection and tracking of the time variable require direct manipulation of the application source code or of the code execution engine; one of these must be accessible when the simulation, like in this case, is based on a high level programming language and not on low level, non-portable assembly code or machine executable. This possibility to access the code may not exist, unless special requirements are specified before programming the code itself or the considered language include in its specification the evaluation of time, as it is the case for many SWSS languages.

3.2. *Analysis by Virtual Machines*

In some very popular cases, like JAVA or MS Windows-based programming environments, the time variable is often masked behind multimedia-oriented APIs or specific system calls; if the former is manifestly a case of non-standard solution, the latter is a typical case of layer of abstraction, discussed in the second chapter, where lower level information is hidden from the programming language (thing that,

in itself, does not exclude a-priori the use of a specific "time" variable). In the case of execution by virtual machines that are interpreted ones like in JAVA, to obtain the required dynamic analysis it is possible to implement the simulation framework directly in the interpreted machine. Going back to Figure 7, this implies the adoption of the JAVA machine instruction set as abstract set and then the abstraction model must be fixed to be a mapping of a real device into the JAVA virtual machine, or a subset of it.

Nothing refrains in any case, except the complexity of the task, to write an interpreter for any imperative language that is likely to be considered a candidate for an abstract simulator: the resulting virtual machine has that language, or a subset, as instruction set. This would permit to have access to the low level methods that deal with time, in order to control it at the simulation level.

3.3. *Dynamic Analysis and SA*

In the previous chapter SWSS languages and MPEG-4 SA have been introduced; almost all of them are based on rate-based variables and on a more or less standard description of a scheduler, which is defined with the language itself and is the base for each program execution. It is evident that SWSS toolsets are conceived for real-time execution of processing and audio applications, and this makes them particularly suitable for any kind of dynamic analysis. It is enough to correctly track the scheduler to be able to follow the time along the performance. In this sense an interpreter for an SWSS language is a perfect candidate for a dynamic analysis of an application, and this is why MPEG-4 SA has been considered the good toolset to define complexity of audio applications. In a context of relatively low resources like a PhD research it was fundamental to find a language where the implementation burden was not too high to develop and demonstrate concepts that, with a higher effort, can be theoretically extended to any other imperative language.

At the same time, the goal was to estimate complexity in function of time, as necessary for multimedia, and to be able to do that in real applications implemented through libraries and dedicated operators; this was not possible using another existing simulation tool, this requirement being never explicitly considered in previous works.

4. Case Study: Structured Audio Orchestra Language

This section presents the application of the proposed method for measuring program complexity to normative Structured Audio decoding, described through the Structured Audio Orchestra Language. It will be shown later in which form this method has been adopted in the MPEG-4 standard to define SA Levels and for SA Conformance testing; an exhaustive description of the technical details can be found in [82],[83]; here a comprehensive survey with a few significant examples will be presented.

The SA standard specifies no algorithms, 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; being SAOL essentially an imperative language, 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 (as usually done in standardized contexts).

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 score and on the runtime dynamic changes of the control parameters arriving via the stream and exposed fields of the BIFS scene. 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.

At this point it is evident that a virtual simulation model like that presented in section 3 can offer a much better solution. Since the implementation of a decoder can be software- and/or hardware-based, as mentioned before it is important to define the complexity vector in a way that it can be useful for the widest possible range of implementations. Moreover, the complexity vector must be available as a function of time. It has been explained in the previous section, and it will be demonstrated 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. Consequently, the complexity vector C_v must be measured as a discrete function:

$$C_v = C_v(t_i) \quad (2)$$

where t_i is a generic instant along the whole execution time axis and it is characterized by a suitable granularity that will be briefly discussed later.

4.1. Parameters for Complexity Analysis of SA Programs

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, trying to avoid the unpredictable overhead coming from a specific decoder solution. In the SA standard [69] some of the statements and core opcodes of SAOL are not specified in full details but only in terms of behavioral description (mainly when 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 (see sections 2.4.4 and 2.4.5).

The SA decoding process can roughly be split into two main parts: first of all it is necessary to compile the orchestra and instantiate the scheduler; then the control is released to the scheduler itself to perform the run-time sound synthesis or processing. The first step is run once at the beginning, and normally it has no strict constraints on execution time or optimization rules. Moreover, the output of this

phase is a static memory structure or a target machine code, which can be roughly considered as related to the orchestra code and which does not change during the real-time performance of the second phase³. In conclusion, for complexity analysis purposes, it is reasonable to disregard the start-up phase and take into account only the run-time execution. The same assumption is made for the execution of some of the core opcodes: since they are always static functions, their initialization is not considered unless it affects meaningfully the amount of allocated memory. The main features of the SA run-time phase will be now considered, and factors influencing complexity will be discussed in relation to the definition of the complexity vector.

4.1.1. *Variables and Tables*

The first step of each execution is the allocation of global variables. Since in Structured Audio a unique numeric format is considered as normative, i.e. single precision 32-bit floating point, for each variable it is necessary to consider an allocated space of four bytes. The same occurs for global tables, allocated at the *orchestra startup* (the default initialization of every performance), and then another four bytes are necessary for each sample of each table. Global variables and tables are considered to be allocated in a single copy over the whole duration of the decoding process. Instead, in the case of local variables and tables, declared inside an instrument code, one additional allocation must be considered for each instrument instance that is active at time t_i . All variables in SAOL are static.

4.1.2. *Memory Accesses*

Memory accesses represent a critical point in complexity estimations. On one side memory bandwidth is in most cases a bottleneck for the execution of programs that allocate large amounts of RAM, but on the other side an access to a variable or to a table can result in the physical access to different locations of the target architecture, e.g. registers, cache memories, static or dynamic data memories, swap space. This means that memory accesses normally have a quite different impact on execution according to the specific memory management technique and to the characteristics of the platform. Moreover, some memory accesses in SA programs cannot be exactly counted if they are in non-normative parts: a typical example is interpolation, for which the exact order, and then the number of fetched data, is not specified. Because of all these problems, memory accesses cannot be analyzed separately from corresponding operators: they are supposed to be only a direct consequence of table operations, interpolations, and so on, and measured implicitly with them.

In the first version of the simulator based on SA, the one used for MPEG-4 Conformance, a memory access has always been considered a measured access to the fast cache memory. In chapter 7 it will be described how the tool has been enhanced with a cache simulator to improve performances.

³ Note that for SAOL, since the source code is received streaming, it is necessary to compile code as a first phase of the execution of the program. This is normally not required by other languages. In the case of JAVA applets, the already compiled byte-code is downloaded.

4.1.3. Summing Buses

The *send* and *route* statements in the global block of the SAOL orchestra provide a summing bus mechanism for the active instrument instances. This means that if the outputs of N instrument instances are summed on a bus, each sample on that bus is generated by N-1 additions. A bus allocation also implies a static memory allocation, which is proportional to the bus width in channels.

4.1.4. Statements and Expressions

A SAOL algorithm is implemented by statements, expressions and core opcode calls. The programmer can also exploit user-defined opcodes, but these latter are easily merged into the main instrument blocks, as actually the MPEG-4 reference software does.

Among the statements, only *spatialize* introduce a heavy computation load: in the case of *spatialize*, the algorithm is non-normative and the required effect can be produced using several different techniques ([84],[87]): this statement cannot be classified in a common set with other operations. All the other statements (assignments, *while* loops, *if-then-else* branches, simple instructions for output and instance control) are much simpler and can be easily counted with other ordinary operations.

Concerning expressions, common logic and arithmetic operators defined in SAOL are supported nowadays by every architecture; while the majority of them can be added up together, multiplication and division can be considered separately, because of their complexity in terms of required logic.

4.1.5. Core opcodes

Core opcodes constitute the main issue in calculating the decoding complexity of SA; in fact, their number and their wide range of functionality deserve an attentive and detailed analysis. Like variables, all the opcodes in SAOL are static and then they are characterized by an internal state throughout execution.

The SA Standard [69] defines 105 core opcodes, which are grouped for convenience in the following groups: mathematical functions, pitch converters, table operations, signal generators, noise generators, filters and spectral analysis, gain control, sample conversion, delays and effects.

For a complexity analysis, the same core opcodes can be divided into four main groups [83]:

- 1) core opcodes for mathematical operations, which have a precise output but can be implemented in many different ways, e.g. *exp*, *log*, *sqrt*, *sin*, *cos*, etc.;
- 2) core opcodes for which the normative text specify the exact algorithm through elementary statements and expressions, e.g. *frac*, *abs*, *min*, *max*, pitch converters etc.;
- 3) core opcodes for which the implementation is non-normative and the solution is left open to the implementers, e.g. *hipass*, *lopass*, effects and others;

- 4) core opcodes like table *oscillators*, *tableread*, etc., which can be reduced to a mostly normative procedure like in case 2), but containing a possibly non-normative interpolation⁴.

In principle, each opcode belonging to group 1 should be counted independently, since it requires a different algorithm that could be available or not in a specific hardware instruction set, and with a different latency in each specific case. The same consideration is valid for opcodes belonging to group 3 and for non-linear interpolation: in this case each platform can be more or less optimized for such operations, and efficiency is strongly influenced by the programming platform and tools used for development ([11], [88]).

For the opcodes in group 2 and normative parts of the opcodes in group 4 it is not difficult to describe them by means of a symbolic language, composed by

- the *if* statement
- some arithmetic and logic operations of SAOL
- a few mathematical core opcodes, and
- *interpolation*, *multiply-and-accumulate*, *round*.

In such a way, following the standard text, each opcode belonging to group 2 and 4 can be described by a precise algorithm, which is a sequence of instructions of the defined symbolic language. A few examples are reported here to better explain the concept:

```
frac(x):    val = x - int(x);
```

In the case of the *frac* opcode the result *val* is obtained by an integer operator and a subtraction.

```
ampdb(x):  val = exp((x-90)*0.05);
```

In the case of the *ampdb* opcode the result *val* (amplitude of a decibel-valued parameter) is obtained by a subtraction, a multiplication and an exponential. Tests have been omitted for clarity. Following some practical and common rules a division by 20 has been considered a multiplication by 0.05.

As a last example, a more complex opcode will be analyzed: *doscil*. The *doscil* core opcode plays back a sample wavetable once, with no frequency control or looping, as it is possible to do with other similar opcodes providing oscillators. All core opcodes in SAOL are static functions, i.e. they preserve their state throughout the execution of a particular instance of the instrument. In particular, the *doscil* opcode preserves the internal phase, while the table is passed as an argument. The normative execution states as follows:

"...the internal phase shall be incremented by TSR/SR, where TSR is the sampling rate of the table and SR is the orchestra sampling rate...If the internal phase is greater than 1 the return value shall be 0. Otherwise the return value shall be the

⁴ In SA the interpolation factor depends on the global *interp* parameter; it is linear if *interp* is equal to zero, otherwise it shall be a "higher-quality method than linear interpolation", if *interp* is equal to one.

value of sample x in the wavetable, where x is given by the product of the table length times the internal phase. If x is not an integer, then the value shall be interpolated by the nearby table values..." [71].

First of all, it is assumed that the internal state of the opcode will also contain the increment value TSR/SR, since it makes no sense to calculate it at every call: this parameter cannot change. Also the length of the table does not change and it can be reasonably considered as a constant parameter of the internal state. The real operations are then:

- 1) the increment of the internal phase (1 addition)
- 2) the multiplication between the phase and the table length to calculate x (1 multiplication)
- 3) the test on x to check if it is an integer.
- 4) the eventual execution of the interpolation
- 5) the return of the result.

In form of code this can be written as:

```
ph = ph + inc;
x = ph * length;
if(x - int(x) == 0)
    return table(x);
else
    return(interp(table, x));
```

In this example two cases are possible: an addition, a multiplication, a subtraction and a test, plus an *int* operator (integer part of a number) are always executed. Then either an interpolation is calculated or a value is immediately returned. This example is meaningful because it shows how it is not possible to evaluate the complexity of the program without executing it, thing that is intuitive. Moreover, it must be noticed that *int* and *interp* (for interpolation, the exact order is not specified if the high quality option is selected for the orchestra) operators cannot be further decomposed in simple operations, since the former is often based on language libraries while the latter is a case of functionality whose implementation is left open to implementers (choice that in this case is extremely unlucky).

In a similar manner all the opcodes in group 2 and 4, and even a few from group 1, can be translated into a precise sequence of instructions whose operators may be easily counted in the complexity vector.

4.2. The SA-based Abstract Simulator

The proposed abstract simulation tool for SA will be now presented briefly; the principles explained earlier has been included, as a first experiment framework, in the actual Structured Audio MPEG reference decoder (saolc, [89]); 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 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 srates 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 gives the specific parameter values over B_i 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 taken immediately before the particular output.

4.2.1. Reconfigurability of the Simulator

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 testing the first 11 columns have entries different from zero. For each meaningful feature of SA, as from subsections 4.1.1 to 4.1.5, it is possible to specify how many units of each parameter are necessary for the execution. For instance, for each core opcode it is possible to specify how many of each parameter is used for one call: some parameters like interpolation are only incremented if they really occur during the execution as shown for the *doscil* example. Additional lines are added to the bottom of the file to specify different options for opcodes that can have branches. Thanks to this flexible mechanism, the operations necessary for each feature can easily be configured by editing the input file; in theory a great number of architectural solution can be modeled in this way.

4.2.2. Experimental Measures

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 modeling (bass), 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 [63], [68]), multimedia audio literature ([84], [90]), or industry. The scores used for simulations were always rather complex, with high degree of polyphony (for synthesis) and changes in control parameters, to force different subtrees in the programs to be executed.

A typical output for interpolations in a classic piano piece is shown in Figure 9. Inside the orchestra, the several piano instances, used to generate sound, are fed into a single instance of a processing instrument that mimics reverberation of a large hall. The score is characterized by sudden variations in polyphony. The horizontal axis is the score time of the performance.

It is obvious from this example that the mean value, reported by the dotted line, is not of any importance to guarantee a real-time performance: 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, being non

optimized, in this case is very far from a real-time performance even on a general-purpose rather fast (600 MHz) computer. Experimental results also show that a time granularity of 1 second is enough and that a fast control rate is only necessary to achieve acceptable reaction times.

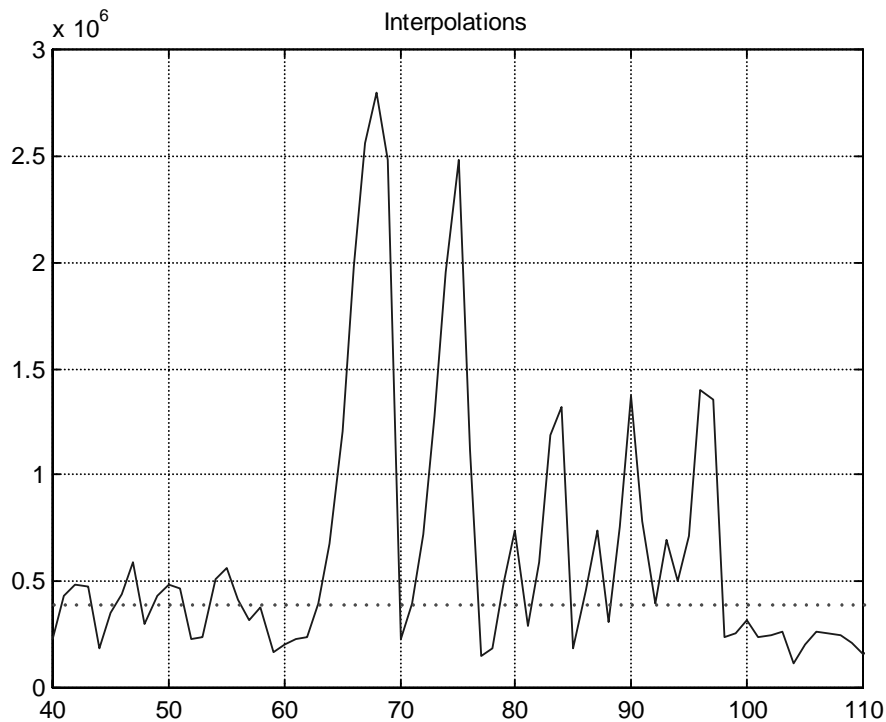


Figure 9 Number of interpolations in "Claire de lune" by C. Debussy. 70 seconds of score time are extracted from the complete profiling. Horizontal axis is time in score seconds; the dotted line is the mean value along the complete decoding (about 300 seconds)

From the same sequence it is interesting to consider also the results reported in Figure 10 and Figure 11. In the first one the same time window of Figure 9 is considered and the number of executed mathematical methods (exponentiations, logarithms, etc.) in a second of score file is plotted. The curve has a shape similar to that of Figure 9, but it is possible to notice that here peaks are approximately four times the mean value, while there they were six times the mean value. This can be explained considering that mathematical methods are used in both the "constant" processing instance and in the time-variant piano instances, while interpolations are only present in the latter case; for this reason interpolations have a higher variance, since they do not have any continuous component. At this point it is straightforward to understand the nearly constant diagram reported in Figure 11, where the number of MACs (multiply-and-accumulate) is plotted. Since the reverberation instance is static throughout the whole performance, as it often happens for this kind of processing, and since MACs are not present in piano instances, the curve present only periodical very small "glitches".

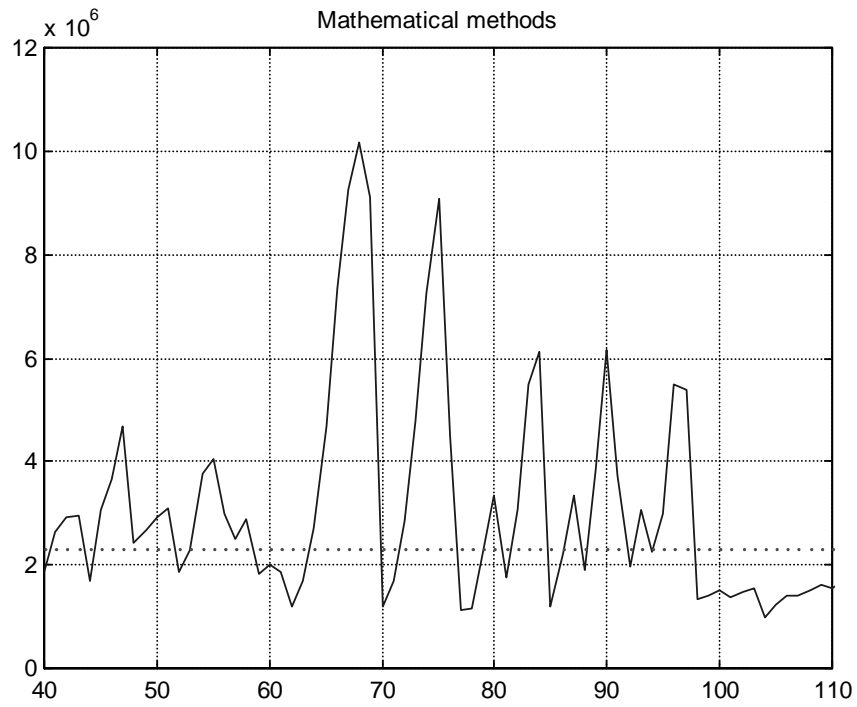


Figure 10 Number of mathematical methods in "Claire de lune" by C. Debussy. Horizontal axis is time in score seconds; the dotted line is the mean value along the complete decoding

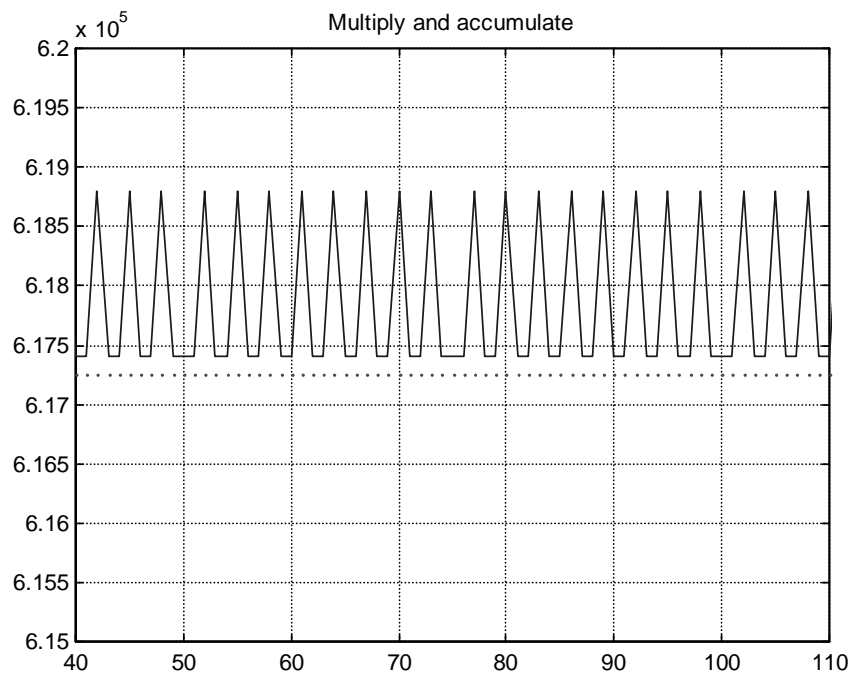


Figure 11 Number of MACs in "Claire de lune" by C. Debussy. Horizontal axis is time in score seconds; the dotted line is the mean value along the complete decoding

This is due to the limitations of precision of the 32-bit floating-point format that must be used for the time variable and its corresponding increment, thing that causes the second of performance to contain periodically one control cycle more or less. The parameters that are plotted in Figures 9 to 11 are grouped as they are for the MPEG-4 SA Conformance testing, as it will be explained in the next and last section of this chapter.

5. MPEG-4 SA Conformance Test

5.1. Complexity Vectors Again

It was explained in section 4 that some of the SAOL functionality is not measurable on an operations-per-second basis, since some of the decoding algorithms for core opcodes and statements are not specified and left open to the implementers; among them, some like interpolation, spatialization, effects and filters can heavily affect allocated memory and computational complexity of a specific program; on the other hand, some hardware platforms could implement specific solutions for these features, some other not. In conclusion, it is necessary to follow some macro-oriented criteria, which are able to make abstraction of the open issues, and calculate them in separate elements of the complexity vector. The choice of an appropriate vector is fundamental at this point.

As remarked in subsection 4.1.5, several core opcodes in SAOL should be counted separately: for instance, even if the output of many mathematical methods like e.g. sin, cos, sqrt, etc. is known with precision, the actual implementation is of course not specified, and potentially they could be implemented in many different ways; at the same time, it is reasonable to assume that many of these cases are in some way similar, in the sense that it is common that a specific decoder solution will choose the same approach to calculate e.g. a sinus or a tangent.

Another issue concerns the precise goal of the simulation, already hinted in subsection 2.4.5; if the SA analysis must be used for implementation purposes, as it will be shown in Chapter 5, it is intuitive that a good "resolution" in the complexity vector will result in a better definition of the sw and/or hw architecture. On the contrary, when standard complexity levels must be set [83], the complexity vector must not be too long, because this could hardly overspecify the decoder: this happens e.g. when the SAOL functionality is not completely exploited and a software solution on a general purpose processor is adopted. This consideration strongly influenced the choice of the vector for MPEG-4 Conformance, in the sense that the number of parameters had to be kept to a minimum, while maintaining a meaningful precision in the simulation.

5.2. SA Conformance Test

In the MPEG-4 Level definitions an 11-elements vector has been standardized to measure complexity of an SA stream [82]; the elements are: *Total opcode calls*, *Floating-point operations*, *Multiplications*, *Tests*, *Math methods*, *Noise generators*, *Interpolations*, *Multiply-and-accumulate*, *Filters*, *Effects*, *Allocated memory*. In this case, studying typical DSP instruction sets ([11], [88]) and multimedia libraries,

several simplifications have been introduced into the complexity vector. For instance, all noise generators are reduced to a single parameter (with different weight functions): it is assumed that each of them does not have a cost meaningfully different from the others, and then the worst case does not differ in a substantial way from the mean case; for mathematical functions the same assumption is made as for noise generators, and division is added up to this element. Effects contains reverb, chorus, flange and the spatialize statement. And so on. Every SA program is mapped into a vector C_{vk} by precise criteria (see [82]), while at the same time each specific decoder solution can have the Level vectors mapped into its actual instruction set.

Table 3 Algorithmic Synthesis Complexity Values for Levels

Parameter	Low-Complexity	Medium-Complexity	High-Complexity
Total opcode calls	2M	8M	16M
Floating-point ops	12M	24M	60M
Multiplications	8M	16M	40M
Tests	2M	8M	16M
Math methods	4M	16M	16M
Noise generators	0.1 M	1M	1M
Interpolations	0.6 M	4M	12M
Multiply-and-add	2M	4M	12M
Filters	0.6M	2M	4M
Effects	0.2M	1M	2M
Allocated memory	64k	8M	16M

Decoder conformance concerning computation capabilities shall be tested against the definition of high, medium or low computational complexity provided in Table 3. The decoder supporting one of the three computational levels shall be able to decode bitstreams for which the associated complexity vector is, for each second of the performance, below the reference vector of the corresponding Level. Rare exceptions are admitted, as explained in the following. The decoding time of each second of the performance shall be executed in a time less or equal to a wall clock second. Five bitstreams are provided by ISO/IEC with their corresponding complexity vectors in function of time, in order to help the correct evaluation of the computational complexity supported by the specific decoder. As an example, a graphical representation of the Conformance test procedure for Structured Audio is represented in Figure 12 and Figure 13; it is reported the case of the test sequence called SY008, which is a case of several instruments generated by different synthesis models and playing at the same time.

The first diagram in Figure 12 is a curve in function of time for floating-point operations in function of time, in a way similar to previous diagrams. Here the

figures for Level 1 (Low-Complexity) and Level 2 (Medium-Complexity) are also plotted. It can be noticed that, while the mean over the whole sequence is below the Level 1 requirements, several seconds of the performance are well above this Level, and then the sequence, independently from other parameters, requires a Level 2 decoder for its decoding process. Figure 13 plots a cross-section along the time axis at the instant $t = 36$ seconds. The x-axis in this case is the complexity vector. Again it is noticeable that in this instant (and in many others) the sequence requires a Level 2 decoder (for 6 parameters of the vector).

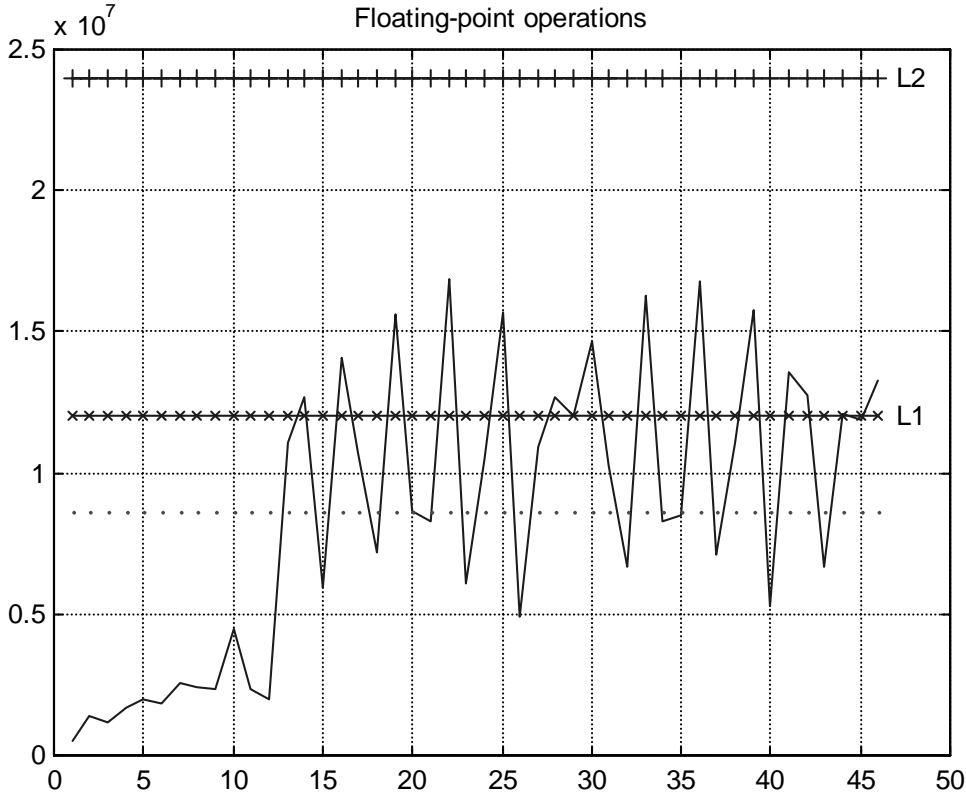


Figure 12 Number of interpolations in "SY008" test sequence. 46 seconds of score time contains several instances of different instrument plus some signal processing. Horizontal axis is time in score seconds; the dotted line is the mean value along the complete decoding. The x-line is the number of floating-point operations required for Level 1; the +line is the number of floating-point operations required for Level 2. It is evident that a Level 2 decoder is necessary to decode this sequence.

5.2.1. Approximation of the decoder

In order to deal with the necessary approximation that was justified in the previous analysis, the SA Conformance standard insert a clause that states how to deal with it. In synthesis, it is not the case, in order to conform to one of the complexity levels in the table, that a decoder must provide the amount of computation shown in the same table for every element of the complexity vector at the same time. Rather, a conforming decoder must be able to normatively decode any bitstream

that is measured with the standard profiling tool as requiring no more than that amount of computation. In fact, it is not possible (unless with a precise but really not meaningful premeditation) to produce a sequence that fit exactly into one of the required vectors, which are instead specified on the base of several different profilings for many sequences. A decoder providing the exact amount of computation for all the parameters at the same time could result in some way overspecified. This is why it was important, in this case, to have a rather short vector with a greater degree of approximation in the grouping of classes of instructions: a platform implementing optimized, but not flexible, solutions for some classes of operations could be required of too much overspecification, obtaining a result which could be the opposite of the goal of optimization. In MPEG-4 the complexity vector is very near to the minimum number of elements to preserve a consistent evaluation of complexity.

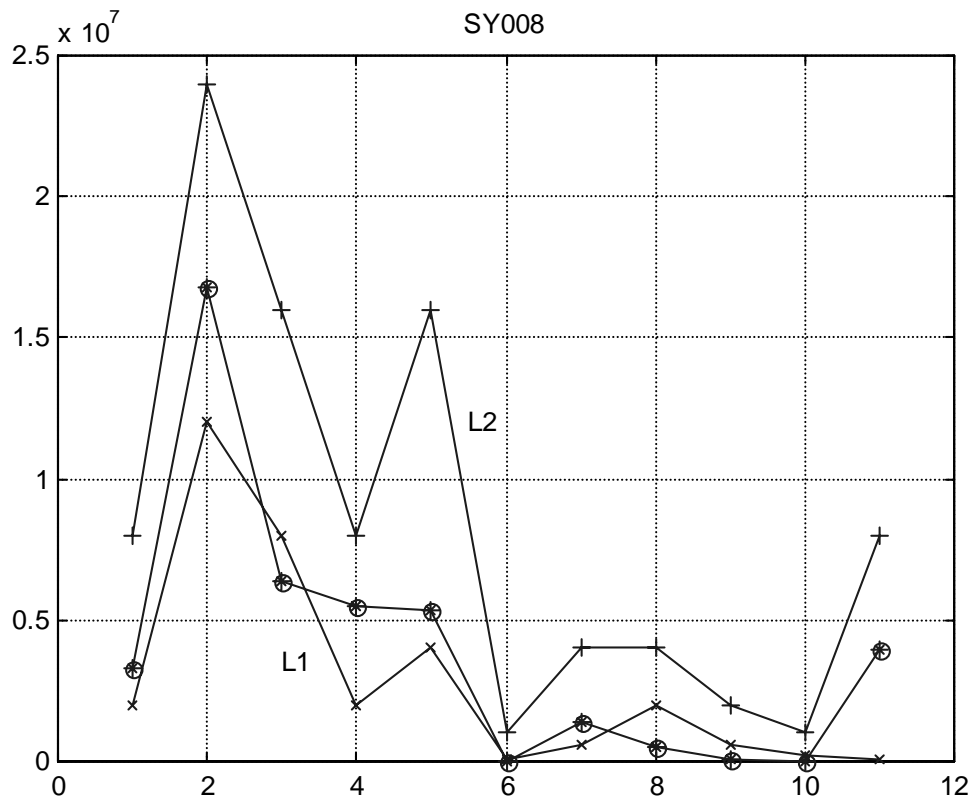


Figure 13 Cross-section along the time axis at $t=36s$ for the test sequence SY008 (star-line). +-line represents the complexity vector for Level 2, while x-line represents the complexity vector for Level 1. X-axis is the complexity vector.

Moreover, when a conforming decoder is implemented with static optimization, it is usually possible to decode a bitstream that contains a certain number of operations per second as measured with the profiling tool by actually using fewer operations per second than this; actually the calculation of the complexity vector is made in a platform independent way on the basis of the normative SA text and this could contain redundancies. Put another way, there are two ways to increase the amount

of computation that a Structured Audio decoder can provide. On one hand, it can run on more powerful hardware. On the other, it can implement more powerful static optimization and thereby provide more effective computation on the same hardware. The measurements shown in the table should be taken as referencing a completely not optimized, but overload-free, SA implementation, and so high complexity decoding can actually be realized on a hardware platform without exactly so much native computational power.

Each programmer should be able to "map" these platform-independent formal vectors into his own implementation using the detailed description contained in annex to the text, in order to calculate his actual complexity vectors and estimate the computational power that his platform will need.

Preliminary experiments made during the MPEG-4 standardization showed acceptable results on general purpose processors even with this quite reduced complexity vector. In spite of the 30-40% of approximation, in the mean, to measured decoding times, the method has demonstrated a sufficient capability to track the order of magnitude of required computation; this is a remarkable result in an almost unexplored domain and with such a simple solution. All the results concerning simulations, based on SAINT instead of saolc, will be presented for coherency and continuity in Chapter 7.

5.2.2. Saturation in computational resources

Concerning the potential problem of saturation in the available resources, developers are also advised that algorithmic synthetic bitstreams often require "burst" processing, where small time portions of the bitstream require considerable amount of processing power. A clear example has been reported in the classical piano score described above, even if in that case peak values had a consistent duration in terms of time; in other cases this duration can be much shorter and more frequent in its repetitions, for example at the first step of an instance execution.

In situations such as this one, where the requirements of a bitstream exceed in rare spikes of time⁵ the complexity of a particular Level, programmers are encouraged to implement a procedure for graceful degradation of decoding. Many such techniques exist, such as voice stealing [63], but they are non-normative and left up to the implementer. Priority bits associated to each new instance and to control events in the score are also provided to support subtle degradation methods.

Such techniques can also result in great benefit in other cases when burst of performance can be present in a way that cannot be foreseen by the system designer and by the content provider. Consider for instance the case of high degrees of user interaction, which could introduce a considerable amount of new instances and modification of control parameters. This case can hardly affect the overall schedulability of the system.

⁵ since the granularity of the profiling is 1 second it can be assumed that a "spike" is in any case not longer than that.

5.3. Extensions to AudioBIFS Conformance Test

The test of AudioBIFS nodes can be carried on extending the basic principles for Structured Audio in two ways, for the AudioFX node and for the complete scene respectively.

5.3.1. The AudioFX node

The extension for the AudioFX node is straightforward. Indeed, it was already explained (see section 2.2.2 of Chapter 3) how this node permits the inclusion of processing programs written in SAOL in the middle of the BIFS scene. It is then not difficult to exploit the same criteria used for the SA audio object and to define some new vectors that will be used for Audio scenes (considering that in this case the processing will be used almost always for processing, and not for synthesis). Table 4 presents Levels for the AudioFX node as they are defined in the MPEG-4 Conformance.

Table 4 AudioFX node Complexity Values for Levels

Parameter	Very Low-Complexity	Low-Complexity	Medium-Complexity	High-Complexity
Total opcode calls	1M	1M	4M	8M
Floating-point ops	0	4M	12M	20M
Multiplications	0	2M	8M	16M
Tests	0	1M	4M	8M
Math methods	0	2M	6M	12M
Noise generators	0	50k	0.2M	0.5M
Interpolations	0	0.3M	1.2M	2M
Multiply-and-add	2M	2M	4M	8M
Filters	0.2M	0.2M	1M	4M
Effects	96k	96k	0.4M	2M
Allocated memory	96k	96k	1M	16M

In the context for AudioFX an additional remark is that, thanks to the flexibility of the definition of complexity, in the case of the Very-Low-Complexity Level it is possible to define a subset of the SA toolset without having to define a new Profile⁶ for it [91]; in this Level, the elements of the vector different from 0 permit the construction of an effect box without requiring a full compiler for the complete and general purpose language but only a quite reduced one: in fact, all these parameters are present only in specific core opcodes for processing and then a program can be written only as a chain of processing blocks.

⁶ In an international standard like MPEG, a Profile is a subset of the complete functionality that represents a meaningful class of applications implementable by this subset only.

5.3.2. Other AudioBIFS nodes

Other AudioBIFS nodes specified in version 1 of the MPEG-4 Systems standard introduce basic functionality for a spatialized Audio composition. A very careful analysis of this functionality has been carried on in parallel with that of Structured Audio ([56],[57]). In this case it was not possible to implement a tool for systematic analysis of the BIFS sequences because a stable and clean BIFS decoder including Audio nodes was not available at the moment of conformance standardization and of the Level definitions. The same approach, based on a macro-oriented criterion, to analyze classes of operations has been anyway extended in a theoretical way, thing that is not too complex given the limited subset of Audio nodes. A complexity vector composed by a few elements (five) has been defined in [93]; values for these five parameters (*scene width, scene depth, allocated buffer memory, computational power for SR conversion and mixing, number of spatialized objects*) are used to define complexity Levels of the BIFS Audio Profile (the subset including all the AudioBIFS nodes, specific for Audio only applications) together with the corresponding vector for the AudioFX nodes, as described above.

Two additional parameters have been introduced to better define complexity Levels that cannot be mapped into precise requirements but deal with the overall QoS: *reaction times* for control parameters (that in SA are explicitly programmed through the control frequency) and a requirement for click-free cross-fading in mixing and switching. The Conformance test procedure for AudioBIFS is also specified in [82].

CHAPTER 5. A PORTABLE VIRTUAL APPROACH FOR DIGITAL SIGNAL PROCESSING: SAINT

After having introduced the principles developed for abstract, platform-independent analysis of applications written in SAOL, this chapter will be dedicated to the implementation of a new SA decoder, inspired by the same language analysis and abstract simulation of typical applications in function of time. First of all, some further SA decoding issues will be analyzed, in particular processing by blocks. After that, the new Structured Audio INTERpreter (SAINT) will be introduced: being it based on a virtual DSP architecture, general features of this new tool will be presented through the definition of the instruction set and the description of the memory structures. Report on the implementation and the experimental results will be completed with some comparisons with other solutions currently available for the decoding of SA.

1. MPEG-4 SA Decoding Issues

In the previous chapter it has been extensively described how SA typical applications can be simulated in order to extract information of the associated complexity; this information is also useful to design efficient decoders. Core functionality can be separated in this way from less used one, and this offers a consistent basic set for a, so far, instruction set of a hypothetical SA-machine. Another interesting issue in SA decoding is the possibility of an execution block-by-block (from now on b-b-b) of processing instructions.

1.1. Execution Block-by-Block

After a short survey of some SWSS tools, it was decided to dedicate a particular regard to the study of the possibility of an execution b-b-b of the block of SAOL code at *srate*, without altering the output of the s-b-s standard language definition. In fact, efficiency of a block-based execution over a sample based one has been previously proven in literature ([95]). This is not difficult to argue, especially if an interpreted language decoder is implemented. Actually, each instruction to be executed brings with it a predictable overhead, which depends on how many instructions of the target machine are necessary to decode and execute one virtual (interpreted) instruction. If several samples are simultaneously processed by the same instruction, then the overhead is partially subdivided among them; partially and not completely anyway, since a control cycle to loop over the samples of the block must be inserted in addition to pure processing (because the block size is programmable).

The decrease in the amount of overhead due to b-b-b execution is limited in practice by several factors. First of all, the time gained has an inverse exponential law, and then, beyond a certain point, a doubling in the block size does not provide anymore a meaningful advantage. Moreover, a large processing block implies a long latency

in output: this is of no importance for an offline playback, but it requires a reasonable trade-off in a real-time, interactive scenario. Finally, a remarkable increase in the size of each allocated variable, imposed by block processing, rapidly takes to a relevant overload of the data cache of the device, thing that at a certain point leads to an inversion in the decrease of decoding time for an increase of the block size.

Because of these three factors, it is not wise and effective to let the block size grow beyond a reasonable value that can slightly change according to sampling rate (but consider that better quality audio also requires shorter reaction times). It will be shown in the experimental results that block sizes between 50 and 200 represent often the best solution.

1.2. Feedback Analysis

In SA what can prevent from executing b-b-b is the presence of an explicit feedback in the code. By explicit feedback it is intended here a feedback programmed using more than one line of SAOL code, while an implicit one is for instance in the *iir* core opcode, where the feedback is embedded in the function itself. Explicit feedbacks have been detected by a simple flow-graph analysis in a few situations. The most obvious is when an audio variable, the only one that can prevent b-b-b execution [95], is assigned to a new value after its first use.

For instance consider the following SAOL code:

```
bridge = lopass(outr, 1000);
tablewrite(wg, b1, bridge);
outtr = tableread(wg,b4);
outbus(my_bus, outr);
```

It is evident that the *lopass* filter cannot be executed on a block of samples of *outtr*, since this variable is modified afterwards by the content of table *wg*.

The second case of feedback is a multiple access at *srata* to allocated memory structures when there are some writings in the code that modify such structures; this happens:

1. with the anomalous *fracdelay* core opcode, which has an object-oriented concept, with different methods to access a delay line;
2. when a table is written through *tablewrite* (a direct indexed access to a table is not allowed in SAOL, it is imposed to pass through an operator).

Finally, a last case is the *while* statement, because it intrinsically induces a feedback from its body to its guard expression.

These four cases have to be detected and treated in a special way, while all the rest of the code can be executed on a possibly large b-b-b basis. As mentioned, "large" is acceptable as far as the introduced delay in the real-time synchronization of the complete MPEG-4 BIFS scene is tolerable, since in this case a drawback for latency is present exactly as in the case of an fft-based processing.

2. Structure of the SAINT Software Tool

2.1. *Towards a Virtual DSP Architecture*

Besides the study of complexity and the definition of Levels for MPEG-4, the complexity analysis of SAOL was also conceived as a mean to provide a systematic approach to real SA decoding and SA-based applications: this means that, given a platform (actual or virtual), typical decoding can be mapped on it in order to detect critical implementation issues, or to design ad-hoc enhancements to speed-up the execution and lighten the main processor, and so on.

The flexibility of the profiling tool described in the previous chapter permits to easily try different solutions, refining systematically the suitable complexity vector and the model of decomposition of each SA feature into simpler and less redundant macroinstructions. In the end, the two most interesting results of the described analysis deal with core opcode implementations and b-b-b execution. En passant, it is useful to remember that in SAOL the defined standard core opcodes are 105, but a careful analysis of them all, validated by the virtual simulation to verify efficiency, revealed that the number of "core functions" necessary to describe and optimise them is much smaller, nearly the half.

The statistically achieved results reported so far on function utilizations and on feedback show 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 DSP*, based on a vectorial instruction set [11],[97]. In particular, the virtual profiling revealed fundamental characteristics of the required instruction set, helping to define it as an extension of a common set of DSP instructions; the feedback analysis (and current trends in multimedia devices) demonstrated that it makes sense to implement this DSP architecture as a vectorial one.

The first step in the direction of the virtual DSP design was the concept of an optimized decoder, specific for processing and common synthesis applications on superscalar multimedia processors. While extensive details about the instruction set will be given in the Appendix A, the general architecture of the proposed software machine will be now described.

2.2. *The SA Decoder*

The SA decoder denominated SAINT (for Structured Audio INTERpreter) aims at two main objectives: first of all its design is oriented towards state-of-the-art DSPs, processors and multimedia processors, in order to conceive an easily portable tool that matches at best the parallelism that can be exploited in many of these hardware devices ([11],[95]); the second goal is the development of an efficient interpreter with a straightforward structure, in order to limit the overhead due to instruction interpretation in an embedded interactive application.

2.2.1. *SAINT and parallelism*

Concerning the first issue, parallelism, SA intrinsically provides two exploitable sources of parallel computation. The first case is a parallelism at the data level, that can be exploited when it is possible to work on vectors of data, as described in

subsection 1.2. The second case is a parallelism at the instruction level, but only when different instances of the same instrument are active: in this situation the same block of code (the code of the instrument) is executed for the several instances and then this precisely induces a SIMD (single instruction on multiple data) type of parallel architecture.

The statistical analysis described earlier invited to concentrate on the data level parallelism, which is almost always present, easy to exploit and easy to port on different platforms: in fact, all of the modern VLIW and SIMD architectures permit good speed-up factors for this kind of parallelism. Instead multiple instances are difficult to predict and variable in their amount (they strongly depend on the score). The intrinsic possibility of exploitation of data parallelism can represent the ideal hook in order to investigate how SA can be more effective than other general-purpose languages like C, C++ or JAVA, making it the reference language for any efficient audio application.

The final decision was then to design a virtual architecture able to execute the instructions necessary to decode an SA program in a vectorial form, with a variable length of the vector, from 1 for the execution of s-b-s code blocks, to N, where normally $N = B_i$ as defined in (1) in Chapter 3.

The other main objective of SAINT is to preserve the simplicity and effectiveness of the execution engine. The typical architecture of an interpreter has been modified in several aspects, in order to further limit the overhead and preserve a good portability. The first task was to divide the complete decoding in two well-separated layers: the scheduler/decoder layer and the instruction layer. The main reason for that is to be able to split the complete process into two independently executable parts, the compiler/controller 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 execute the intensive processing possibly in the same CPU, but with a comparable effectiveness in a separate co-processor, single or even distributed; this is achieved through a sequence of method calls and monodirectional data flow, after a specific resource allocation i.e., allocation of the native methods and of the program code. This concept influences the software structure in its main features.

2.2.2. *The SAINT compiler*

The compiler/controller task is conceived as a transcoder from the SAOL code to an intermediate format, which is forwarded to the computation engine; the transcoder translates the core opcodes and statements into the appropriate short sequence of instructions that are interpreted by the execution unit. While doing that, the SAOL *compiler* is also able to break all the nested calls, theoretically infinite in allowed levels; the vectors of values returned by each operator are stored in intermediate registers according to their rate. In core opcode calls this permits to avoid waste of time in useless evaluation functions; this happens when the actual rate of the parameter is lower than the rate allowed by the formal opcode definition, thing that is allowed in SAOL. The generated blocks of code, for instruments and opcodes, are additionally split into three different blocks, according to the rate of the statements to be executed.

For instance, let's consider the SAOL example of the following line of code:

```
outbus(bus, loscil(tmap[no], cpsmidi(note), cpsmidi(bass), loop, len)*amp);
```

where the output on the `bus` bus is calculated by the oscillation over table `tmap` (whose base frequency is the equivalent in cycles per second of the midi note `bass`) at a new frequency `cpsmidi(note)`. *loscil* is indeed very similar to the already described *doscil*; the two additional parameters `loop` and `len` say to the `loscil` opcode how to loop over the table, i.e. where to start and end repeating a part of the table to provide a longer sound, when needed: this normally correspond to the last part of the table. The result of the oscillation, multiplied by the amplification coefficient, is sent to the output bus. *Italic font* represents here the opcodes potentially executed only at the initialization rate.

The listing in Figure 14 gives a pseudo-code description of the execution with a typical interpreter structure for the above code. Indentation means a nested call. *Eval_* is a generic prefix used to identify an evaluator function of a statement or an operator; in particular *Eval_var* is a fetching of a value from memory, i.e. a "load".

In Figure 15 the execution of the same line of SAOL code is instead described with the virtual DSP approach. Again, *italic font* is used to emphasize instructions at the initialization rate. The *cpsmidi* core opcode has been explicitly decomposed into macroinstructions reproducing its functionality:

$$\text{res} = t * 2^{(x-69)/12}$$

being `t` the global tuning frequency.

Nested calls have been broken using intermediate registers. For sake of clarity, some parts of the code have been omitted, and in particular the second *cpsmidi* opcode conversion, the lines dedicated to parameter checking and the variable evaluators for each line, i.e. the memory or register accesses.

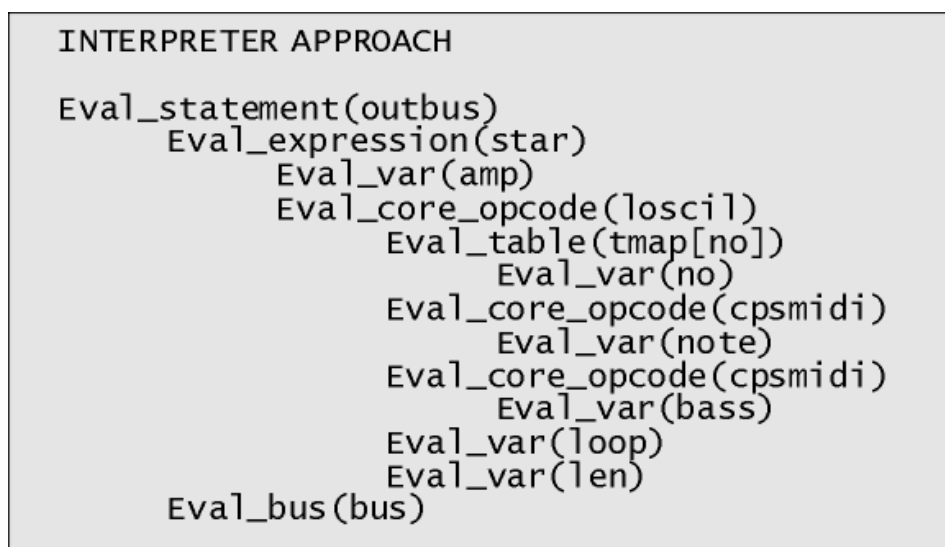


Figure 14 SAOL code interpretation by a common approach: each nested call is represented by an indentation.

VM 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)
```

Figure 15 SAOL code interpretation by the VM approach: execution at the initialization rate is represented by italic font

After the code decomposition, the block of code at *srate* is checked for explicit feedbacks in variables, and in *tablewrite*, *fracdelay* and *while* occurrences, as explained in subsection 1.2; the actual compiler labels a certain number of contiguous instructions as *s-b-s* in the middle of the whole block; this means that all the instructions and possible sub-blocks between the first and the last *s-b-s* line are labeled as *s-b-s* and executed in such a fashion, after and before two blocks executed by vectors, i.e. *b-b-b*.

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 simpler blocks, corresponding to the different execution rates. On the other hand it introduces an additional number of instructions to execute; this is experimentally proved as not being too heavy: see last section for experimental results.

3. The SAINT Virtual DSP Architecture

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

3.1. Memory Structures

In SAINT there are two main groups of memory structures, relating respectively to the *Instrument* and its *Instances*. The Instrument Memory first contains the Instruction Memory, i.e. the space allocated to contain the several instructions for both the instruments and the used user-defined opcodes. Opcodes, either core or user defined, are expanded inline into the main block, except in the case of an *oparray*⁷, when a static and self-standing "opcode space" is required for each

⁷ *Oparrays*, i.e. static arrays of opcode-based "processing cells", are a powerful extension of SAOL.

element of the array. Besides instructions, the Instrument Memory also contains registers; there are two main types of register:

- general-purpose (vectorial) registers, which contain intermediate calculations of expressions, and
- specific registers, which mainly contain SA global standard variables plus some architecture-specific variables used for code processing: the *PC*, *start*, *end*, the *block* size (these three for block processing) and *ret_address* (to return from sub-blocks).

The Instance Memory contains:

- memory space for local variables
- memory space for actual parameter lists of instruments and opcodes
- local standard names
- allocated space for buffers and delay lines (in filters and opcodes operating on memory).

In SAINT, the memory structures for the instance are visible to the instrument block through indirect addressing; consequently, the switching among the possible several active instances is simply done by changing the content of the base address. The main memory structures are summarized in Figure 16.

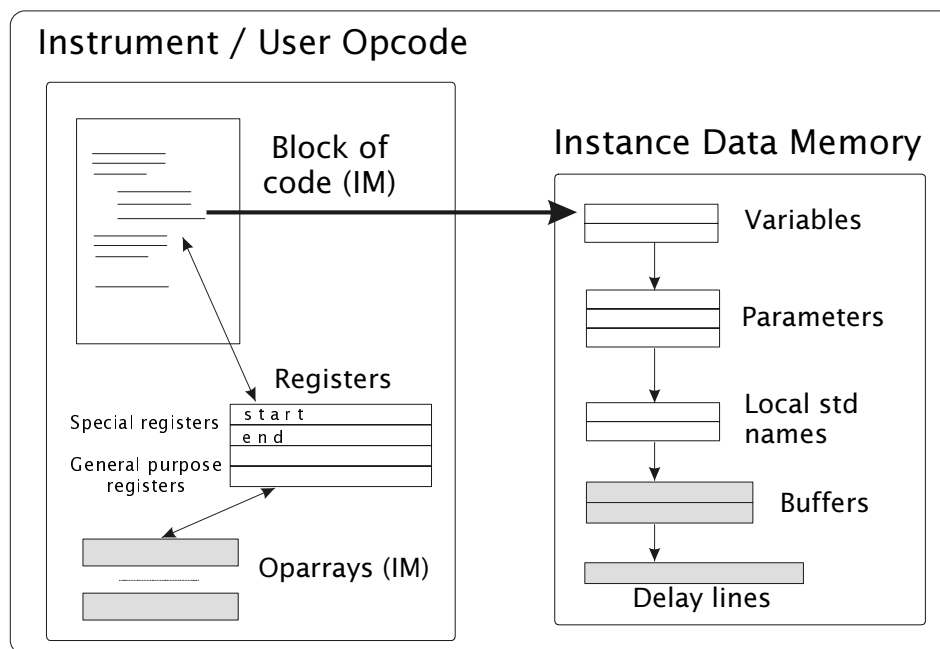


Figure 16 Main memory structures of the SAINT virtual DSP architecture: an instrument can have one or more active instances, each one with its own local allocated space. During performance, active instances are alternatively linked to the instrument, by changing the corresponding memory base address.

3.2. The Instruction Set

The instruction set of the SAINT virtual DSP is composed by two main groups: *macroinstructions* and *instructions*.

3.2.1. Macroinstructions

Macroinstructions 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 [98],[99], in SAINT there is no memory stack to work on, all the instructions directly operate on memory locations, and then they are defined with (normally) two or three addresses 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.

In practice, macroinstructions are conceived to behave similarly to JAVA native methods, and then optimized blocks of native code for the hardware device can be allocated to execute the SAOL code. The complete instruction set is described in Appendix A, while here only a basic classification and some examples are reported. 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, i.e. the language library, 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. The complexity analysis helps to reach the objective to isolate the computationally more complex and frequent routines to give them an entry in the instruction set table. For instance, many of the mathematical methods are important and used intensively or in bursts during initializations; instead some of them are seldom used and can be decomposed in one or a few lines.

Another 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 two most popular synthesis methods), are based on two main operations, i.e. interpolation and modulo increment of the phase (other than the unavoidable memory accesses and scalings). For instance, in the case of the *doscil* core opcode, reported as example in the complexity analysis of the previous chapter, the functionality is decomposed in SAINT as in Figure 17.

There, three vectorial operations, at the third, fifth and sixth line, are executed every control cycle for a block of e.g. 100 samples. The other oscillators and *tableread* can be implemented in a similar way (see Figure 15, exploiting the same macroinstructions).

```

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]);
           //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 17 Decomposition in SAINT bytecode of the SAOL dossil core opcode. Lines in italic are executed at *srate*, lines in normal font are executed at the *init rate*.

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 of SA 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 with long execution latency 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, about 50 macroinstructions are enough to represent all the opcodes. Considering statements and operators, the current definition of the macroinstruction set for the *virtual ALU* is composed of about 70 elements [81], [85]. Different macroinstructions for different rates are not useful since the vector length is flexible.

3.2.2. Instructions

To understand the role of the few *instructions* it is better to analyze, as an example, the execution of a complete control cycle of an instrument.

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 possibly greater than 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, like shown in Figure 18. Other instructions are used to access global variables, and in general for communication with the scheduler.

Considering the scheduler, in the proposed architecture it can be seen as being nothing else than a "hardwired" master DSP, able to coordinate the complete real-time process. Since its algorithm is fixed and it cannot be modified in any way, it is effective to implement it in native non-portable code, also because some of the interpretation overhead could derive from it. In conclusion, the complete resulting system is practically dedicated to Audio real-time programs only, since the "operating system" scheduling algorithm cannot be changed.

In common DSP systems the program to be executed is fixed; in a common platform based on an general purpose operating system every kind of program can be scheduled. In this case SAINT operates as an intermediate system where the scheduler responds to precise execution rules but different programs can be loaded without any particular constraint, apart that of being written according to the syntax of SAOL. A lack of flexibility on one side corresponds to a potential enormous extension in a purely DSP context.

A complete, multi-DSP SAINT system will be presented in Chapter 6.

```

// block of code at control rate
...

// first b-b-b audio block
p_set start 1
p_set end 100

// here b-b-b block of code

p_set end 1 // start s-b-s block: set end to 1
sbs: p_jump block end length;
      //jump to block if values of end and block are equal

// here 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 18 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. Implementation and Experimental Results

The virtual DSP architecture described in the previous sections has been implemented in C (compiler) and C++ (execution unit) as far as possible in a portable way. In particular the lexer and parser were written from scratch in standard C, to help portability at maximum. It is not here the place to report

extensively about practical implementation issues, but some details will be anyway presented in order to justify the strategic choices.

4.1. Implementation of SAINT

4.1.1. The compiler

When implementing a new compiler two choices are possible:

- a) Exploit one of the available generators of lexers and parsers, that are based on some pseudo-description language; or
- b) Conceive and implement the whole compiler from scratch

Some tools that can be used for the first approach are very popular. Probably the most known and used are *lex* and *yacc*, freely distributed for several platforms by GNU [100]. *Lex* is a lexer, short diction currently used for lexical analyzers. A lexer takes some input, normally in text format, and divides it into units (which are called *tokens*). *Lex* takes a set of descriptions of possible tokens and produces a C routine that will be used to identify those tokens. The set of descriptions that must be given to *lex* are calls a *lex specification*.

Yacc (for Yet-Another-Compiler-Compiler) is instead a parser. After the input is divided into meaningful tokens, a compiler needs to find the relationship among the tokens. For instance, a C or SAOL compiler needs to identify the expressions, statements, declarations, blocks, and procedures in the program. This task is called *parsing* and the list of rules that define the relationships that the program understands is a *grammar*. *Yacc* takes a description of a grammar and produces a C routine that can parse that grammar.

The MPEG-4 SA reference software, *saolc*, is based exactly on *lex* and *yacc* for compiling the code. SAOL specification and grammar for these two tools are distributed with the reference software, so that they may be slightly modified to eventually correct bugs or create more enhanced different compilers. Knowing rules to write descriptions for *lex* and *yacc* can then save a meaningful part of time in writing a new SAOL compiler. In spite of this remarkable advantage, two major drawbacks have been identified in the use of this kind of tools.

First of all, if the pseudo-language to feed *lex* and *yacc* is not known, any change to the compiler requires to practice these tools, and this compensate for a part of the time saved in writing the whole compiler. Second, and much more important: the output generated by *lex* and *yacc*, two C files to be included in the rest of the software project, are far from being readable and understandable by a common human programmer and, even worst, by some common C/C++ compilers. For instance, at the beginning of the SAINT initiative, compiling the MPEG-4 reference software used to be a demanding task with the last BorlandC++ compiler, since parts of the *lex/yacc* output generates there compiling errors and unrecognized structures.

Since the focus of the project has always been on real-time applications, portability and multimedia devices, the common solution based on *lex* and *yacc* has been considered too workstation-dependent, and heavy to be applied once it could be necessary to compile the code for e.g. a low-power processor contained in a mobile

wireless device, for which a compiler could be available that is not as robust as those for general purpose platforms. For this main reason, it has been decided to start the implementation of the SAOL compiler from scratch, including the lexer and parser of the language. This decision was also supported by the fact that many tutorials are nowadays available in literature for writing interpreters and compilers, and some of them also provide source code for the complete application [101]. In this way, a completely new standard and rather robust SAOL compiler has been developed and coded in ANSI C, modifying an existing compiler source code. This provides a painless portability on every tested platform.

4.1.2. The Virtual DSP

In parallel with the design and implementation of the compiler, the execution engine has been conceived and realized. In order to better mimic a HDL-based tool and obtain a software structure of the virtual DSP as near as possible to a hardware device (except perhaps for the rather high complexity of some instructions of the set) an object oriented language was preferred; being the compiler developed in C for opportunity, the language chosen consequently for the virtual DSP has been C++. It has been also verified that many devices, even some DSPs, offer today development environments with both C and C++ compilers, and that in any case it is not difficult to rebuild constructors, destructors and some other C++ features in simple C (and this is actually what e.g. the TriMedia C++ compiler automatically does before producing executables).

Aiming as a first stage at readability of the code and portability on multimedia devices, it has been chosen to implement a completely interpreted virtual DSP. This conveys a major drawback in term of performance, as the JAVA experience has recently demonstrated; but on the other hand it helps a fast porting at best, since the compiler and most of the scheduler can stay untouched.

The approach of a hardware-supported virtual machine has also been considered [103], but porting of such a resulting device has been considered not affordable in a reasonable time. Moreover, not all devices are conceived to support the direct virtualization of their instruction set [102].

Finally, it was expected that processing mainly through macroinstructions, operating on vectors of samples, could permit to limit the overhead by well-optimized blocks of native code.

4.2. Experimental Results

The first draft release of SAINT has been tested on two different platforms: an Intel Pentium II 400 MHz, with 32 Kbytes of Level 1 cache (16 kBytes for instructions and 16 kBytes for data) and 128 MB of RAM running Windows NT4, and a Sun UltraSPARC Ili 333 MHz, with 512 MB of RAM running Unix SunOS 5.6. The decoder was compiled on SUN using the default SunOS cc/CC compiler, while the PC version was compiled using BorlandC++ 5.2; optimization for speed was introduced.

Four different groups of simulations will be described in the following, reporting as a result the decoding absolute time elapsed from the instant $t=0$ of the scheduler (orchestra start-up) until the end of the performance. For each of the considered

examples different measurements have been conducted with different versions of the decoder and simultaneously with the latest release of the MPEG-4 reference software. In Figures from 19 to 22, the values of the six columns from left to right are the absolute decoding times for: 1) the MPEG-4 SA reference software; 2) the SAINT decoder without any b-b-b optimization; 3) the SAINT decoder with a block-by-block execution, when possible; 4) the previous decoder with the flattened structure for interpretation 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 in performance time (theoretical upper limit for real-time decoding).

Column 2 is generated with a version that only includes the basic execution engine, with the best solutions that have been reached for the implementation of the involved macroinstructions and with the partially optimized memory management that has been described previously. This is useful to test the efficiency of the SAINT structure against *saolc*, at the same level of development. The decoder used for column 3 includes feedback analysis and execution on vectors of samples, i.e. the vectorial instruction set has been enabled. The comparison between columns 3 and 4 is useful to verify the eventual overhead that could be provoked by the increased number of macroinstructions to execute when the code structure is flattened through intermediate registers; it is interesting to check whether this overhead is compensated by a simpler structure, at least until a certain limit, since a much smaller number of indirect addressings and nested allocations is required by the flattened solution.

In Column 5 a last interesting case is reported. The Optivec vectorial libraries are freely available on the Internet [106] and are an example of speed-up provided by handcrafted assembly programming over the code generated by high-level language compilers. They are provided for main Windows-based C/C++ compilers in integer and simple precision floating-point format, which is exactly the one needed for normative SA decoding. The inclusion of these libraries to execute corresponding macroinstructions (in cases where functionality matches), can help to provide a concrete idea of how effective the SAINT architecture is.

4.2.1. *Saolc*

The MPEG-4 reference software was developed and implemented at the MIT Media Labs and is called *saolc* [89]. This software has been donated to MPEG as a reference implementation of the SA standard, and for some time it has been the only existing decoder of Structured Audio. From the beginning its purposes has been readability of the code, clear description of the algorithms and completeness; these features not often match with efficiency in execution. *Saolc* too is an interpreter, but it is conceived as a step-by-step reproduction of the standard specification, and then it follows the sample-by-sample approach described there. Moreover, its execution engine is a classical interpreter, in the sense that each sub-expression is called from within the calling expression, and this causes an automatic uprating of lower rate expressions when this is not necessary, as explained in section 2. Of course this tool has been used as a reference also for the implementation of the SAINT

execution engine, and then it is quite straightforward to report, as a first comparison with other decoders, the gain in decoding time over saolc.

4.2.2. Examples

The first example is a quite common wavetable synthesis example, where a stereo piano at 44100 Hz is generated from monophonic wavetables, essentially through the *loscil* core opcode, and filtered by a reverberation based on a classic Schroeder scheme with 2 allpass and 4 comb filters [90]. The mean polyphony of the score file is approximately 3.5; the score duration is 18.5 seconds. The comparative results for the Intel-based platform are shown in Figure 19.

These results deserve some comments.

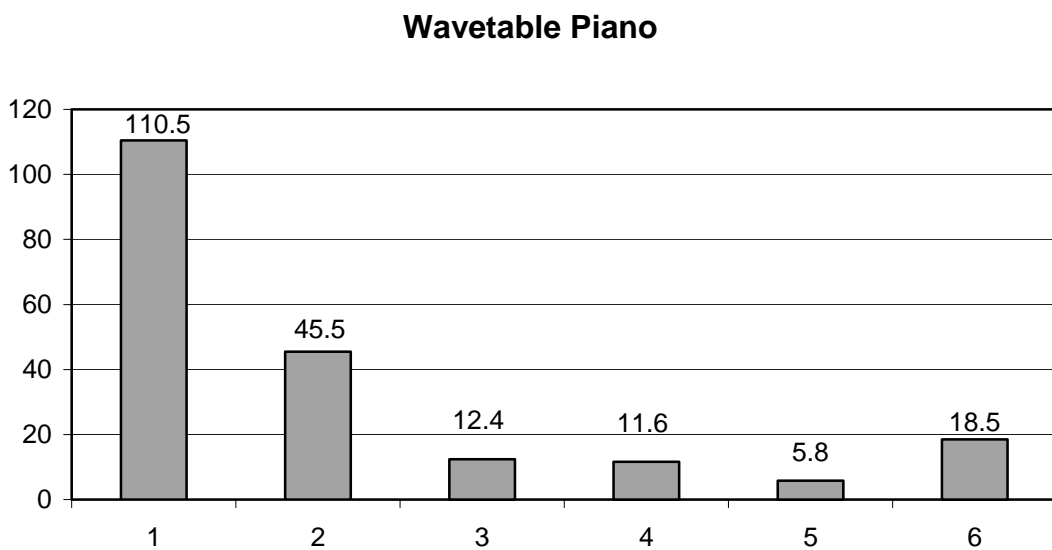


Figure 19 Experimental results for different decoding approaches (1). Y-axis is time in seconds. The values of the six columns from left to right are the decoding time for: 1) the MPEG-4 SA reference software; 2) the SAINT decoder without any optimization; 3) the SAINT decoder with a block-by-block execution, when possible; 4) the previous decoder with the flattened structure for interpretation; 5) 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: 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 100, a much lower factor than other results reported in literature for different tools (see for instance [95]); this is due to the fact that in both b-b-b and

s-b-s execution the control rate is the same: having the possibility to change the structure of the decoder to execute s-b-s (column 2), it is not necessary to use the "trick" of executing the control block of code at the audio rate to obtain the two measurements. In this way the small amount of pure processing operations is not increased (in the other way everything practically becomes a processing), while only the interpretation overhead is reduced.

When the block of code is flattened, without nested calls (column 4), the decoding time does not vary relevantly, it slightly decreases; this is a good result, because it permits to simplify the execution without any penalty in speed; this demonstrates that further overhead is more than compensated by flat code and by the elimination of unnecessary updates.

Finally, the introduction of the Optivec vectorial library on some basic functions (in this case only for interpolation, mathematical operators and summing bus) clearly shows how much this approach can be effective: consider in fact that parallelism is exploited here only at the software level, by assembly-written code, while the vectorial instruction set may be optimized with a greater speed-up factor on a truly parallel co-processor (see next chapter).

Comparative results for Sparc- and Intel-based platforms are shown in Figure 20 for the same example. Being 128 MB of RAM more than enough for this task, it seems that the main difference comes from the clock speed of the processor; for this reason further results will be reported only for the Intel-based machine.

Wavetable Piano

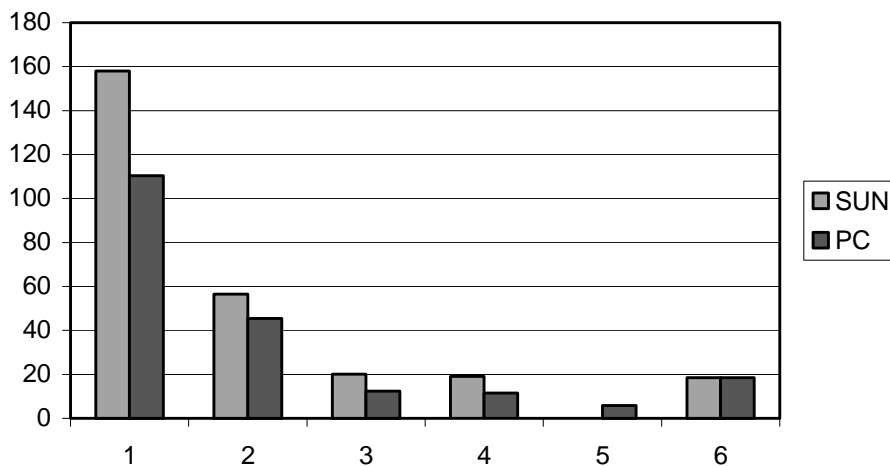


Figure 20 Comparative results for PC and Unix workstation platforms. Y-axis represents time in seconds. The values of the six columns from left to right are the same decoding times as described in Figure 19. Column 5 is empty for SUN since suitable vectorial libraries were not available.

A second synthesis example is an FM-generated brass with wavetable-generated attack (nearly 1 second); the frequency modulation part of the algorithm is

essentially based on the *oscil* core opcode (oscillator over a table that does not have its own base frequency), in a very similar way to examples tested in literature [68]; the orchestra contains a reverberation effect computationally similar to that used for the wavetable piano. Experimental results are reported in Figure 21. 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 SA simulator reports that, in one second of score, peaks are present of 7×10^5 interpolations, more than 3.5×10^6 multiplications and 3×10^6 mathematical methods, without considering tests and other floating-point operations. Due to this purely mathematical content, the gain obtained with vectorial libraries is impressive.

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 sub-block between two parallelizable blocks. The results for a short monophonic score without any additional processing are reported in Figure 22. 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 it was not possible to produce meaningful results for this case.

FM + Wavetable Brass

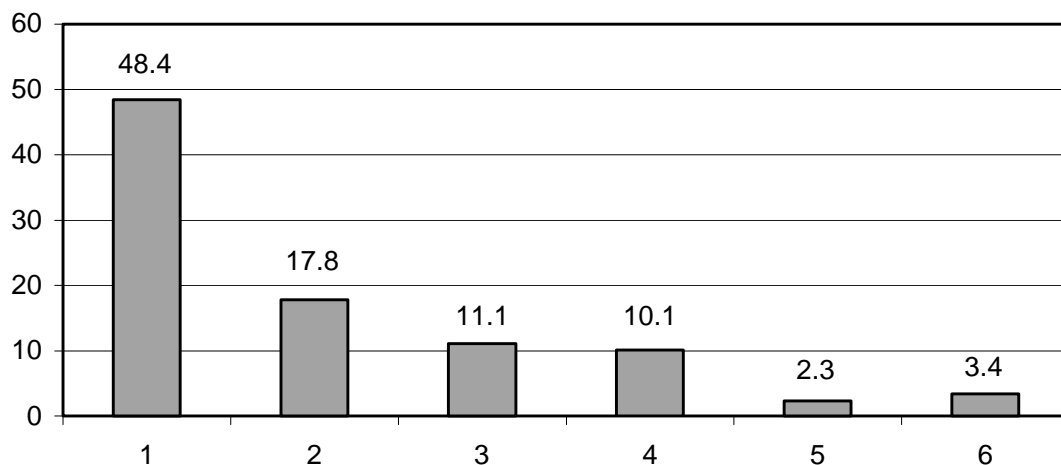


Figure 21 Experimental results for different decoding approaches (II). Y-axis represents time in seconds. The values of the six columns from left to right are the same decoding times as described in Figure 19.

To conclude the series of reported simulations, it is finally considered a pure processing algorithm, and precisely a digital mixer stripe composed by a low-shelving filter, a bandpass bell filter and a high-shelving filter. The stripe was used

to filter (in mono) the available output of the wavetable piano example, and the results are reported in Figure 23. In this case, like in the previous one, the vectorial library does not provide the necessary functions, and then the gain in the fifth column over the fourth is negligible.

Waveguide Physical Bass

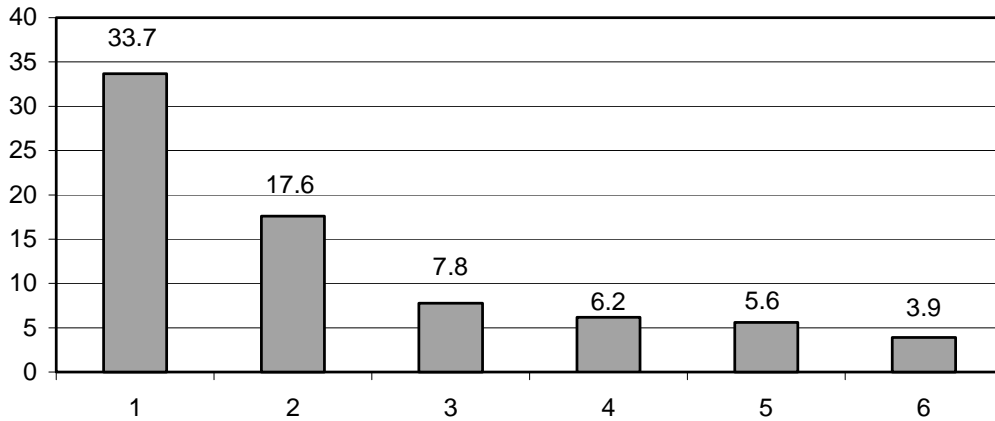


Figure 22 Experimental results for different decoding approaches (III). Y-axis represents time in seconds. The values of the six columns from left to right are the same decoding times as described in Figure 19.

Mixer Stripe

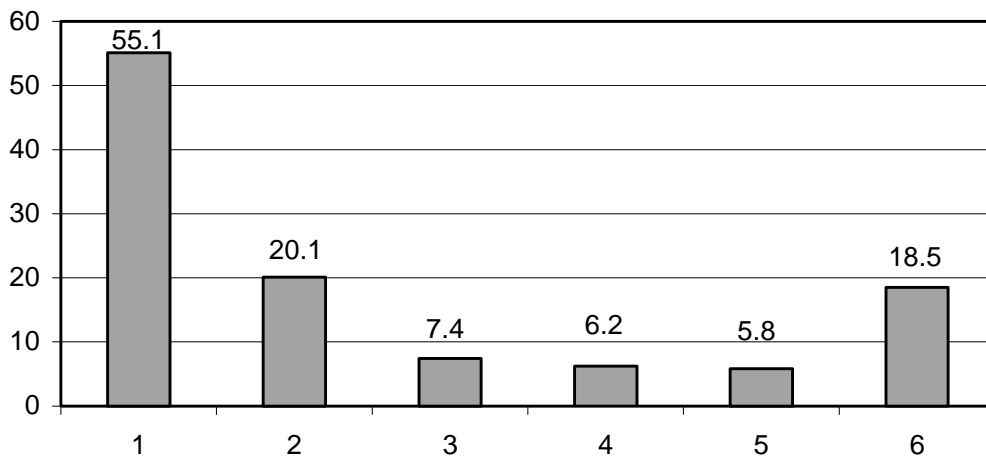


Figure 23 Experimental results for different decoding approaches (IV). Y-axis represents time in seconds. The values of the six columns from left to right are the same decoding times as described in Figure 19.

To conclude, it is interesting to report some simulation results of the wavetable piano at different block lengths, to measure how the decoding time evolves varying

this important parameter. Results are shown in Figure 24. The approach of varying block size through changes in the values of *krate* was followed (exactly as done in [95]), confirming this time results reported in literature about the saturation in speed-up with the block size. White columns refer to the complete C++ implementation, black columns to the SAINT version with vectorial libraries.

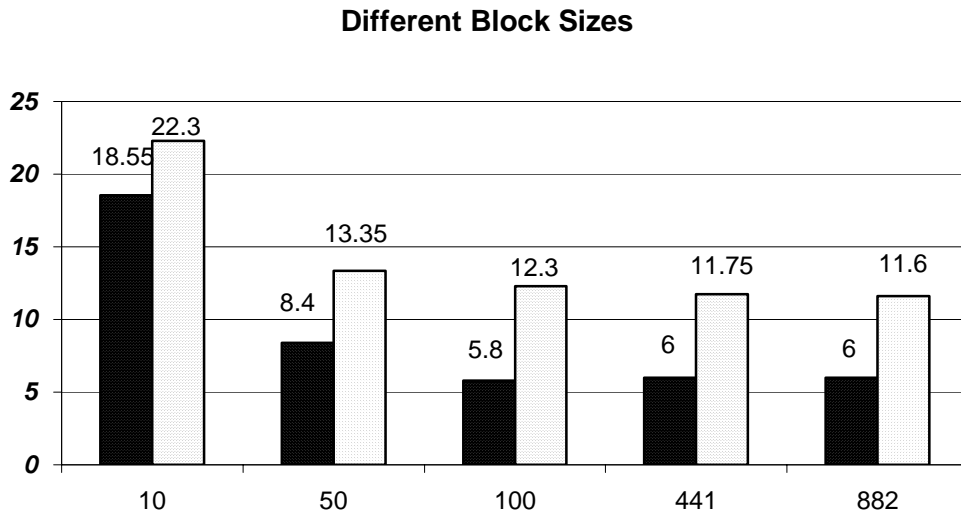


Figure 24 Performances for different block sizes on Intel platform. The considered sequence is the same of Figure 6. Y-axis represents time in seconds. X-axis represents the block length in samples. White columns are calculated by the SAINT decoder with a block-by-block execution, when possible; black column are calculated by the SAINT decoder with the "Optivec" free downloadable vectorial libraries for Pentium.

4.3. Another Comparison with Existing SA Decoders

When one wants to execute a program that is written in a high level programming language, three solutions are possible: an interpreter, a cross-compiler to another existing language, a compiler to native hardware instructions. The first one is the approach described so far for SAINT, the easiest to conceive (not forcedly to implement) and the most intuitive and straightforward. The second one requires a consistently lighter effort in development, since many kind of optimizations and solutions are left to the existing compiler that will be exploited to translate the intermediate code into the machine code; this approach is reasonable when the intermediate language is widely accepted and its compilers extensively optimized. The last approach is the most demanding one, and it is cost effective probably only in case of a language that is supposed to have a wide use enough to justify the production of very efficient code on a particular platform.

Of course, portability of the solutions decreases from the first to the last.

At the moment of the redaction of this dissertation no big companies announced product based on SA decoders yet. A further comparison than saolc will be proposed with a tool with a completely different approach, *sfront*.

4.3.1. Sfront

A different solution to SA decoding, proposed a few time after SAINT, is *sfront*, developed at the Computer Science department of the University of Berkeley [80]. Sfront has been conceived to quickly obtain a fast and complete decoder mainly for educational purposes and then the chosen approach has been the cross-compiler to C language, since general purpose platforms are quite suitable for this task, having many C compilers available. Most of the optimization task is transferred in such a way to the C compiler, and this is an obvious advantage, together with the fact that the generated code is processor native code. The generated C code has nested calls at slower rates separated from srate code, but no kind of optimization on block execution is present.

The clear disadvantage of such an approach is the necessity of a C compiler installed on the machine, and the split of the decoding process into three stages: the SAOL/SASL translation into C, the C compiling process and the execution of the obtained program.

An additional disadvantage of the *sfront* approach is the difficulty to dynamically add instruments and modify the orchestra in runtime. This is of no importance if only SA is considered, since SAOL is a static language and dynamic allocation of instrument is forbidden as well as dynamic changes in routing among instruments. It will be shown in the next chapter how SAINT has been extended for complete BIFS scenes, and in this case BIFS is a dynamic description framework, where new nodes can be linked to the scene and routings among nodes can be redefined. In such a context an approach like the one of *sfront* is clearly hard to optimize. At a higher level it could be possible to argue that the cross-compiler solution will have to move towards more "interpreted" (i.e. dynamic) allocations of code, getting in this way nearer to a solution like SAINT, which moves from a dynamic approach and loads macroinstructions of statically optimized native code. The comparison between SAINT and *sfront* is summarized in Table 5.

Table 5 A comparison between SAINT and *sfront* decoding approaches

Feature	SAINT	Sfront
Decoding solution	Interpreter	Cross-compiler to C
Implementation language	C/C++	C
Main operating systems	Windows - Solaris	Linux - Windows
Portability	Requires a C/C++ compiler <i>for</i> the target device	Requires C compiler running <i>on</i> the target device
Intermediate output	SAINT bytecode	Native machine code
Static optimization	Fast code execution Fast instruction decoding	Fast C code Optimization of C compiler
Dynamic extensions	Easily to plug in	Requires dynamic linking

Sfront is written completely in C and developed on the Linux operating system, but full support is anyway provided for Windows and Unix platforms.

4.3.2. A Comparison in Performance

For several reasons an objective comparison in performance between SAINT and sfront is difficult; it can be anyway estimated with a good approximation. Among the reasons that make this comparison hard it is possible to mention:

- the Optivec vectorial library that has been linked as fast SAINT instruction set for validation does not provide the complete needed functionality, and then some important operators are still evaluated compiling non optimized C code.
- sfront does not include yet a high quality interpolation, then all the comparisons can only be made with linear interpolation, thing that does not correspond to the main target application area of SAINT.
- the two decoders are not yet completely optimized in all their aspects, and then different examples can give slightly different comparisons depending on the structure of the SAOL code.

Given these fundamental remarks, in Table 6 some comparisons are reported between SAINT and sfront on two different platforms, the first one being the same Windows NT platform described before and the second an older Windows NT system with a Pentium MMX processor at 200 MHz and with 64 MB of RAM and 256 kB of secondary cache. The C file produced by sfront has been compiled in the same conditions of SAINT by the BorlandC++ compiler.

Table 6 A comparison between SAINT and sfront

Processor	Sequence	SAINT	sfront
Pentium II 400	Brass	2.33 s	2.40 s
Pentium II 400	Physical Bass	5.59 s	2.70 s
Pentium MMX 200	Brass	5.01 s	3.42 s
Pentium MMX 200	Physical Bass	7.72 s	4.18 s

Results for the older WindowsNT platform have been reported to show how, in the case of the brass sequence described earlier, SAINT has a performance that is the 40% slower on the Pentium MMX and even slightly better on the Pentium II. This comes from the fact that one of the main disadvantages of a vectorial execution on a non parallel device is the drawback in additional memory allocation, which can result is considerable slowdown when cache memory size is not sufficient to avoid additional data transfers.

In the case of the physical bass sequence, the performance of SAINT is 80-100% slower than sfront; this had to be expected, since the physical bass sequence contains a considerable part of the code executed in sample-by-sample fashion and then the b-b-b optimization cannot be exploited. A comparison between column 2 and 3 in Figure 21 and Figure 22 can immediately justify this assertion.

To conclude, it is clear that on a pure performance level, an interpreter approach is in the mean slower than a cross-compiled one, and this is no news [4]. On the other side many multimedia target applications requires a more flexible and portable solution that can be obtained at a much lower cost by the proposed virtual interpreted device.

CHAPTER 6. THE PROJECT THREEDSPACE: A COMPLETE 3-D AUDIO RENDERING FRAMEWORK FOR MPEG-4 APPLICATIONS

This chapter will introduce the major applicative outcome of the described method of analysis by virtual model finalized into the SAINT decoder, namely the ThreeDSPACE Project.

The first part of the chapter will be dedicated to a preliminary overview of the project and to a more detailed description of advanced features of the AudioBIFS framework, which make it the best candidate to support the desired functionality; then extensions to SAINT necessary for compliance with AudioBIFS will be described. The last part, moving from requirements to the evaluation of candidate solutions and to the final implementation, includes an overview of the chosen Audio rendering terminal and a detailed description of the realized ThreeDSPACE system.

1. Advanced 3-D Audio Applications and AudioBIFS: Scope of ThreeDSPACE

In the last years it has become quite common to read or hear about 3-D Audio, spatial sound, virtual Audio spaces and so on. In fact, every Audio board, active loudspeaker system or consumer electronics amplifier claims to provide 3-D Audio capabilities. Unfortunately this is far from being true and often the announced features reduce to some kind of cheap reverberation with stereo panning, things that are well known since many years and that are not able to produce a truly immersive sound field.

3-D Audio is not a cheap and easy issue and its correct implementation requires hundreds of MIPS, as it is possible to argue from literature reporting years of research on the topic (for instance [86], [109], [110], [111], but the list could be huge). This is why it is not easy to produce (and, consequently, sell at low prices) systems that can be really classified as 3-D Audio ones. In such systems all parameters reported in Figure 25 must be potentially under the content provider's manipulation and recognizable by a trained listener [84].

1.1. Overview of ThreeDSPACE

ThreeDSPACE moves from a true 3-D Audio context and aims at producing true 3-D rendering, which is often classified as "advanced" when compared with the simplistic solutions mentioned above, where only source azimuth and environmental context (reverberation) are normally taken into account. Starting from necessarily medium-to-high cost and professional equipments the project wants to propose a technology that has the potential to be developed for low-to-medium price mass products.

The main scientific and technical objective of the ThreeDSPACE project (for Three-dimensional audio Devices and System for MPEG-4 Audio Composition and rendering) is to implement advanced 3-D audio processing and rendering methods, compatible with the MPEG-4 AudioBIFS (Audio Binary Format for Scene description) specification. These methods can represent the basis for a family of products exploitable first of all in theatres, viewing rooms, video conferencing rooms, and high-end multimedia systems.

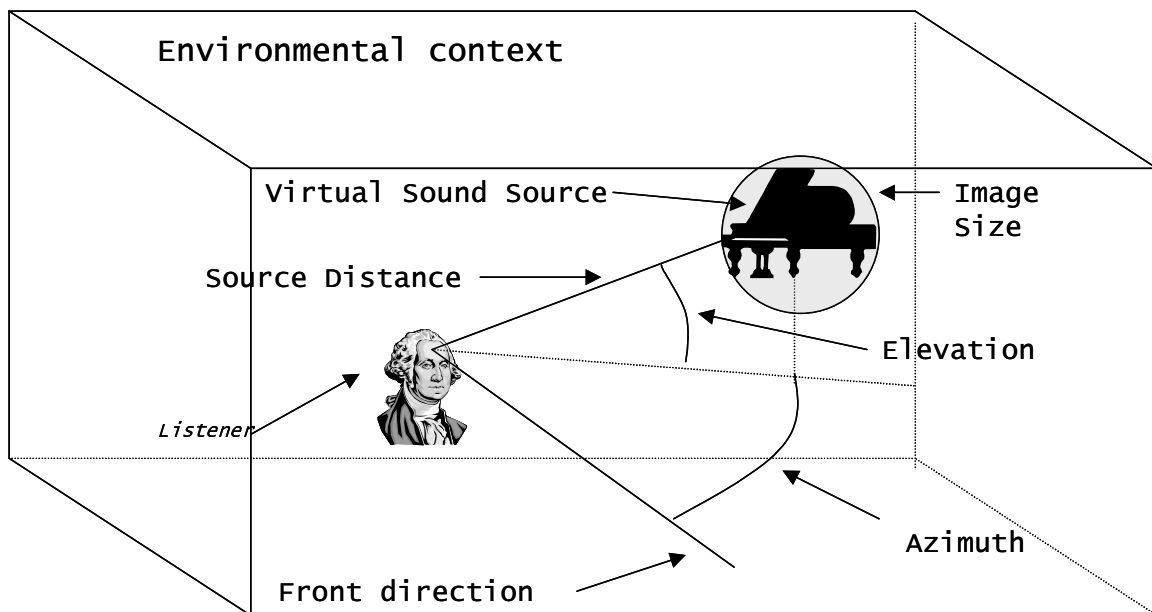


Figure 25 Parameters normally manipulated by the content provider and perceived by the listener in an advanced 3-D Audio description. Azimuth is the angle between the front direction of the listener and the direction of the sound perceived by the listener, in a plane parallel to the floor. Elevation is the angle between the same plane and the direction of the perceived sound. In a mass solution the several listeners should be placed relatively to an absolute reference listener.

Among many available and potential solutions (see subsection 3.1 later), wave field synthesis (WFS) has been chosen as the rendering technology. With WFS loudspeaker stripes, containing a high number of separate channels, must be mounted directly to walls. The signals are generated by dedicated signal processors, which need suitably encoded sound-field representations as inputs. MPEG-4 AudioBIFS is the only current standardized format that can provide the necessary information for the sound field without having to encode many sound channels even for a simple monophonic sound source.

As shown in the second part of this chapter, a spherical wave coming from a virtual source behind the array, or the extrapolated wavefront of a source in front of the array can be reproduced using the WFS solution.

After having just introduced the scope and goal of ThreeDSPACE, it is now necessary to detail further the AudioBIFS features and syntax, to show how they match perfectly the capability to represent parameters of Figure 25; support of these

features require modifications to the SAINT virtual DSP that transform it into a general multimedia-oriented Audio device.

1.2. BIFS and Advanced AudioBIFS (AABIFS)

In Chapter 3 it was said that BIFS inherits much of its structure from VRML, but the Audio subtree introduces indeed a completely new functionality. The only point of contact between the two standards is given by the *AudioClip* and *Sound* nodes, which are used in VRML to attach a sound clip to the scene and to place it in the scene respectively. These two nodes also belongs to version 1 of BIFS, together with the already discussed AudioFX node and other simple nodes providing the capability to mix, to switch, to delay and to buffer sounds [55].

1.2.1. AudioBIFS v.1 and ThreeDSPACE

The *Sound* node specifies the location (spatial position) of a sound object in a BIFS (or VRML as well) scene. The sound object is attached through a field called *source* and can be provided as a generic Audio subtree. The resulting sound is located at a point, in the local coordinate system, specified by the *location* field. The scene renderer emits sound in a frequency independent ellipsoidal pattern, with the orientation of the ellipsoid defined by the *direction* field.

The audible sound field produced in a scene by the *Sound* node is detailed in Figure 26. It consists of two nested ellipsoids whose shapes are defined by fields *maxBack*, *maxFront*, *minBack* and *minFront*. Within the inner ellipsoid, the sound is scaled by the intensity field and there is no attenuation, i.e., the sound level is independent of the location of the virtual listener. Between the inner and outer ellipsoid, the sound level decreases linearly on a dB scale from 0 dB (the level inside the inner ellipsoid) to -20 dB. Outside the outer ellipsoid, no sound is rendered.

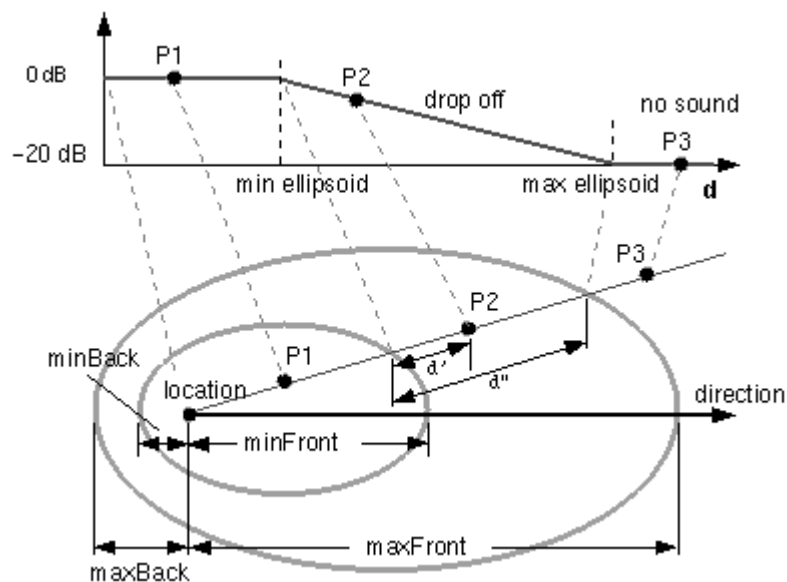


Figure 26 The ellipsoids that define the Sound spatial rendering in both VRML and AudioBIFS. At the top of the picture is represented the drop in sound intensity inside the rendering space (this picture is pasted here from a draft VRML document).

The *spatialize* field of *Sound* specifies whether or not the audio object will be spatialized when rendered. If the *spatialize* field contains the value TRUE, the virtual listener's *direction* and the relative *location* of the *Sound* node are taken into account during playback. However, the method of spatialization is not normative; it is assumed that the renderer uses the maximum sophistication available, and this is possibly a weakness of the Systems specification.

When multiple *Sound* nodes are contained in a single scene, a decoder adds together the sound from each of them to create the overall audio scene. A simple example of BIFS code is represented in Figure 27.

1.2.2. AudioBIFS and interaction

Features of MPEG-4 also allow the construction of highly dynamic and interactive content.

```

Group { children [
    DEF VP ListeningPoint {
        position 0 0 3.3
        orientation 0 -1 0 0
    }
    Group {
        children [
            Shape {
                ...
            }
            Sound {
                source AudioSource {
                    Url 3
                    StartTime 0
                }
                direction 0 0 1
                spatialize TRUE
                location 0 0 0
                priority 1
                minBack 2
                minFront 4
                maxBack 10
                maxFront 10
                intensity 0.6
            }
        ]
    }
] }

```

Figure 27 An example of BIFS code, where a simple Audio Object is spatialized through the *Sound* node capabilities. For simplicity the graphical part of the scene (*Shape*) is omitted.

Most of the fields in the AudioBIFS nodes are defined as *exposed fields*. This means that their values can be changed dynamically during the content playback. These changes could come from the source server as well as from an interactive event-routing model; in this latter case special node called *Sensors* are used to detect user

actions (for example mouse movements and clicks) and then routed to the exposed fields. For instance, if the content contains a user interface that allows the user to manipulate the values in the *matrix* field of an AudioMix node, the result is to give the listener control over the “fader levels” in postproduction.

The scene graph itself may be dynamically modified through a special stream called the *BIFS Update* stream. The effects of the BIFS Update may be very simple and changing one field in the scene parameters, or very complex and replacing the entire scene graph with a new scene graph.

1.2.3. *AudioBIFS v.2 and ThreeDSPACE*

In true composition for virtual reality the goal is to recreate a particular acoustic environment as accurately as possible. Sound should be presented spatially according to its location relative to the virtual listener in a realistic manner; moving sounds should have a Doppler shift; distant sounds should be attenuated and low-pass filtered to simulate the absorptive properties of air; and so on. The VRML-like *Sound* nodes do not offer suitable functionality for such acoustical features like sound reflections, reverberation time, the Doppler effect, frequency-dependent distance attenuation or more sophisticated modeling of sound-source directivity. MPEG-4 Version 2 extends, thanks also to the active contribution coming from the ThreeDSPACE project, the simple virtual reality model of version 1 to include two richer techniques for creating virtual audio environments. The first technique is physical: modeling of the acoustic environment is bound to the physical reality defined by the visual scene. The second is perceptual: creation and modification of environmental sound characteristic are based upon perceptual parameterization.

1.2.4. *Physical (geometrical) approach*

In physical modeling of acoustic environments processing of sound is made in such a way that the acoustic effect corresponds to the visual scene. This involves modeling individual sound reflections of the walls, modeling sound propagation through objects, simulating air absorption, and rendering late diffuse reverberation, in addition to the 3-D positional rendering of source locations. This type of environmental spatialization is sometimes referred to as auralization or virtual acoustics [118].

In Version 1 of the MPEG-4 standard, as in the VRML standard, the virtual reality model of the sound source only provides techniques for placing the sound source in the 3D space and for coarse simulation of sound source directivity by the elliptical sound source patterns, described in subsection 1.2.1.

To improve this model, the spectral content of the sound should change as a function of distance [58]. This occurs in natural environments because of the low-pass filtering effect of air absorption. Another improvement is that to enable more flexible simulation of the frequency-dependent radiation patterns of real sound sources. For example, a brass instrument has more high-frequency spectral content when listened to from the front as compared with behind. Finally, the sound source model in Version 1 AudioBIFS does not take into account effects of the environment. Among these are the Doppler effect caused by the coupling of the propagation delay

of the sound to the relative movement of the sound source and the listener, and the interaction (reflection, transmission, occlusion) of sound with other objects in the scene.

In the second version of MPEG-4, three new nodes (*AcousticScene*, *AcousticMaterial*, and *DirectiveSound*) have been introduced for advanced auralization of audiovisual scenes. With these new nodes it is possible to define geometrical regions in the scene where different acoustic responses are applied to sound according to the virtual locations of the sound source and the virtual listener. The *AcousticScene* and *DirectiveSound* nodes together allow the specification of properties of sound propagation and attenuation in the "medium". The *AcousticMaterial* node gives visual and acoustic properties to polygonal surfaces.

1.2.5. *Perceptual approach*

Audio spatialization can also be approached by non-physical concepts, investigating the perception of spatial audio and room acoustical quality. This process is called the perceptual approach to acoustic environment modeling [119].

Perceptual parameters have also been introduced into MPEG-4 Version 2 as another method of creating environmental acoustic effects in the scene, independent of the visual and physical reality. These parameters enable the creation of environmental effects separately for each sound source, adjusted to characterize the perceptual quality of the source and the environment in a 3-D space. High-level perceptual parameters (such as source presence and brilliance, room reverberance, etc. [119]) are used to derive low-level energy parameters for the control of direct sound, and the different parts of the room impulse response, i.e. the directional early reflections and the late reverberation [86].

Based on these parameters, a real-time spatial sound processing scheme has been derived in MPEG-4, which enables computationally efficient and perceptually relevant 3-D audio rendering. The only input required from a geometrical representation of the acoustic space and its objects is the distance and orientation between the source and the listener. The perceptual rendering engine does not need to utilize other geometrical information about the acoustic space.

This approach is meant mainly for applications where the environmental response does not have to correspond to the visual environment, but where high-quality virtual room acoustic effects are nevertheless desired. The perceptual parameters and processing are therefore also useful for audio-only postproduction in MPEG-4.

An example of version 2 BIFS code (Advanced Audio BIFS, AABIFS) is reported in Figure 28; the node *DirectiveSound* replaces the node *Sound* and permits to include much more meaningful information to the acoustical scene description.

1.3. *Conclusions*

ThreeDSPACE has very demanding requirements that are nowadays satisfied in a normative way only in MPEG-4 AABIFS. The first target of potential ThreeDSPACE-based products, museums, theaters and so on, does not necessarily require a standard coding format, but the use of a proprietary format could hardly preclude in

the future any exploitation on the mass market. This is why the ThreeDSPACE team since the very beginning of the project collaborated with MPEG and intended to exploit this advanced technology for its purposes.

```
Group { children [  
    DEF VP ListeningPoint {  
        position 0 0 3.3  
        orientation 0 -1 0 0  
    }  
    AcousticScene {  
        size 100 40 100  
        center 0 0 0  
        reverbTime 0  
    }  
    Group { children [  
        Shape {  
            ...  
        }  
        DirectiveSound {  
            source AudioSource {  
                url 3  
                startTime 0  
            }  
            direction 0 0 1  
            spatialize TRUE  
            roomEffect FALSE  
            location 0 0 0  
            intensity 0.6  
            speedOfSound 20  
            distance 100  
        }  
    ] }  
]
```

Figure 28 An example of BIFS code, where a simple Audio Object is spatialized through the DirectiveSound node capabilities. For simplicity the graphical part of the scene (Shape) is omitted.

Two solutions are possible to implement the MPEG-related part: the first one is to implement the complete version 1 Audio subtree and to transmit more advanced audio information (such as room responses) as SA tables in an AudioFX-related stream. The second solution is to implement the complete AABIFS functionality, which requires a more demanding effort in software implementation. Both these solutions lead to the necessity to extend the SA decoder described in the previous chapter to become a complete AudioBIFS decoder. This process will be described in the next section.

2. Multimedia extensions to SAINT

An architecture and an instruction set conceived and optimized for Structured Audio only, like the case of SAINT, reveal some limitations when the device must be used for more general multimedia-oriented applications. Even if SAOL can potentially be used to describe any kind of algorithm, being a programming language and then a general audio coding scheme as previously mentioned [73], many fundamental features of a typical multimedia context cannot be described with the necessary effectiveness.

Moving from the purpose of extending the MPEG-4 SA decoder to a complete AudioBIFS subtree processor, several extensions have been included in SAINT to support at best functionality of higher-level multimedia description languages.

2.1. From SAINT to BIFSAINT

The implementation of the complete AudioBIFS subtree requires the solution of non-trivial implementation challenges, first of all the correct synchronization between the Structured Audio built-in scheduler, which is active to process the AudioFX nodes, and the BIFS scene scheduler. An evaluation of practical implementation issues [57], [129] suggests that an effective solution can be obtained by the integration of the complete audio subtree into an "extended" orchestra (the complete SA program), where all the several nodes are transformed into SAOL functions (instruments) and linked to the input/output buffers of the eventual AudioFX nodes in the correct sequence.

2.1.1. BIFS to SAOL

This solution requires to "cross-compile" the AudioBIFS subtree to a sequence of extended SAOL functions and to proceed then to the execution of the complete audio scene using the SA scheduler as the overall mechanism for synchronization. In this way a correct synchronization with the video part of the BIFS scene is only required, whenever this part should be present (supporting the "Main" Profile).

Consider as an example a scene containing an *AudioMix* node with two inputs and two outputs, where two monophonic inputs are mixed to a stereophonic output. In BIFS this will be written as follows:

```
DEF myMix AudioMix {
    children [
        AudioSource {...}
        AudioSource {...}
    ]
    matrix [0.4 0.6 0.5 0.5]
    numInputs 2
    numChan 2
}
```

matrix is an exposed array of parameters with `numInputs*numChan` elements; the value of the output buffer for an *AudioMix* node is calculated as follows. For each

sample number x of output channel i , $1 \leq i \leq \text{numChan}$, the value of that sample is:

```
out[x][i] = matrix[(0)*numChan + i ] * input[1][x]
           + matrix[(1)*numChan + i ] * input[2][x]
           + ...matrix[(numInputs-1)*numChan + i]*input[numInputs][x]
```

where $\text{input}[i][j]$ represents the j^{th} sample of the i^{th} channel of the input buffer, and the matrix elements are indexed starting from 1. The node of the considered example can be then translated into a SAOL `myMix` instrument as follows:

```
instr myMix( ) {

    imports ksig matrix[4];
    asig ii, output[2];

    ii = 0; output = 0;
    while(ii < 2) {
        output[0] = output[0] + matrix[ii*2]*input[ii];
        output[1] = output[1] + matrix[ii*2 + 1]*input[ii];
        ii = ii + 1;
    }
}
```

The control array `matrix` is imported from the global block of the orchestra, so that it can be modified by user interaction; `input` is a standard variable of each SAOL instrument that contains the values of the bus that is routed to such an instrument. Of course an even more straightforward solution is to get rid of the while loop and directly insert the index values for every element of `matrix`.

It is not wise instead to specify a generic *AudioMix* instrument and define each such node as an instance of that instrument, because a completely parameterized solution is not efficient. Moreover each *AudioMix* node can be potentially routed to a different bus, thing that is SAOL is much easier to be done with different instruments.

In a similar way all the *AudioBIFS* nodes can be translated into SAOL code and then into SAINT bytecode, in order to be executed by the virtual DSP. For an effective interpretation it is in any case a necessary to extend the current SAOL functionality; these extensions are requested in both the instruction set and the memory management of the DSP, to deal essentially with more dynamical characteristics of BIFS (and also of other commonly used multimedia languages like VRML).

2.1.2. Extensions to the SAINT Virtual DSP

Dynamic routing support. In SAOL the several instruments are statically defined at the beginning of the decoding process, when the bitstream header is received, and so are the routings among them; routings define the relationships among the input

and output buffers of the different instruments. Only commands to create new instances of a defined instrument can be transmitted by streaming information.

In BIFS, as also in VRML, this is no more enough because routings among the nodes are dynamic and new nodes may be instantiated in the middle of the scene representation when new commands are received via the *BIFS Update* bitstream (see subsection 1.2.2). Since in the proposed system the interaction between the parser/compiler and the execution engine is kept as far as possible separate, the virtual DSP must be made able to process streaming commands dealing with dynamic configurations of the input/output buffers, and above all with instantiation of new instruments. For example, a new mixing *AudioMix* node may be instantiated, with a mixing matrix processing four inputs to produce two outputs, linked with specified sources and to target specified post-processing nodes.

Advanced functionality support. A second group of extensions is necessary to support functionality provided by the most advanced nodes of BIFS. If some functions like mixers, delay lines and switchers are not difficult to translate into SAOL as it is, on the other hand AudioBIFS provides the possibility to spatialize monophonic sources using the surveyed complex schemes based on room modeling or perceptual parameters. It is evident that in such cases some dedicated instructions are necessary to limit the overhead of the interpreted decoder to a minimum and to better optimize the transcoding into the intermediate format. For instance, even the *Sound* node described above can be implemented in many different ways, according to the target of the application; in any case it requires an amount of computation well above the mean SAOL core opcode and moreover it has a direct relationship with the output configuration of the system. The specification of dedicated instructions to speed up e.g. the geometric calculation of early echoes reflections or the management of multi-tap delay lines can result very effective.

Multiple suborchestras support. Finally, AudioBIFS allows the presence of nodes working at different sampling rates, and lower rates must be converted to the higher ones at the most advanced point in the processing tree. In fact, sampling rate conversion can be seen as a kind of "hidden BIFS node" [57], which is placed inside the scene according to the sampling rate requirements of the several subtrees.

Different sampling rates are not allowed in standard SAOL, where all the instruments must work at the same audio rate. It is then necessary to extend the scheduler with the capability to treat different subgroups of instruments (i.e. different orchestras) that are combined through asynchronous sampling-rate converters at the correct point. In practice, converter nodes become "hidden SAOL instruments" that operate under the scheduler control at the moment of linking the actual input of the current orchestra to its input bus.

2.2. The Final BIFSAINT Architecture

In Chapter 5 it was mentioned the particular role of the SA scheduler in the SAINT virtual architecture: an hardwired master DSP characterized by a fixed algorithm to synchronize the several instances and sequences of instruments in an orchestra. With the extensions presented in the previous subsection, the SAINT (or now,

BIFSAINT) scheduler assumes even more the role of master of a complete multi-DSP system as shown in Figure 29. Different orchestras are synchronized, according to their sampling rate and block length. Each orchestra is actually executed in a different virtual DSP, whose output is routed to another DSP or composed to the output by the scheduler output manager.

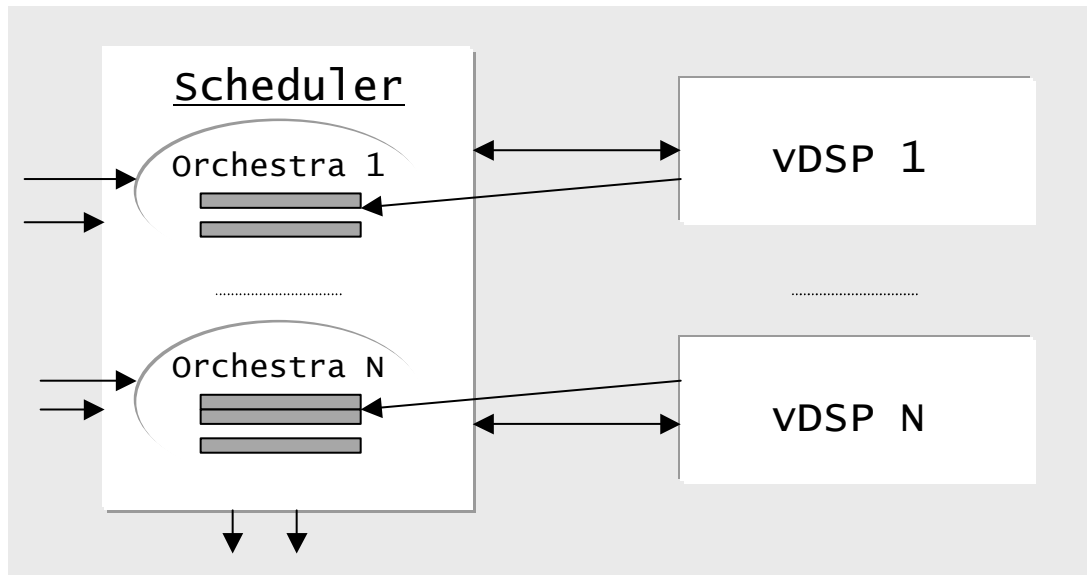


Figure 29 The final BIFSAINT multi-DSP architecture

3. Overview of Wave Field Synthesis

To better understand the potential of ThreeDSPACE and how the BIFSAINT virtual device fits into it, it is interesting to describe the main features of the innovative and not well known yet wave field synthesis technology; this technology provides a dramatic improvement over the current spatial audio solutions.

3.1. Current 3-D Techniques

The great majority of current advanced spatial audio techniques are limited to single listeners, because they aim at reproducing ear signals, with headphones or small, near field loudspeakers. The target places of fruition of ThreeDSPACE, however, demand for so-called volume solutions, i.e. the correct spatial reproduction of acoustic events within large areas. Techniques normally exploited by the entertainment industry for cinemas, home-theaters and virtual simulation rooms, rely on 6 or 8 loudspeakers (5.1 and 7.1 formats [112],[113]). The major drawback of these techniques is that the perception of the correct spatial information is tied to the so-called "sweet spot", away from which the best perception is progressively lost [116].

The same is true for another format well known among researchers, Ambisonics [114],[115]. Ambisonics works with a minimum of four loudspeakers, but spatial perception can be progressively improved increasing this number, thing that can be done thanks to the particular encoding format. When only four or six speakers are used, ambisonic sound has bad performances for moving listeners and its channel

separation is limited ([116],[117]). These issues can be partially solved using more *encoded* channels, but the quantity of information to transmit or store also dramatically increases. Moreover, Ambisonics remains a non-standardized format.

3.2. Wave Field Synthesis

In the late eighties, a fundamentally new concept was proposed [120]. In this new concept, wave theory plays an essential role and individual loudspeakers are replaced by arrays of loudspeakers that generate wave fronts from true or notional sources. Unlike all the other existing methods reviewed in the previous section, the wave front solution is a so-called volume solution, i.e. it generates an accurate representation of the original wave field in the entire listening space (and not at one or a few sweet listening spots).

In the ideal situation the listening area is surrounded by planes of loudspeakers, which are fed with signals so that they produce a volume flux proportional to the normal component of the original sound field at the corresponding position. For practical purposes, this method has been adapted to make use of linear loudspeaker arrays surrounding the listening area, rather than planes of loudspeakers [121],[122].

For the reproduction of a single source the basic scheme contains a tapped delay line with n taps and the corresponding scale factors in order to feed an array of n channels. The coefficients must be chosen to generate the correct curvature of a virtual source located in front, or behind the array. Practically, a maximum spacing between the individual channels of about 4 to 8 cm. should be ensured at high frequencies (above 2000Hz), and about 10 to 20 cm. below, in order to avoid spatial aliasing [122]. An envisaged system, suitable for a framework as ThreeDSPACE, should incorporate about 40 high frequency channels and 10 low frequency channels over a length of 2 m. A complete WFS renderer consists of a small number of such modules, attached to the room walls.

3.2.1. The model-based approach

Current technical implementations of 3D-rendering schemes are based on models of room impulse responses, containing delay lines for the simulation of early reflections, and a module for the generation of a diffused late reverberation. Direction and spatial distribution of the audio sources are reproduced with the aid of head-related transfer functions (HRTFs) and then often restricting the reproduction mean to headphones.

The model-based approach can also be applied to Wave Field Synthesis, as shown in Figure 30. The echo taps are connected to IIR filters, simulating specific wall absorption characteristics. Direct sound, delayed discrete echo sources, and the output of the late reverb filter, which consists usually of appropriately cascaded IIR allpass and comb filters [90], are connected to delay lines. As explained before, the parameters of these lines are chosen such that the sources appear at their correct positions in the virtual environment, simulating spherical or, in case of far sources, plane waves. The individual outputs, which correspond to the speaker channels of the array, are summed and passed to a bank of pre-filters that are designed in order

to compensate the frequency response characteristics of the room where the arrays are installed, the listening room.

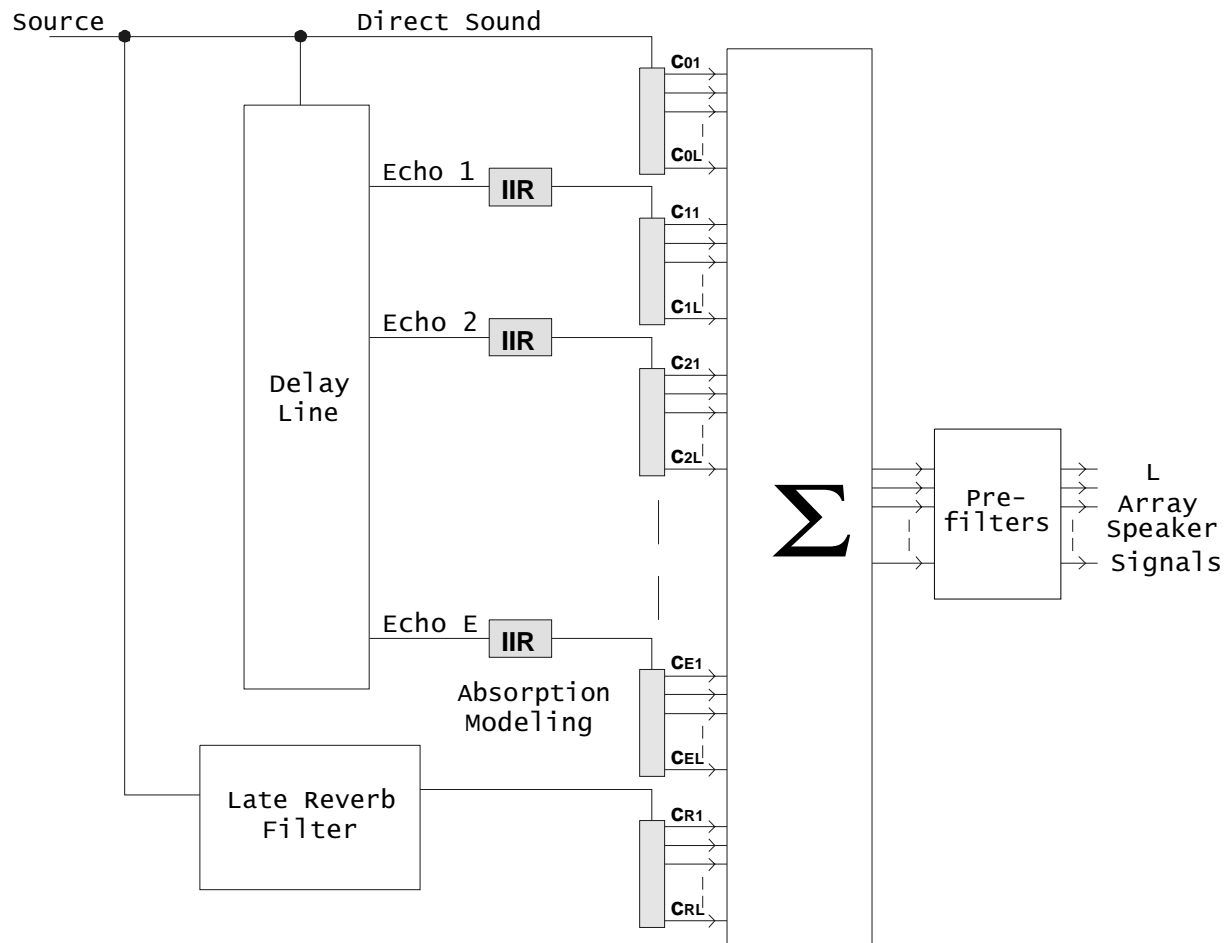


Figure 30 Model-based Wave Field Synthesis system

The main advantages of a model-based system are scalability, i.e. the complexity of the model can be adapted according to the available processing power, and the fact that the parameters can be recalculated with moderate processing cost when source positions change in real-time.

3.2.2. The data-based approach

In the data-based scheme, each source signal is convoluted with long FIR filters, as many filters as the number of channels of the speaker. See Figure 31 for the case of n sources.

The FIR filter coefficients are directly computed from measured or simulated room impulse responses. A very high degree of realism can be achieved, if, for example, the environment that should be reproduced is measured with suitable microphone arrays. A procedure for that measurement has been developed in [125]. The goal of the procedure described there is to capture the acoustics of a bounded sector of a possibly large recording room, with respect to a limited number of sound sources

that remain at arbitrarily chosen, fixed positions. To give an order of magnitude, in the case of an area of e.g. 4 m. times 5 m., 225 impulse responses per source are recorded, each of length 10 Ksamples, so that about 2MB of data are sufficient to characterize the acoustics completely. In order to compute the filters that are required for the array, back- and forward-propagating waves must be separated, to be recovered by array elements attached to the front and back, or left and right walls in the listening room, respectively (see again [125]).

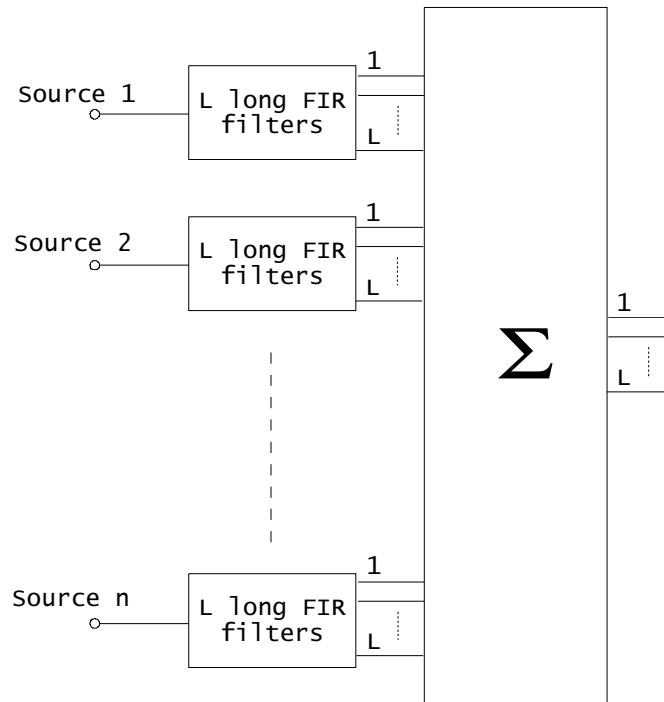


Figure 31 Data-based Wave Field Synthesis system

3.3. WFS, MPEG-4 and ThreeDSPACE

In several ways MPEG-4 corresponds to the requirements of a correct WFS system implementation, since it allows to code all the necessary information in a high-level format, in order to avoid waste of bandwidth through a structured description. This of course implies a more demanding effort at the decoder side, since the "lower-level" samples must be recovered by computation using one of the two models presented above.

With respect to ThreeDSPACE, there are two possibilities to code in MPEG-4 the WFS information, corresponding to each of the two approaches. In a first way, to be exploited for the more realistic data-based approach, it is possible to transmit the whole data, describing a certain number of sound sources, once over the channel; this can be done for instance including responses as tables of an SA object. Once forwarded to the filters, these data can be deleted from the decoder memory by MPEG-4 stream commands and only coded monophonic sources are further received. Responses can also be updated dynamically in the same way.

Otherwise data can be computed out of the existing geometric or perceptual environmental descriptions that are supplied by MPEG-4 version 2 AABIFS (this method is suitable for the model-based approach). In this case the BIFS interpreter must process the control information and supply to the WFS system the same type of data that are made available by the data-based approach.

Since the ThreeDSPACE system is supposed to be made compliant with MPEG-4, and then able to support version 1 and version 2 nodes, both methods must be available at the same time at the end terminal.

4. The ThreeDSPACE Actual System

This section describes the practical implementation of the complete ThreeDSPACE system. At the beginning hardware and software requirements are summarized. Then potential solutions are evaluated and among them the final system architecture is presented together with practical criteria that led to its definition.

4.1. *Hardware and Software Requirements*

Merging together different and demanding technologies like Structured Audio, BIFS and WFS results in a very flexible, powerful and complex system.

It is necessary to have at least a preliminary evaluation of the overall complexity to estimate a reasonable partition of the complete system.

4.1.1. Overall System Complexity

A complete BIFS decoder is necessary supporting at least the Systems Audio Profile (essentially the subset of Audio-related nodes, [55]). It is necessary to have spatialized objects, and then the very low-complexity level is not usable (see [82]). At the moment when specification of the project was laid down, it was not possible to rely on a stable version 2 conformance, and then version1 tables had to be considered.

To have the possibility to spatialize 4 to 16 objects independently it is necessary to implement the Low or Medium Level of complexity. In version 1 of AudioBIFS (and also in version 2 indeed) most of the computation is used for actual spatialization and, in a minor amount, for potential sampling-rate conversion. A separate analysis is necessary for AudioFX, of course. Low Level requires 4 spatialized objects to be decoded and approximately 16 MFlops for S-R conversion; Medium Level requires 16 spatialized objects to be decoded and 64 MFlops for S-R conversion. From literature it is not difficult to argue that spatialization of one object by WFS requires the complete power of a mean DSP device.

For AudioFX nodes, in the case of Low Level the requirements are 10M of instructions or operations that it is possible to classify as "simple" (potentially executable in a single operation on common processors or DSPs) and 2.6M of "complex" functions, where methods like filters, interpolations or effects cannot of course be implemented without using several simple instructions. In the second case, the Medium Level, the requirements are 30M of simple operations and 9M of more complex functions. The experience of SAINT can teach that at least 100 MFlops are necessary to implement with a good quality (of course much depends on

how these macro operators are implemented) the Low Level on a Pentium processor, while as much as 400MFlops could be necessary for the Medium Level.

When one wants to stay compliant with the first version of MPEG-4, it is possible to imagine a typical, or some typical, spatialization environment that can be stored directly at the decoder side. This was the decision taken for a first release of ThreeDSPACE, and this decision can allow to rely on the Low Level, which is characterized by poor memory demand. It is anyway expected that Conformance for version 2 will keep more or less the same approach of the first one, and then the memory necessary to dynamically encode and transmit WFS schemes will require the implementation of a, so far, theoretical Medium Level for version 2.

Finally, it must not be forgotten that, if ThreeDSPACE does not include work on natural Audio decoders and uncompressed sources are used, a potential application of the project (see last Chapter) will further require to support some Audio-related Profile.

Putting all the figures together it is not difficult to realize that the Low Level could be hardly implemented on a standalone last generation processor, while the Medium Level is well beyond such capabilities.

4.1.2. Considerations on Hardware/Software Practical Issues

The reason that makes WFS and MPEG-4 compatible is that AudioBIFS is the most efficient, high-level coding scheme for a demanding technique as WFS, where hundreds of channels are necessary to reconstitute the sound field. Once this advantage is achieved through MPEG coding, it is natural to delay the generation of the so many channels until the latest stage possible in the processing chain, in order to eliminate the overwhelming required bandwidth. In particular, cabling to the loudspeaker is absolutely not practical if the channels are produced outside them. The choice to have active loudspeakers with DSP power inside to generate the WFS channels "in-place" is then quite straightforward.

This first step in the definition of the system corresponds then to cut the non-normative processing of the *Sound* or *DirectiveSound* nodes into two parts, where the very last one is done inside the loudspeakers themselves. In this way a considerable part of the complete scene decoding, the real 3-D Audio spatialization, is confined outside the main processing block(s) and constitutes a standalone block in itself.

For what has been said so far, all the rest of the processing can be carried on by the BIFS/AINT extended virtual DSP; it is then the case to decide if the virtual DSP can be made run on the same CPU of the decoder/compiler, as tested in its preliminary release, or effectively separated as theoretically possible. From the previous subsection it is noticeable that, aiming at version 2, and then at the Medium Level, at least 500 MFlops are necessary for scene decoding and AudioFX processing, without considering the potential pre-processing that may be necessary for AABIFS nodes.

At the moment of the ThreeDSPACE specification, no devices were available that could provide such an amount of processing power, even if they could have been easily foreseen already for the first phase of the project. In any case another

consideration led to the final system, and this is the fact that BIFS is a high-level coding framework where Audio represents a complex issue but it is anyway a small percentage of the processing power that could be foreseen for video decoding, not to say for video synthetic generation and SNHC spatial composition [51].

From this consideration the decision came to exploit the full potential of the SAINT solution, in order to avoid the saturation of the decoding host operating environment and to leave in the system free resources for addition of further decoding capabilities and user interaction.

It results then an overall system composed by three main blocks, where the processing is divided among three different CPUs: what make this system sustainable is the fact that there is no feedback among the parts, i.e. data follow a monodirectional path from the MPEG decoder through the virtual DSP into the array of loudspeakers.

4.2. *The Loudspeakers*

The industrial partner of ThreeDSPACE, Studer Professional Audio, has an enormous experience in DSP programming, above all for Motorola (for fixed-point) and SHARC (for floating-point) platforms. In this sense the choice of the DSPs and of the configuration of the array of loudspeakers has been left to them.

At the first stage the whole demonstrator has been imagined composed by 6-8 array elements, each of 1.28 m. length. Required inputs are an 8-channel, single wire/fiber audio interface according to the ADAT standard, and an RS422 serial control interface. The array elements are connected to a chain, such that each module receives the same signals. The arrays contain 16 separate channels, with 32 tweeters, having a distance of 4 cm. to each other; two adjacent tweeters are connected in parallel. The low frequency band is supported by eight woofers, the signals of which are derived from the 16 outputs by means of analog crossover circuits.

The first estimations showed that the Motorola DSP56307 (100MHz, 48Kx24 internal data memory, filter coprocessor) is capable of generating all the 16 signals for the array element, but only for one or two sources at maximum. It was then necessary to aim at a modular approach, allowing as many DSPs to be plugged in as sources exist. Nearly no glue logic is required, except some external SRAM in case of a high-quality data-based system.

At the beginning, the control data were conceived as being distributed internally via the 8-bit host interface, and connected to the PC (RS422) using a microcontroller.

To limit at a minimum the interconnection complexity, in a second stage it has been decided to exploit the last ADAT channel as control channel, where control information for the seven audio channels and for the overall room are multiplexed in a TDM format to achieve the same sample rate (48 kHz) of the audio information. The format of the eighth ADAT channel will be described later in Table 7.

4.3. *The MPEG-4 Decoder End*

The decoding terminal of MPEG-4 must contain a network connection and have the capability to unpack the coded bitstreams and transform it into a usable format.

Moreover, for the case of SA and AudioBIFS, the decoded material must be parsed and compiled to generate the program whose execution will constitute the real decoding. All these are typical general-purpose issues and are difficult to be implemented on platforms other than typical workstation or PC ones. In particular the compiler task requires a composite mix of fast data management, string processing, memory allocation-deallocation and numeric format conversion; all of these features are hardly executable by a processor that is not general-purpose.

Between PCs and Unix-based workstations a number of factors made the first preferable. It is undeniable that workstations represent a more solid and reliable platform for developing complex systems; moreover, the real-time event management provided by major PC operating systems was quite not affordable until a few times ago. But apart from that some practical and technical aspects are in large favor of PC environments.

First of all, all the major sound board producers have chosen to develop their top line products for PC platforms, considering that the price of workstations and the cost of development are much higher than for PCs. Using this latter, the price can stay competitive with professional musical synthesizers and pre-production mixing consoles (a few thousands of dollars), while the former solution easily increment the total price of a factor 4 or 5.

Secondly, and consequently, an even greater number of interesting software audio applications are available on the PC-related market, and this can consistently speed-up some parts of the implementation task

Finally, MPEG-4 requires enormous amount of interaction and data manipulation, that means interfaces and APIs; modern compilers for PC are much more oriented towards this kind of user-friendly applications than the Unix frameworks.

It is not a problem to admit that, since MPEG-4 Systems was developed in large part by expert groups coming from the PC multimedia and video game worlds, the resulting standard could not have been better supported, at least at its higher level layer, than by a PC machine.

Drawbacks in robustness and real-time speed can be partially overcome by the development of a one-way processing chain, where data transmitted to the sound board are never recalled back, and some interaction can be made available directly to the extern, processing board. To improve even further the situation, new releases of PC operating systems privilege the real-time aspect of computation, and then the obtained performance is coming closer and closer to that of professional audio workstations.

4.4. *The Physical DSP Platform*

Enclosed between the more or less forced choice of the platforms for the rendering terminal and the MPEG-4 terminal, the choice of the physical DSP platform to support the execution engine of SAINT is a delicate one. On one hand it should guarantee a direct connection to the ADAT input of the DSP board of the loudspeakers, on the other it must be a well supported PC board, with as far as possible the possibility to develop for it, and not only to exploit existing development environments to create sounds and music.

A first evaluated idea has been that to develop a custom board, based for instance on the SHARC DSP. The development time for such a solution has been almost immediately considered overwhelming in comparison to the resources of the project and to the foreseen medium term exploitation of a potential product. In particular the development of an appropriate SDK for a Windows-based operating system requires an effort well beyond that of a small team concentrated on another primary objective.

Existing professional audio and/or DSP boards range widely in price and processing power; a *sine qua non* condition was of course the investment in a technology that could guarantee in the medium term the theoretical 500 MFlops necessary to support the required MPEG-4 complexity Level. In this sense all the consumer or semi-professional low-cost boards provided by mass-market vendors have been discharged, because of their low processing potential, lack of adequate I/O bandwidth and proprietary access to development utilities.

Some potential solutions will be now briefly described; technical data refer to situation at mid-1999, while of course updated releases of boards and processors are on the market. Foreseen evolution has been considered at the evaluation time but it has not considered as a criterion for the choice.

4.4.1. Professional Audio Cards

Among the considered solutions, many are the PC-based processing boards produced by electronic musical instrument companies like Roland and Yamaha. In particular Yamaha provide a class of powerful sound boards, like the SW1000 family, a multi-processor board conceived for digital mixing and hard disk recording. Despite the competitive price, promising capabilities and adequate I/O bandwidth and format, these cards are not supported by producing companies with development SDKs and APIs (both Yamaha and Roland exploit proprietary DSPs), so that the development of an application outside the scope of the target market area (that is not narrow indeed) is practically impossible.

4.4.2. CreamWare Pulsar Board

A PC sound board that is widely accepted by audio researchers is the Pulsar, by CreamWare [126]. The Pulsar board provides a conspicuous amount of processing power, thanks to its 4-SHARC architecture and good I/O capability. Among its quite good hardware characteristics, the ones that made it considered for use in ThreeDSPACE are:

- PCI connection
- 4 x 60 MHz SHARC DSPs
- 20 I/Os (2 x ADAT, S/PDIF, analog)
- 32 bit internal bus resolution
- 24 bit I/O resolution
- MIDI-interface (eventually useful for control)



Figure 32 The CreamWare Pulsar Audio board

If it is true that direct support for development on the Pulsar board is not supplied by CreamWare, on the other hand the SHARC DSP is well-known among developers, many development tools are distributed by Analog Devices and third party companies develop many multi-processor boards similar in concept to Pulsar and that could be used to assemble the necessary application code. The architecture of the Pulsar card is very simple, as noticeable from the picture in Figure 32.

What is not attractive in the potential design flow is the cost of the development environment itself. Multi-SHARC DSP boards are quite expensive (several thousands of dollars, 5 to 6 times the cost of the Pulsar board) and the low-cost development environment provided by Analog Devices is not exploitable because of its single-processor conception.

4.4.3. Merging Technologies Mykerinos Board

A completely different approach to Audio computation is the one proposed by Merging Technologies, which provides the Mykerinos sound board based on the Philips TriMedia multimedia processor. Even if the TriMedia has been originally conceived for video MPEG-2 decoding, its V-LIW 32-bit core makes it a powerful DSP with interesting Audio and Video I/O formatting units. The important characteristic of this board is that it is based on a single TriMedia processor, since the V-LIW core alone can provide the necessary power for a considerable amount of processing. The block diagram of the Mykerinos board is shown in Figure 33.

Its main features are:

- PCI connection
- 120 MHz TriMedia processor (240 MFlops sustained, 480 MFlops peak)
- 16 ADAT I/O channels (2 x ADAT) + analog daughter cards
- 32 bit internal resolution
- 24 bit I/O resolution
- On-board sync connectors for video data

These features are then very similar to those of the Pulsar board, even if a single TriMedia has some limitations, at least in theory, in sustained processing power in comparison with four SHARCs (TriMedia will be discussed briefly in the next section).

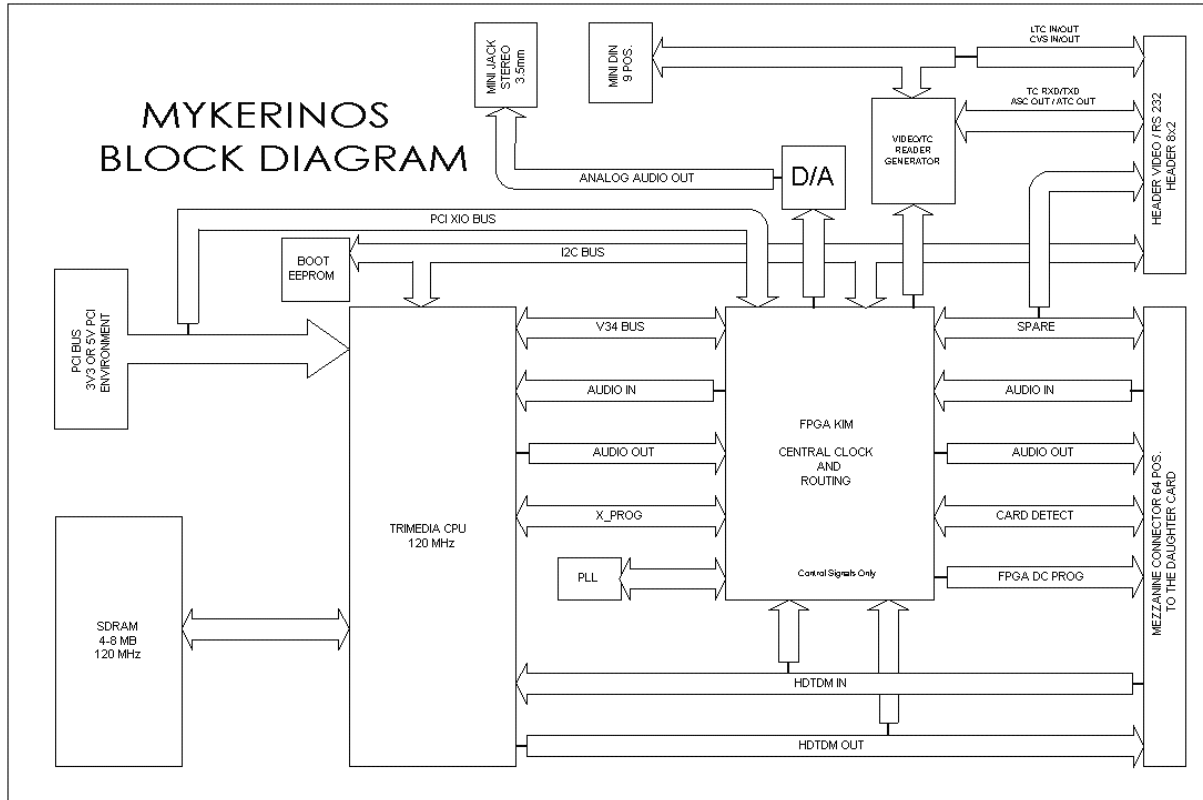


Figure 33 The Mykerinos soundboard block diagram

In this case the board is conceived in a way that its PCI interface is transparent to the programmer in comparison with the TriMedia evaluation board, and then the official TriMedia SDK can be exploited that is available at an extremely low price, as for every development kit of the type.

4.5. The Final System Structure

Even if the Mykerinos board is not as known and used as custom ones or even as the Pulsar board, the decision was to adopt it as the DSP platform, for a series of reasons that go beyond the simple technical features of the card but that involve ease of development, human interaction with producers and software issues. These reasons, given an almost unnoticeable difference in technical features, are:

- the great efficiency of the TriMedia C/C++ compiler, where many resources have been invested by Philips and that guarantees a solid development environment without having to go down to assembly programming.
- the lack of a C++ compiler for the SHARC, and the rather low usage that is reported of the C one in literature, which does not make it as affordable as the TriMedia one, used by every developer as a daily tool.

- the possibility to have an already debugged and working API for the PC-to-Mykerinos interface, kindly provided by Merging Technologies
- last but not least, the possibility to have a direct human interaction between the ThreeDSPACE team and the Mykerinos team, being Merging Technologies R&D division very near to the Integrated Systems Laboratory, where SAINT is developed. This can provide an invaluable and fast progress in the debugging and test of the system.

As a consequence the final system is composed of the three main blocks represented in Figure 34. The general purpose Intel Pentium processor, driven by the WindowsNT operating system, receives the MPEG-4 bitstream as input and it is linked through the PCI interface to the Mykerinos board and to the TriMedia processor. Data that are transmitted along this interface are configuration data to instantiate the virtual DSP on the TriMedia, and then byte-code for the SAOL programs to execute. Additional information is transmitted in runtime in SASL format for the case of streaming data and/or interaction. The ADAT output of the board is fed into the array of loudspeakers, where Motorola DSPs take care of the necessary final stage of processing.

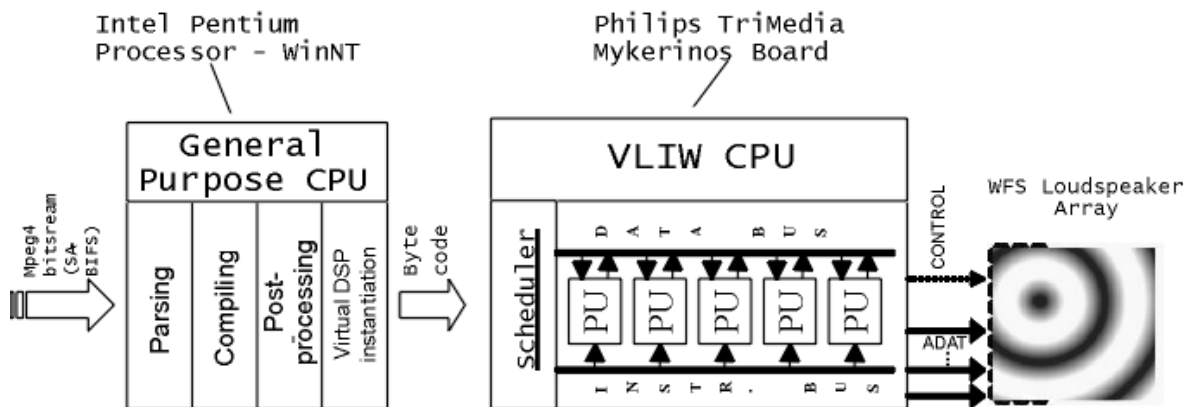


Figure 34 High-level block diagram of the complete ThreeDSPACE Audio system

As stated earlier, the first demonstrator of ThreeDSPACE contains some simplifications, since the system is not ready yet to support at full power the complexity required by MPEG-4 Medium Level. Only one of the two ADAT optical links made available by the Mykerinos board are actually used; of the eight available channels, the first seven are used for monophonic audio channels, the eighth one has been momentarily dedicated to transmission of the control information. Audio is transmitted at a sampling-rate of 48 kHz and block length is 64, which gives a control rate of 750 Hz. Each sample is 24-bit long.

To avoid different formatting on the control channel, control information is simply packed in a sample-like way as shown in Table 7.

The sync field is the only one with a value different than 0 in the first bit and is set to 0x800000; all other values must be scaled to fit into the range of 0x000000 to 0x7FFFFFFF.

Table 7 Format of the control channel over the 64 24-bit samples

Field no.	+0	+8	+16	+24	+32	+40	+48	+56
0	Sync	Source0 Par0	Source1 Par0	Source2 Par0	Source3 Par0	Source4 Par0	Source5 Par0	Source6 Par0
1	Room Par 0	Source0 Par1	Source1 Par1	Source2 Par1	Source3 Par1	Source4 Par1	Source5 Par1	Source6 Par1
2	Room Par 1	Source0 Par2	Source1 Par2	Source2 Par2	Source3 Par2	Source4 Par2	Source5 Par2	Source6 Par2
3	Room Par 2	Source0 Par3	Source1 Par3	Source2 Par3	Source3 Par3	Source4 Par3	Source5 Par3	Source6 Par3
4	Room Par 3	Source0 Par4	Source1 Par4	Source2 Par4	Source3 Par4	Source4 Par4	Source5 Par4	Source6 Par4
5	Room Par 4	Source0 Par5	Source1 Par5	Source2 Par5	Source3 Par5	Source4 Par5	Source5 Par5	Source6 Par5
6	Room Par 5	Source0 Par6	Source1 Par6	Source2 Par6	Source3 Par6	Source4 Par6	Source5 Par6	Source6 Par6
7	Room Par 6	Source0 Par7	Source1 Par7	Source2 Par7	Source3 Par7	Source4 Par7	Source5 Par7	Source6 Par7

The source parameters are decoded from the parameters of the nodes Sound or DirectiveSound in the following way:

0. location[0] (x position)
1. location[1] (y position)
2. location[2] (z position)
3. direction[0] (direction along x)
4. direction[1] (direction along y)
5. direction[2] (direction along z)
6. tbd
7. tbd

The room parameters are taken from the parameters of the node AcousticScene in the following way:

0. size[1]*size[2]*size[3] (volume)
1. ReverbTime[0]
2. ReverbFreq[0]
3. ReverbLevel[0]
4. ReverbTime[1]
5. ReverbFreq[1]
6. ReverbLevel[1]

In the case of Sound node, these parameters are not available in the scene description (in version 1 there is no direct relationship between the video scene and

spatial audio) and some predetermined sets of parameters can be used. In all cases some pre-processing is necessary to be able to transmit pre-calculated or recorded room impulses, as previously explained. Figure 35 shows in more details the block-diagram of the rendering system of ThreeDSPACE.

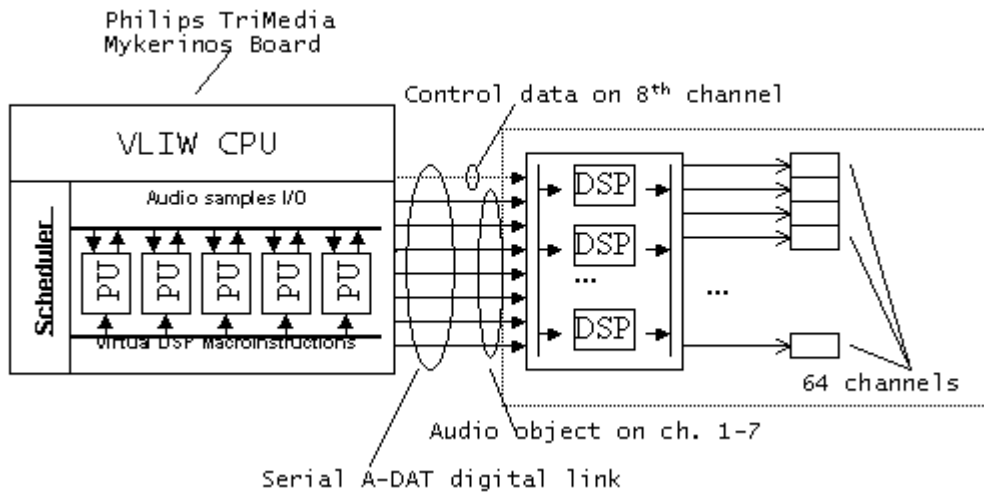


Figure 35 Block-diagram of the ThreeDSPACE rendering system. A monodirectional optical link is enough to guarantee the necessary information flow from the BIFS/AINT execution engine to the active arrays of loudspeakers

5. Porting SAINT on a VLIW Architecture

One of the fundamental issues on which the whole SAINT architecture has been conceived is the possibility to implement and optimize it at a reasonable cost on modern superscalar architectures, with the concrete possibility to actually exploit the parallelism they provide. The porting of the SAINT virtual DSP to the VLIW TriMedia processor has represented an important test bench for the SAINT concept and for the ThreeDSPACE system as a whole.

In this section the TriMedia processor will be briefly introduced and then the implementation of SAINT on this device will be described. Some optimization techniques will be shown that finally validate the vectorial instruction set approach and the feedback analysis on which many resources have been spent.

5.1. The TriMedia Multimedia Processor

The TriMedia processor was disclosed to the market in 1996; its most innovative feature in the domain of multimedia was that it is built around a VLIW unit called the DSPCPU. As this name says, this unit is not designed as a completely general purpose CPU, but with the double functionality of CPU and DSP. In the same chip only its core processing master unit presents a VLIW architecture. The first version of this processor was conceived most of all for MPEG-2 applications, but one of the claimed main targets is also real-time audio processing and synthesis (wavetables, FM, physical shaping, [127]). The general architecture of TriMedia is shown in Figure 36.

The central processor is also in charge of coordinating all the activities on the chip. This unit runs a small real-time operating system that is driven by the interrupt from other units. DMA-driven audio and video input/output units operate independently and properly format data. The TriMedia also contains independent specific units to deal with common video decoding tasks like a VLD coprocessor for Huffman decoding or a 5-pixel interpolation filter.

The internal data bus connects all internal blocks together and provides access to internal control registers (in each function unit), external SDRAM and the external PCI bus. The internal bus consists of separate 32-bit data and address buses, and transactions on the bus use a block-transfer protocol. On-chip peripheral units and coprocessors can be masters or slaves on the bus.

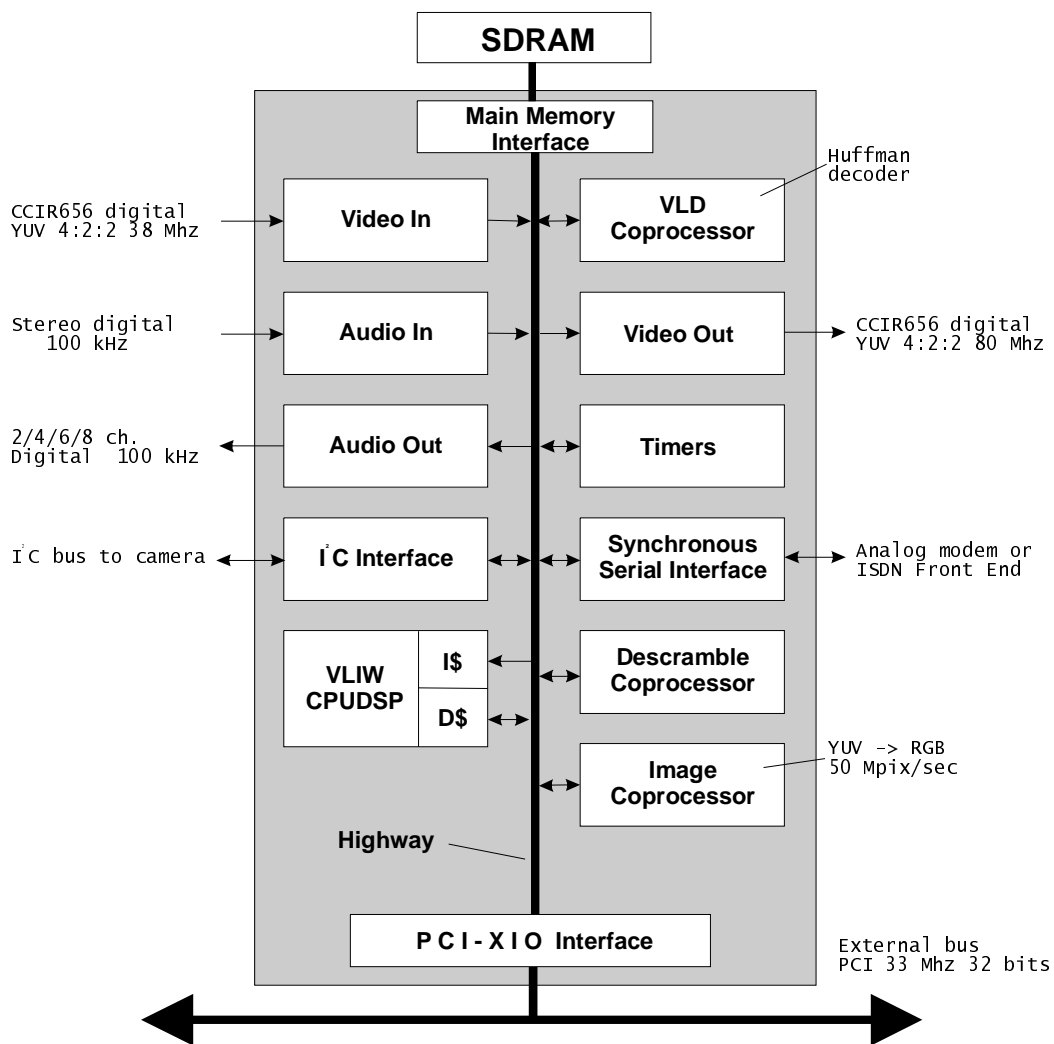


Figure 36 The Philips TriMedia architecture

5.1.1. The VLIW core

The heart of TriMedia is its 32-bit DSPCPU core. The DSPCPU implements a 32-bit address space and 128 general-purpose 32-bit registers. The registers are not separated into banks; any operation can use any register for any operand. The core

implements a VLIW architecture that provides up to five simultaneous operations to be issued. These operations can target any 5 of the 27 functional units in the DSPCPU. The functional units are described in Table 8. Latency is the time required to obtain the result whereas recovery is the number of cycles between two identical operations can be issued. Divisions and square roots are very costly and have to be avoided as far as possible. These units include integer and floating-point arithmetic units and data-parallel DSP-like units. SIMD operations are possible on 8-bit or 16-bit integer data. Being the SAINT target the floating-point format, this SIMD add-on and five integer units have little impact on the application: they can be useful for video applications.

Floating-point additions, multiplications, loads and stores require 3 cycles. This has a strong impact on optimization. In the TriMedia there is no special hardware to control loops, as in many common DSPs instead, so loop control requires an inline comparison on its associated decision branch. Branching requires 3 cycles.

Each operation is a RISC-like instruction guarded by a condition (ex: if R0 = 1 R1 = R2 + R3).

Table 8 The 27 TriMedia functional units

Functional Units	Quantity	Latency / Delay	Recovery Cycle
Constant	5	1	1
Integer ALU	5	1	1
Load / Store	2	3	1
DSP ALU	2	2	1
DSP Multiplier	2	3	1
shifter	2	1	1
branch	3	3	1
Multiplier Integer/Float	2	3	1
Floating ALU	2	3	1
Floating Comparator	1	1	1
Float Square Root / Division	1	17	16

5.1.2. Memory structures

Two different caches are also present: 32 KB for instructions and 16KB for data. The cache block is 64 bytes long. The data cache serves only the DSPCPU and is controlled by two memory units that execute the load and store operations issued by the DSPCPU. When a miss occurs, the data fills the block containing the requested word from the beginning of the block. The CPU is stalled until the entire block is transferred in the cache. Special Data-cache operations allow the user to lock some regions of the cache and to keep the cache coherent with memory. Memory structures and main buses are summarized in Figure 37.

5.1.3. The Programming Environment

The main programming languages for the TriMedia are C and C++ (C++ is actually cross-compiled to C). Assembly language for the VLIW instruction set is too complicate to be well managed and then it is understandable how fundamental is a robust and optimized compiler. An important number of APIs is provided with the development kit ranging from host communication protocols, to stream management or multimedia unit control.

The first step to build TriMedia applications is the C preprocessing and representation of decision trees: the program flow is decomposed in subtrees to allow branch prediction and probability evaluations. The instruction scheduler browses these trees and generates VLIW instructions. During this process, the scheduler adds some conditional code. According to the TriMedia programming manuals [127], optimizations for parallelism and are performed at this step. It will be shown that this is often a false parallelism, and that the intrinsic SAINT structure and handcrafted optimization can get rid of this and produce exploitation of real parallelism.

The last step after linkage is the instruction compression, which increase code density and allows improving instruction transfer between cache and external memory.

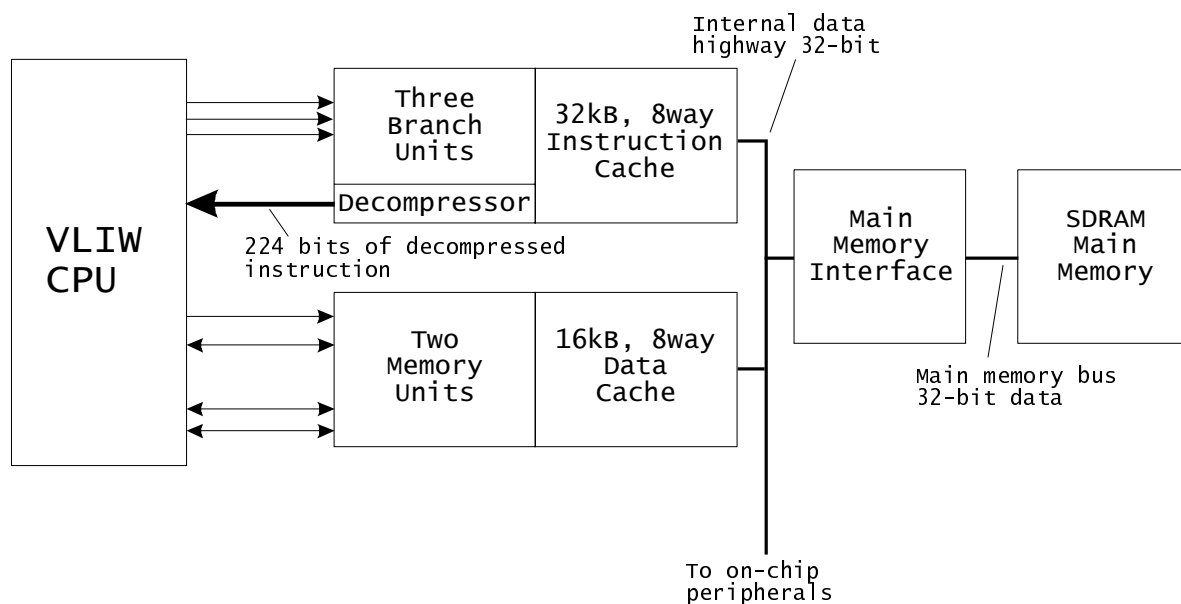


Figure 37 The TriMedia memory structures and relating interconnections

5.2. Code Optimization for the TriMedia

This section describes the techniques exploited to port and optimize the SAINT vectorial instruction set and to tune performance. It describes various optimization methods supported by the TriMedia compilation system as well as techniques for exploiting the fine-grain parallelism of the TriMedia architecture. Given the architecture of SAINT, it makes sense to try first of all to optimize the implementation of the vectorial instruction set, being the scheduler a separate and

low cost task, which moreover does not offer many indications for a data-level parallelism.

Even if the porting of the scheduler and the allocation of the SAINT virtual DSP on TriMedia required some work, most of all to understand and exploit the Mykerinos API, it is only interesting to report the work of optimization of the vectorial instruction set; this task corresponds to the implementation of a vectorial DSP library for the TriMedia, since an already available one like the mentioned Optivec was not found.

Once more, in order to explain more efficiently the different techniques, it is better to rely on an example, in this case the *lopass* filter.

In SAOL, in addition to the possibility to implement FIR and IIR filters by the corresponding core opcodes, four filters are defined (*lopass*, *hipass*, *bandpass* and *bandstop*) for which the functionality is defined but the algorithm is left open to the implementer; consequently they constitute good candidates for an entry in the MPEG-4 conformance complexity vector and in the SAINT instruction set; the only constraint to be satisfied is the respect of an SNR mask defined for conformance test that has a slope of approximately 20 dB /octave after the passband frequencies and a maximum passband ripple of 6 dB.

After some practical comparisons, a Butterworth's IIR filter has been selected both for his stable ripples and the low order required (5) to satisfy the constraints of the filter. For computational reasons an IIR of order 4 has been first implemented.

Given this information, the SAINT C++ code used to implement the *lopass* macroinstruction for a generic platform is almost completely represented in Figure 38.

```
// loc_static_arrs[0][i] are the b coefficients of the IIR filter
// loc_static_arrs[1][i] are the a coefficients of the IIR filter
// loc_static_arrs[3][i] is the delay line containing last outputs
// par_list->get_elm(1)->value[i] contains the input to the filter

for(i=start; i<end; i++) {

    float temp = 0.0;
    float *samples = par_list->get_elem(1)->value;

    // accumulate products
    for(int r=0; r<num_b; r++)
        temp = temp + loc_static_arrs[0][r]*samples[1+r];

    for(int s=0; s<num_a; s++)
        temp = temp - loc_static_arrs[1][s]*loc_static_arrs[3][1+s];

    left->value[i] = temp/factor;    // output result

    loc_static_arrs[3][cur] = left->value[i];

}
```

Figure 38 General C++ code for the SAINT *lopass* macroinstruction

In the following subsections the main optimization techniques that have been exploited will be introduced. Some of them are peculiar to the TriMedia, some others are instead typical of common parallel programming.

5.2.1. *Profile-Driven Compilation*

The TriMedia compilation and simulation system provides a performance tuning facility based on a compile-profile-recompile cycle. After the program is compiled for the first time including some special profiling information, the program is executed as far as possible in typical conditions of performance. The output of this profiling process is further exploited to assist the second compilation, where information about most used paths and memory accesses is available to better structure the parallelism of the code. It is possible of course to proceed with multiple iterations of this process.

5.2.2. *Grafting on profile information*

It is well known that frequent branching behavior results in underutilization of pipelined and/or parallel processors. To help solving this problem, the TM compiler generates an intermediate representation of a program known as a decision-tree representation. Decision trees are derived from basic blocks; a basic block is a sequence of instructions with no jumps in it, except to the first instruction, and no jumps out except at the last instruction.

A decision tree is similar to a basic block in that the decision tree can be entered only at the beginning. However, a decision tree can have multiple exits. Decision trees are larger than basic blocks and potentially have more fine-grain parallelism that can be exploited during optimization. Grafting increases parallelism within decision trees; this technique replaces any jump with a copy of the destination tree and thus “grows” larger decision trees. As a result, the program size increases but the number of branchings reduces. Also the lines of code to be eventually loaded in the instruction cache increase.

The grafting optimization is also based on the profiling run, and can be added as a further option. The compiler can do a better job of grafting if it has information about decision-tree execution counts and branch probabilities.

Grafting is a technique similar to loop unrolling, but does not reduce the overhead of the loop as manual loop unrolling can (see subsection 5.2.4).

5.2.3. *Loop optimization*

Loop optimization is obtained by moving critical code off the control flow path, so that the inner loops of the program can be reduced to a single decision tree. There are several techniques to achieve loop optimization, and the TM compiler automatically performs most of them.

Strength reduction consists of reducing operations that are more costly to less costly ones, and of bringing them outside loops if their operands are constant. For instance, division by *factor* (17 cycles) in the *lopas* example of Figure 38 is automatically replaced with a multiplication times $1/\textit{factor}$ (3 cycles), where this division is brought outside the main *for* loop. This technique, even if automatically

detected, belongs to tricks well known by programmers and then does not really introduce a meaningful added value.

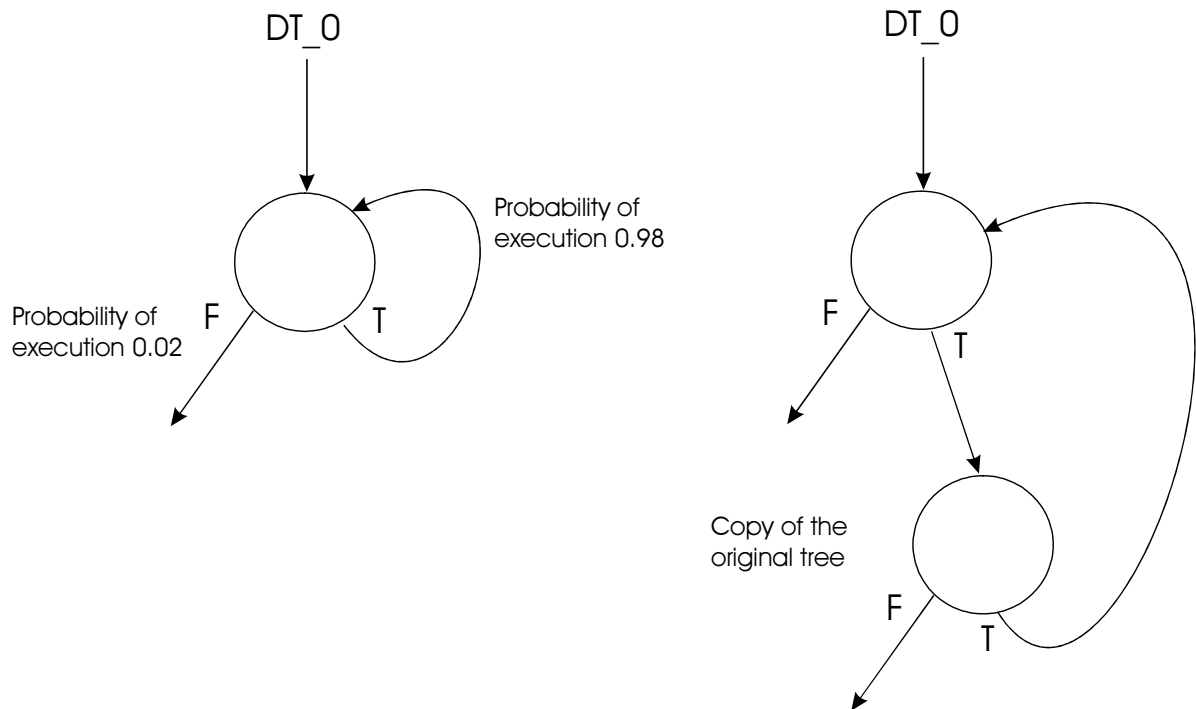


Figure 39 Grafting optimization. The compiler replaces an instruction “go_tree DT_0” with a copy of the tree DT_0, doubling its size and deleting one branching.

Another optimization automatically performed is the conversion of externals and reference parameters to local ones. For instance all the references to *loc_static_arrs* in the example will be replaced by local pointers for which calculation of the indexes is not required at each iteration. This optimization partially compensates for the lack of a dedicated DSP-like address generation unit in the TriMedia.

5.2.4. Manual loop optimization

Loop fusion is another technique well known by parallel programmers. In the *lopas* example, but also in many other SAINT macroinstructions, the main *for* loop over the size of the block contains other smaller loops over the order of the filter. If, like in this case, the number of iterations of this inner loop is fixed and known, it is possible to remove the *for* loops and replace them with the full code. Again this increments the code size, and then the eventual cache overload, so that a trade-off between load time and speed of execution must be investigated.

Loop unrolling can be introduced to further reduce the number of branching in the code. Loop unrolling consists of repeating several times an iteration inside the main loop, if this is not too big, and then increment the step of the loop of the same number of times. Of course, attention must be paid to the perfect understanding of the algorithm. In the case of SAINT, the size of the block, i.e. the size of the vector, is programmable, and then the number of iteration of the main loop cannot be “hardwired”. In any case, since the size of the vector is normally between 50 and

500, a good compromise is to perform for instance a loop unrolling over four iterations, where in the mean 2 operations will be wasted since not executed on valid values of the vector (in this case all vectors must be allocated with a size multiple of 4).

Note that loop unrolling is a specialized version of grafting. In loop unrolling, a conditional jump from a decision tree exiting back to itself is replaced with the code for the decision tree. The main difference is that grafting replaces the jump part of the conditional jump with the destination decision tree but leaves the condition in place, which causes control dependence between iteration of the loop to the next one. In the end, a good understanding of the algorithm can still do better than any statistics-based automatic optimization.

Finally, another well know optimization technique is that of removing vectors with indirect addressing by using pointers with explicit increments. The reason for that is that, using a vector, it is necessary to check the position in the array. It is true anyway that nowadays all the modern compilers are able to detect continuous processing over an array, and this is no more a useful technique to be applied by hand.

5.2.5. *Restricted pointers*

At the compilation time, the compiler does not know if a pointer to an allocated memory area is overlapped with another pointer possibly accessing the same data to modify them in a read/write process. That is to say, the compiler normally assumes that two pointers may refer to the same or overlapping memory locations, i.e. be aliased to each other. This implies that operations of two different statements having those pointers as operands cannot be executed in parallel. However, if it is known that all the pointers point to distinct memory areas and thus never alias, it is possible to convey this information to the TriMedia compiler by declaring these pointers as *restricted*. Based on this information, the compiler decides that different variables and/or restricted pointers do not alias. It is a programmer's responsibility to verify that the assertion is true; proper use of restricted pointers reduces the amount of dependencies and, therefore, increases dramatically potential parallelism (see experimental results in the next subsection).

In the specific case of the considered example, but for SAINT in general, the vectorial macroinstruction intrinsically contains the certainty that memory areas are not overlapped. In fact, detection of feedback is performed by the SAOL compiler before execution of the bytecode, and then potential overlappings are already eliminated and relating instructions treated as sample-by-sample (that again avoid overlapping by making the virtual DSP work on one sample at a time). All pointers used in *for* cycles from *start* to *end* automatically work on non-overlapping memory areas. When delay lines are used, like in the lopass example, data are written and read, but a single pointer is used to access this buffer (in the specific case `loc_static_arrs[3][x]`).

5.2.6. Experimental results

In this subsection experimental results are reported that show how the vectorial instruction-set matches the parallel potential of the TriMedia processor. Results are shown in Figure 40 and Figure 41.

The experiments were made on a large block of samples, to isolate computation and reduce potential overhead coming from too many cache memory refreshes. Vertical axis represents the million of cycles to filter $2 \cdot 10^5$ samples, while the horizontal axis reports the described methods of optimization in the following order:

- Opt.1 Profile-Driven compilation
- Opt 2 Decision-tree grafting
- Opt 3 Loop optimization: Strength reduction
- Opt 4 Loop optimization: Local reference parameters
- Opt 5 Loop optimization: Loop fusion
- Opt 6 Manual Loop unrolling
- Opt 7 Manipulation of pointers
- Opt 8 Restricted pointers

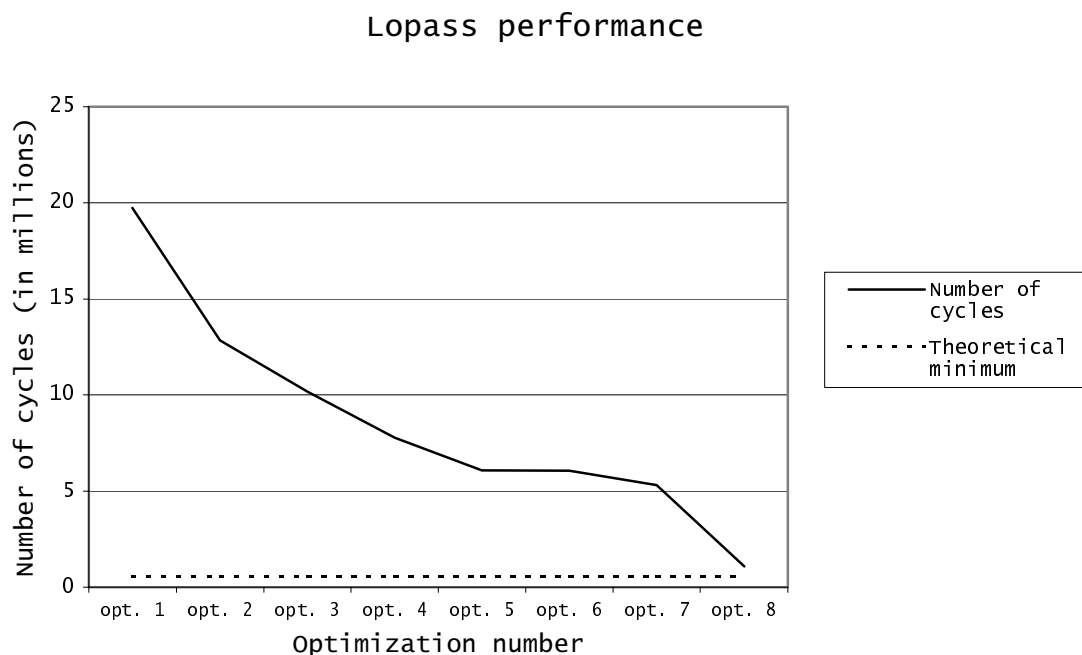


Figure 40 Performance of the lopass macroinstruction on a large block of samples ($2 \cdot 10^5$ to better profile execution on the TriMedia) for different levels of optimization.

Optimizations are cumulated, i.e. in column 2 optimizations 1 and 2 are used and so on. The theoretical minimum in Figure 40 is calculated in a conservative way counting only multiplications and additions; considering that the TriMedia cannot allocate the five slots to floating point operations, it is supposed that at the same time necessary loads and stores can be performed. A theoretical minimum is often difficult to estimate on a parallel architecture and then this comparison must be taken as a not too strict reference, with some margin of error.

On the other side the values reported in Figure 41 are objective, since the vertical axis represents the calculated Instruction Level Parallelism (ILP), which for the TriMedia has a precise upper limit of 5 (no more than five execution slots can be allocated per cycle). In this case the open issue could be if all the operations executed in parallel are necessary or not, but the theoretical limit is precise. Given these two groups of measurements and the potential margins of error they contain when considered separately, conclusions have to be drawn by a comparative analysis of the two figures.

Instruction Level Parallelism

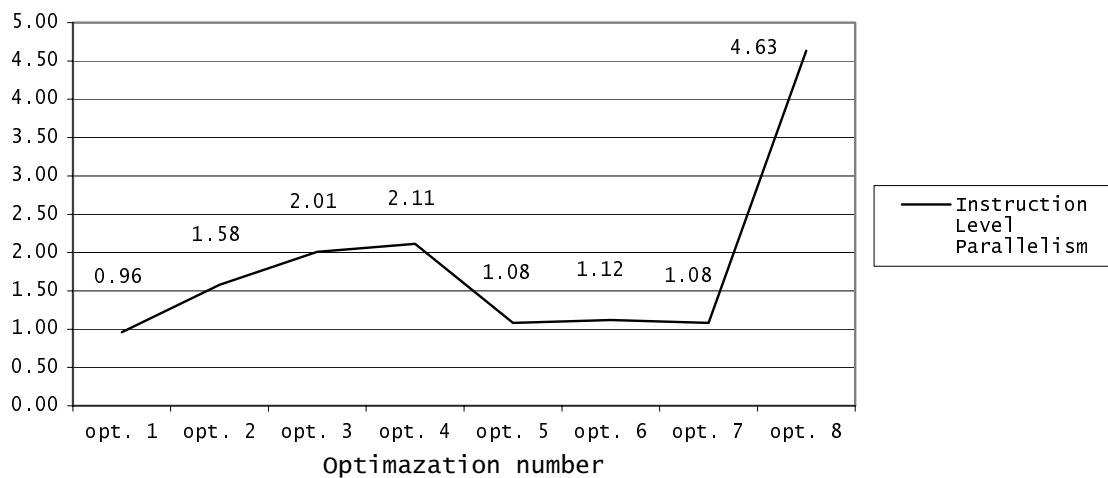


Figure 41 Instruction Level Parallelism (ILP) in the case of the lopass macroinstruction for different levels of optimization.

The first four optimizations introduce a reduction of an about 2.3 times in the number of cycles. This has its counterpart in an increased ILP from 0.96 to 2.11 and then it is evident that there is no reduction in the overall number of executed operations but only the compiler is able to detect some code inefficiencies and to translate them into parallel operations. When some handcrafted code manipulation is introduced, loop fusion and manual unrolling, the number of cycles further reduces but the ILP falls down to nearly 1. This has a simple explanation: the parallelism introduced by the compiler so far was a false one, in the sense that it compensated code written in a too formal way: once these formal redundancies are removed it is clear that only one multiplication or one addition are executed at a time and no true parallelism is exploited at all. Change in the way of addressing arrays does not improve the situation, it just reduces a little the number of cycles. The real improvement comes at the last step, where the structure of the instruction set of SAINT and its execution procedure allow the indiscriminate application of restricted pointers. This special declaration tells the compiler that it can really schedule processing to happen in parallel, and while the ILP jumps by a factor of almost 4.3 at the same time the number of cycles decreases of a factor 4; this means that in this case the processor is really capable to calculate two floating

point operations and two loads or stores in parallel, also being capable at the same time to manage the integer arithmetic of the outer *for* loop from *start* to *end*. Also, saying this with the required caution, the number of cycles is not very far from the theoretical minimum.

5.3. A Vectorial Library for the TriMedia

Of course, not all the SAINT vectorial macroinstructions are essentially based on "simple" operations like additions and multiplications and on delay lines, things that DSP-oriented compilers know quite well. Needless to say that not for all the macroinstructions it was possible to reach a performance near to the theoretical limit and with high degrees of parallelism.

It is possible to group functions implemented for the necessary vectorial library in three main groups:

- functions of easy implementation, like arithmetic and logic operators, the group of filters and so on; for these functions performances like those shown in the previous section has been reached.
- functions with some internal implicit feedback and parametric loops, like some specific effects, *fir* and *iir* macroinstructions, where some open parameters (like the order of the filter, for instance) prevent from a complete unrolling of the functionality and from a complete exploitation of the restricted pointers on real data; for these functions performance in the order of 2-to-10 times the estimated theoretical minimum has been reached.
- functions requiring some calls to C standard functions like mathematical operators like *log*, *exp* or the like; for these functions low ILP has been reached and a theoretical minimum is hard to guess due to the opacity of functions. In the complexity analysis they constitute independent elements that need dedicated benchmarking.

The last group of operations constitutes a delicate one. On one side it is evident that results reported in Figure 42 are not satisfactory, and this is due in main part to the call to the *log* C function that processes one sample at a time and with a low degree of parallelism; this call constitutes a high percentage of the complete execution. Once more it is evident that without a complete vectorial set of operations it is not possible to really speed-up the performance of the SAINT engine on the whole range of functionality.

On the other side, writing vectorial libraries for mathematical and trigonometric operators on a truly parallel processor requires a great skill in numerical computation techniques and could constitute a research topic in itself.

5.3.1. SAINT on the TriMedia processor

Once the necessary macroinstruction set has been optimized for the TriMedia, the whole SAINT virtual DSP has been compiled to run on the TriMedia, while the SAOL compiler was kept running on the Intel-based platform. The API made available by Merging Technologies to interface to the Mykerinos board is very useful in letting the virtual DSP be allocated on the TriMedia by the SAINT post-compiler itself.

At this point, measuring performances on the new platform is not straightforward: the TriMedia simulator cannot be used anymore, because it is almost impossible and not useful to compile the whole application for this processor. The Mykerinos API provides some facilities to monitor and measure the behavior of the processor, but this hardly perturbs the overall performance.

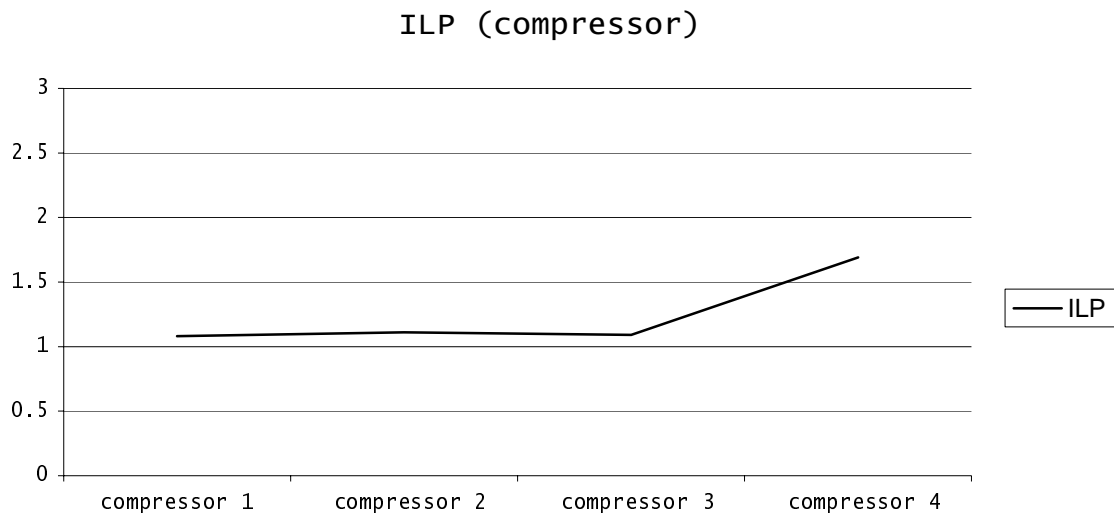


Figure 42 Instruction Level Parallelism for the compressor vectorial macroinstruction. The four values correspond to Opt 1, Opt 3, Opt 7 and Opt 8 of Figure 39 and 40.

In the end, thanks also to a CPU load monitor, the best thing that can be done is to run SA programs that were also used for benchmarking and tests on the PC version and compare the results, to see where exactly the actual real-time limits are. Running some examples like the FM clarinet, the wavetable piano and so on at different degrees of polyphony, it has been possible to verify that performance of the system, based on a 100 MHz TM1100 processor, so far is comparable with that of a Pentium 200 MHz system with the Optivec libraries installed. Moreover, instead of saving output to files, the Mykerinos version of SAINT can produce real-time Audio output formatted in ADAT, like required by the ThreeDSPACE project.

The software still requires optimization. Other than the operators described above, also the scheduler runtime process needs a complete analysis in order to find potential sources of parallelism that could further reduce the interpretation overhead. In any case these results let understand that the approach can be considered successful and that current and future new versions of the TriMedia and of the Mykerinos board, aiming at increasing bus speeds and processor speed in a 64-bit VLIW/SIMD architecture could provide performance near to that of fast PCs, probably enough for the target Level of MPEG-4.

CHAPTER 7. A MACRO-ORIENTED VIRTUAL APPROACH FOR SIMULATION AND DESIGN OF MULTIMEDIA ARCHITECTURES

The MPEG-4 Structured Audio Conformance and SAINT, described so far, originates from a platform-independent analysis of SAOL that hides two main simulation issues: the eventual overhead coming from the execution of a program and, above all, interaction between the program and the physical or virtual device memory.

If the first of these two issues can be often considered not meaningful, in the sense that only an interpreted approach to SA can bring with it an appreciable overhead, the second issue can result in a heavy simplification of a problem that could considerably affect the performance of a program on a specific platform.

The goal of this chapter is to remove the limitation on memory aspects from the SA-based simulation tool, in order to estimate the approximation introduced into the SA conformance specification and, above all, to extend the simulator itself (until here an instruction set one) to a complete architecture analyzer; at that point, given the generality of the SA toolset, the tool becomes capable of supporting the higher level design tasks for all the applications that can be described through SA; in this context SA becomes a high level description language for a block-based design of multimedia architectures.

In the first part of this chapter a macro-oriented approach to memory simulation will be introduced, the resulting tool will be briefly described and experimental results for architecture simulations will be presented.

In the last part of the chapter, after a short comparison with other similar simulators, it will be shown how the proposed tool can be exploited for the design of multimedia architectures and systems.

1. Memory Simulation: a Macro-Oriented Approach

It was said in Chapter 4 that tracking memory accesses is probably the hardest task when profiling an application; this is due to the different types of storage devices, which are characterized by access latencies that range from the few nanoseconds of registers and cache memories to the few milliseconds of virtual memories on a hard-disk. In the case of real-time multimedia applications this latter storage medium must be avoided, since the latencies that it introduces cannot be tolerated by any useful real-time system. It is instead very important to track the behavior of the system in its "bouncing" between the cache memory and the normal dynamic RAM memory, since very often applications are small enough to be contained in the system DRAM but at the same time are big enough to saturate the physical cache memory.

1.1. *Precise Cache Memory Simulation*

In the other reviewed simulation tools the cache simulation is always performed precisely; an exact model of the cache memory (or memories) is used and any access to a datum is tracked; cache misses are compared against a precise cache update policy so that the exact content of this fundamental device is always available. If this approach to memory simulation has the obvious advantage of permitting a precise track of the behavior of the application at the "sample level", it brings with him a certain number of disadvantages:

- a precise cache simulation requires the availability of a precise description of the cache memory and, above all, the availability of the precise update policy in case of a cache miss; if it is true that often caches use similar techniques that are based on the use of data during the program, it is often not possible to know the exact mechanism, for instance if it is completely deterministic (the most remotely accessed data are removed) or partially stochastic (frames of data usages are kept and refreshed). Moreover, cache could be divided into Level 1 (normally on-chip) and Level 2 (normally on-board), and this further complicates the model
- such a complex emulation process forcedly requires a huge amount of computational power and then the process results remarkably slow.
- due to the weight of the simulation, in the past many simulation systems were even divided into tracers and cache analyzers, as described in Chapter 2. The integration into a single tool of an execution tracer and a simulator resulted in such a slow tool to often discourage the designer from using it as a true design tool.

To improve the situation many optimization techniques have been designed and many statistical reduction methods introduced. This approach breaks the concept of exact cache simulation in favor of an approach based on statistically relevant data accesses.

1.2. *Approximated Macro-oriented Cache Simulation*

The whole analysis of imperative languages described in Chapter 4 relies on a basic assumption: operations are divided into classes of operations, where all the members of one class can be assimilated to a behavior in terms of clock cycles and memory usage that may not be precisely the same for all members; for instance, library functions can be decomposed in approximated sequences of instructions so that realistic applications can be monitored.

Moving from a similar assumption, the macro-oriented cache simulation introduced here aims at isolating the relevant data transfers that involve the cache memory and at making abstraction of minimal deviations from the precise cache status; the main purpose of this approach is to consistently reduce the complexity of the simulation without possibly losing an accuracy in the simulation that can be enough for estimations in high-level, block-oriented design flows. Block of data are moved in and out of the cache at an abstract level, discarding small, implementation-dependent control or initialization data and keeping track mostly of variables and tables described through the SAOL code.

In fact, in many phases of a real design process, approximations are often introduced by many tools and, later, when the design goes down to the silicon level, simulations cannot be run without a certain, often relevant, degree of approximation. In the same way, it is often not possible to perfectly preview the behavior of an application, particularly in real-time interactive frameworks. In the end, there is a strong feeling that a certain amount of approximation in the simulation of an application is, if not welcomed, at least an unavoidable element of a realistic and reasonable design flow.

Of course it is also to be expected that such a hardly simplified approach introduces some drawbacks and critical situations. It will be shown in the following of the chapter how programs with many sample-by-sample blocks and/or relevant overhead coming from inefficient coding can lead to remarkable over- or under-estimations of the execution time on a processor, according to the specific platform solution. Since the acceptance of such an error is intrinsic in the platform-independent simulator described in Chapter 4, it is interesting just to verify if the virtual cache model improves the consistency of the simulation or on the other hand it is not useful to arrive to a more meaningful and usable tool. In most cases it has been found that the introduction of a cache simulation brings to more than acceptable margins of error in typical applications and on very different kind of platforms.

2. A Macro-Oriented Virtual Simulator for Multimedia Architectures

The first version of the simulator for Structured Audio programs was implemented as a plug-in to the MPEG-4 SA reference software; this was unavoidable, being it at those times the only more or less complete implementation of the SA standard.

As clearly shown in Chapter 5, saolc is an extremely clear but at the same time inefficient decoder, and then once SAINT was available as a stable tool, the platform-independent profiling tool has been plugged into this latter environment; having both tools an interpreter structure, the porting was nearly straightforward. Since the goal of this porting was above all that of building a complete and efficient platform simulator out of the first tool, the compatibility with the MPEG-4 conformance has been neglected; the new tool was configured to count the operations actually executed by SAINT and not the theoretical number of SAOL operations: in practice, operations optimized to be executed at a slower rate have been kept instead of being updated as in the saolc decoder. In doing so, it was considered that it was now no more necessary to find compatible metrics among all the possible decoders, but instead to build a tool able to simulate in an efficient way audio programs written in SAOL and consequently able to produce effective architectural solutions from the program simulation. The adopted strategy better corresponds to any normal practice of good programming.

2.1. *The Cache Memory Tracer*

The most important task when transforming an already time-dependent tool like the developed one to a complete simulator is the integration of a cache tracer. Once

more, the interpreter structure of SAINT is intrinsically not the best in terms of absolute performance but it allows the straightforward inclusion of many hooks to track in details several aspects of the simulation. In the case of the cache tracer, it is possible to link it directly to the execution engine; nevertheless it is possible to keep the simulation slow-down into acceptable limits exploiting some techniques for fast memory simulation similar to those that have been presented previously in literature [28],[31]. The implemented cache simulation scheme and data structure are shown in Figure 43.

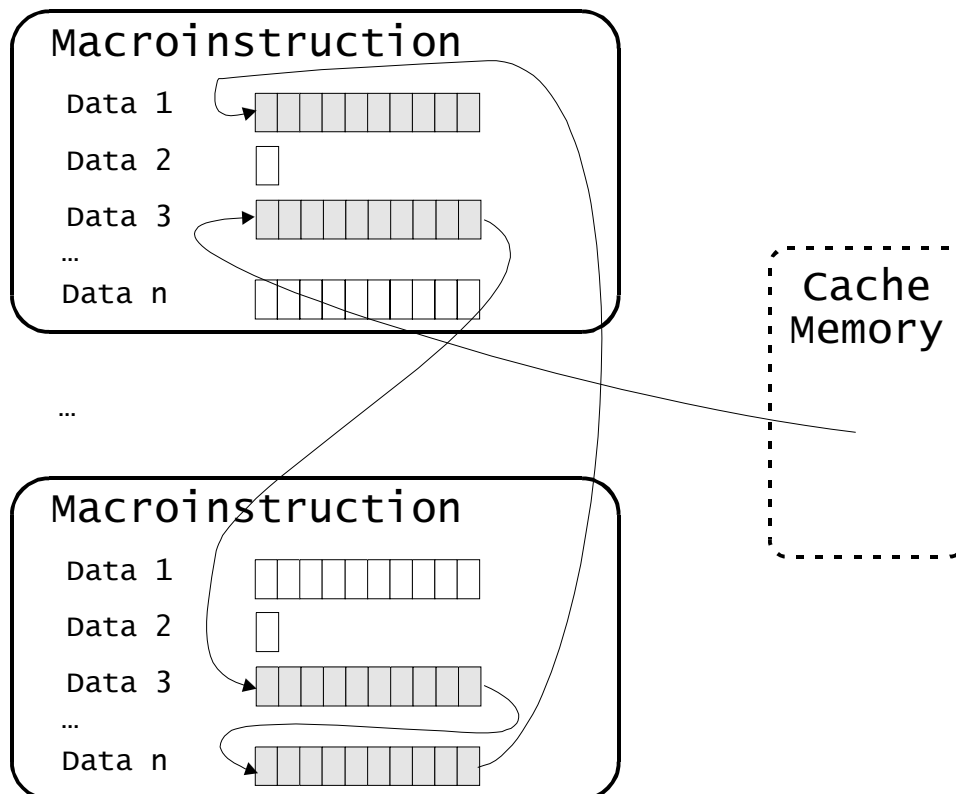


Figure 43 Structure of the cache memory simulator. Data blocks in grey (variables belonging to the static macroinstructions) are present in the data cache and linked to each other through a list.

Instead of having an allocated model for the cache memory itself, where memory pages are stored and where it is necessary to check for the searched variable to be present or not, according to its address, the cache is implemented as a delocalized entity, composed of a list of data blocks that belongs to the static macroinstructions that own data. In this way, each time a macroinstruction is executed, it is immediately known if the requested data are stored in the cache or not (in practice, in the first case the pointer has an allocated space otherwise it is set to NULL). The same macroinstruction is charged of updating the *use* index of the data, in order to keep track of the most recently used locations. When it is necessary to select a victim to load new data in the cache, i.e. when a cache miss

takes place, only in this case the list is scanned to find the data in memory to be deleted, according to the chosen policy. In the case of cache hit instead, the only additional operation required by the tracer is the writing of a new *use* index in the data block record, without any call to an external cache evaluation procedure.

Once the cache tracer is implemented, since it is automatically linked to the execution of the program, no further action is required to obtain a complete report of the time-dependent evolution of the application. In particular, at the selected time granularity (control-rate or seconds) it is possible to obtain the following information:

- number of executed calls for every element of the complexity vector
- number of really processed data for every element of the vector
- number of operations executed by blocks or by samples respectively
- number of operations executed at the k-rate or at the a-rate respectively
- total data transfer to and from the cache
- detailed data transfer to and from the cache according to the specific function (instrument instance or opcode)
- bus traffic

As previously mentioned, the cache tracer system has been built with some degrees of approximation to keep it the more general and the more easily adaptable possible to a specific architecture. The most relevant approximation introduced is the abstraction of the concept of cache page. Instead of considering a paging mechanism, it is assumed that if a variable is not present in the cache the entire space for it is loaded (e.g. 4 bytes for a scalar, 400 bytes for a 100-value block, and so on), and the same it is completely removed if the oldest in memory. Only tables and delay lines, for obvious size reasons, have also two indexes associated to them that keep track of the portion present in cache.

In this way the cache simulator has a relatively coarse granularity, and it is to be expected that it will not work well on programs with many small data blocks and feedback loops, while on the contrary it should have acceptable error margins on block-based direct data flows. Since most of the simulated examples have been verified as belonging to this second group, especially for multimedia processing, this coarse granularity has been foreseen as acceptable. The same, it was already verified that in most cases a time granularity of 1 second is enough to track the most meaningful aspects of the complexity of an application.

3. Experimental Results

3.1. General Remarks

To measure the goodness of a simulator is necessary to compare its estimated execution, i.e. processing time and relating memory traffic, against the actual execution measured on the target platform. This measure can give an idea of how well the simulator will eventually behave when the architecture does not exist and only its high-level architectural features are known.

3.1.1. *Simulated Platforms*

Experimental measures for the proposed architecture simulator will be reported for two different platforms. The first one is a PC platform based on an Intel Pentium processor and running the Windows NT operating system; results for this platform will be present exhaustively and will represent the main validation of the virtual approach described in this dissertation.

The second platform for which some results will be provided is the TriMedia processor described in Chapter 6; due to the difficulty to obtain meaningful measures on this processor, results will be rough and mostly qualitative, in the sense that it was not possible to compare the results of the simulator with measured performances other than the real-time audio listening and a rough indicator of the CPU load provided by the Mykerinos board API.

3.1.2. *Mapping of the Complexity Vector*

In contrast with the results normally reported in literature it was considered *fundamental* to track results on an architecture completely different from the one where the simulator was developed on. In fact, purpose of the simulator is to emulate a platform, even a non-existing one, and to predict its behaviour when a specific application is run on it. Only in this way a simulator can be really useful for design purposes, and not only for software optimization.

Since the proposed tool was designed in platform-independent manner, it leaves open all the specific features of an actual platform and a virtual complexity vector is instead computed. Practically this means that, to be able to estimate a real platform it is necessary to know, in some way, how to map the virtual parameters of the complexity vector into actual parameters for that same platform. This can be done in three main ways, which, in order of potential precision, are:

- measure the actual parameters of the complexity vector; the more precisely are these parameters estimated, the more effective is expected to be the fidelity of the simulations, given the fact that it is necessary to verify that the introduced approximations are not too important to make the estimation unusable. The conformance of MPEG-4 SA is based on this assumption and its consistency will be verified in the next sections; the results reported will be based on direct measures of the complexity parameters, not necessarily made by the SAINT-based simulator.
- exploit some existing benchmarks for the relevant operations, which are normally available for the most common DSP platforms; the problem can be that some parameters can be incorrectly estimated or not available at all, and this introduces a further degree of approximation that could heavily affect the precision of the simulation.
- estimate the execution time and memory requirements of the relevant complexity parameters, based on theoretical criteria deriving from the design of some architectural blocks or instruction set, or on existing architectural blocks available under form of HDL code or simulated results. This third option is not necessarily inferior in precision to the second one.

In any case, results provided for the first option will show that, once the complexity parameters (in this case, the macroinstructions) are known with a good degree of confidence, the precision of the simulation is good enough to evaluate if a platform can be considered suitable for the target application.

3.1.3. The Scheduler and Block-by-Block Execution

The proposed architecture simulator relies on the SA scheduler and on the b-b-b execution scheme. Clarifications are necessary about these two important issues.

The presence of the scheduler is not directly profiled. The algorithm it executes is fixed and well described in [71]. It has been assumed that the impact of the scheduler is not relevant on the execution time, and this assumption has been confirmed by results reported below. Moreover, the scheduler is not part of the simulated virtual DSP; in the SAINT framework it is a side unit that coordinates the program execution and then does not interfere with processing except that by communication through synchronized memory sharing (global variables). As remarked in Chapter 5, the design of an application in SAOL implies the use of its normative scheduler; a designer must consider it as an existing hardware or software block and according to the overall program he may decide to leave it out (in a purely processing program, for instance, it just has to update global variables and release the control to the processor again) or estimate its impact simply as a macroinstruction at the control rate.

Block-by-block execution instead does not affect the generality of the simulator. Even if the SAINT simulator operates possibly on vectors, the number of executed operations is profiled by samples and corresponds to the SAOL number of operations. The virtual model is independent from the tool actually implementing it, and the mapping of the complexity vector does not depend then on the specific SAINT structure. The designer has only to correctly introduce the expected or measured value for a mean operation of that class. Apart from the already discussed uprate problem of practical efficiency, the same results could have been obtained simulating SAINT through an e.g. saolc-based simulator.

3.2. Results for a PC Platform

The general purpose PC platform is the one where research and development were carried on, and then it is obvious that the most consistent block of experimental results is available for this platform; in particular, even if software was continuously compiled and tested on both Windows- and Unix-based machines to verify the correctness of the achieved results, the selected system for the last group of simulations has been again the one running the Windows NT operating system. The previous choice was confirmed because of the great friendliness in development tools for PC and of the availability of different compilers and debuggers that can provide a more robust refinement of the software.

However it is undeniable that measures possible with the NT operating system are less precise than those made under a Unix operating system, for two main reasons: first of all precision with NT is allowed only on a millisecond scale [131], while all Unix environments provide system calls running at the microsecond precision;

secondly, the NT system is known to have a coarser process management, and the potential overhead that may pollute measures is more relevant.

In chapter 5 results of simulations and comparisons with other SA decoders were presented for a relatively fast Pentium II -based PC. In this section results will be presented for a slightly older version of the same processor, i.e. the Pentium MMX processor running at 200 MHz. This choice has been made for several reasons: first of all, this is the machine where most of the decoder and associated simulator have been developed from the very beginning, and if it is easy to just move the final executable on a different processor it is more annoying to move a complete established development environment in a few time and with affordable results.

A second reason, a more technical one, is that a Pentium running at 200 MHz gives results that are more comparable to those of the available version of the TriMedia, running at 100 MHz on the Mykerinos board and on the Philips EVM board. Finally, it has been considered that when careful measures are required, a slower processor better matches the limits of the NT operating system, and the same measures can result more accurate since unpredictable overheads are better masked by longer processing times.

3.2.1. Measure of the Complexity Vector

Because of the millisecond time granularity of the Windows NT operating system, it is not possible to measure the execution time of an element of the SAINT instruction set (i.e. an element of the particular complexity vector that is the SAINT instruction set) directly during the execution of a program⁸. Instead it is necessary to modify the source code to make it execute every macroinstruction several times, or to write an ad hoc program for those measures, and extrapolate afterwards the mean value of the single execution. In the specific case, since the modification has anyway to be introduced and the measure is done once forever, it has been decided to loop each operation 10^5 times, in order to obtain a measure in the order of magnitude of seconds to the tenths of seconds and to avoid in this way the potential overhead coming from system calls for clock recovering.

A measure made in this way additionally permits to achieve another remarkable result: this measure, since a single operation at a time is repeated many thousands of times on the same data, is practically "cache-independent". This means that, if any stall in the execution of that small portion of code may result from the necessity to load code itself in the instruction cache and relating data in the data cache, this is essentially done at the first iteration. As a result, once the mean is calculated, the cache latency is divided by 10^5 and then practically eliminated. This is particularly important for instructions executing faster. Moreover this kind of measure is more consistent with benchmarks reported in data sheets or simulated from instruction sets of architectures under design.

⁸ Note that the same would be true even on a Unix-based platform for simple instructions that execute in a few nanoseconds and for which a macroinstruction of e.g. 100 samples executes in a microsecond or a few microseconds.

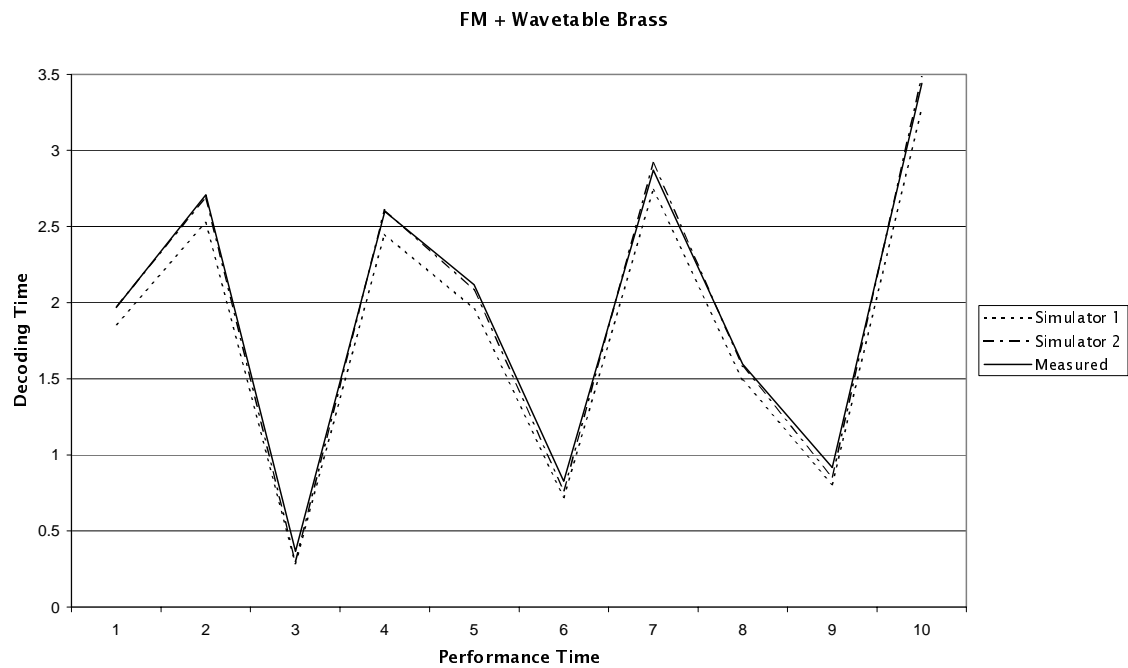


Figure 44 Estimated and measured decoding time for the Brass sequence with a block size of 50. Dotted line represents the simulator without cache tracing; dash-dotted line represents the complete simulator. Times are in seconds.

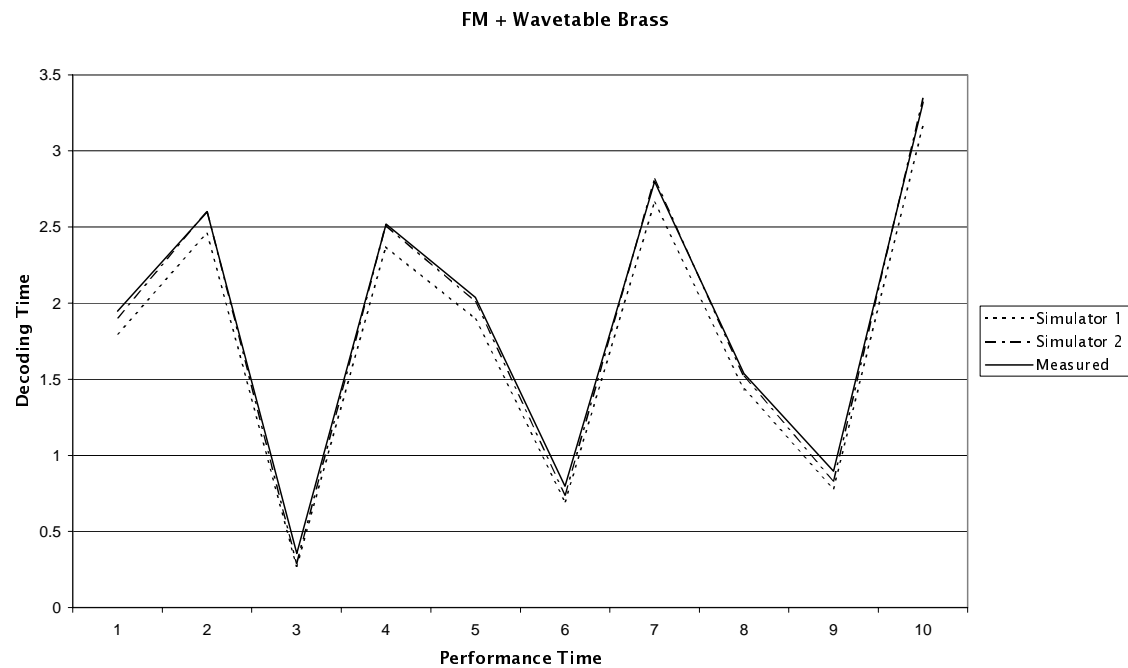


Figure 45 Estimated and measured decoding time for the Brass sequence with a block size of 100. Dotted line represents the simulator without cache tracing; dash-dotted represents line the complete simulator. Times are in seconds.

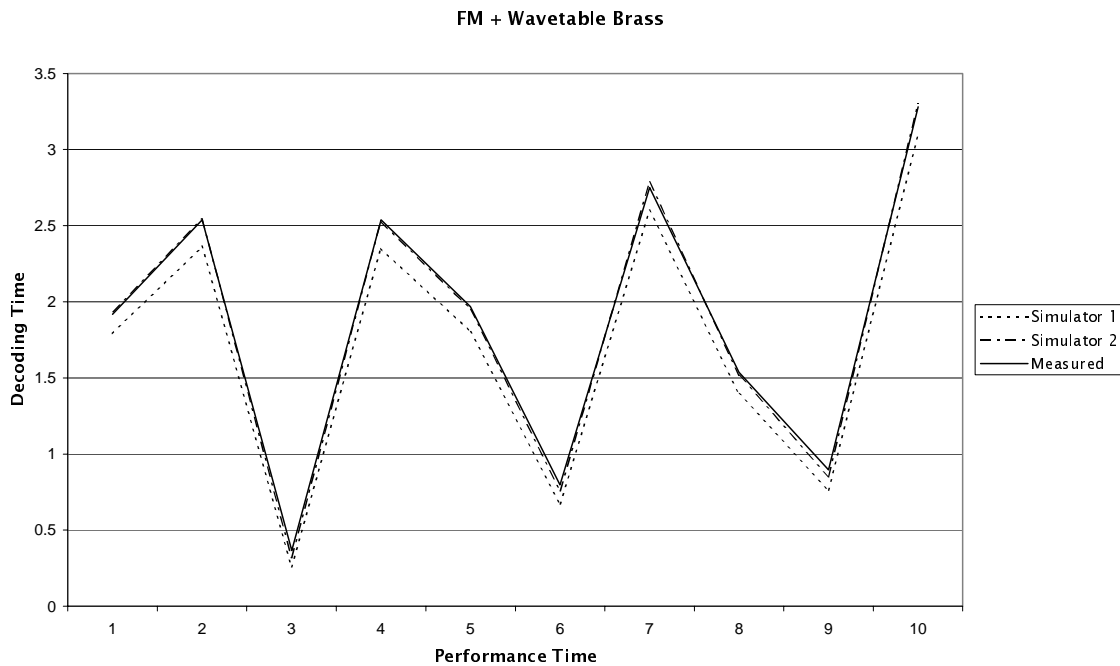


Figure 46 Estimated and measured decoding time for the Brass sequence with a block size of 441. Dotted line represents the simulator without cache tracing; dash-dotted line represents the complete simulator. Times are in seconds.

3.2.2. Estimation of the Interpretation Overhead and of the Scheduler

The estimations of the SAINT-based simulator will be compared with the measured performance of the SAINT decoder. Being the target platform an interpreter, the execution overhead could be relevant and then it needs to be estimated.

Criteria for the MPEG-4 SA Conformance make abstraction of this overhead, since it is only a function of the specific implementation; they assume the programmer to be able to evaluate the cost of its own solution and to be able to add to the estimated complexity vector $C_v(t_i)$ of the program the expected overhead that comes from those parts of the execution process that are independent from the real data processing. In the same way the simulator based on SAINT, which is an extension of the MPEG Conformance criteria, is able to make abstraction of any specific platform solution, but it provides hooks to permit the estimation of the different kinds of overhead that may occur in practice, and in particular it can trace the number of executed macroinstructions (or instructions, if no blocks are used), the number of scheduler cycles and the number of instance swaps that are executed by the scheduler throughout a complete simulated program. In the case of solutions not based on an interpreter the first kind of overhead is normally not present, while the second and the third, as said, are unavoidable in simulations and design based on the SAOL language; at this regard, it can be remarked once again that in any real-time and interactive program a cyclic control subprogram is present and unavoidable; in this sense SA corresponds to the programmer's needs and the overhead introduced in this way is in any case a "necessary" one and not a lack of

generality in the proposed simulation environment; on the contrary, a normative scheduler structure contributes to the coherent design of the control task.

Being the overhead quite small in an optimized decoder like SAINT, a direct measure of the three overhead kinds is not easy. In this case, the following solution has been adopted: assuming the three overheads as three variables, thing that implies the approximation of considering all the macroinstructions and instance swaps as executed exactly in the same way, three equations are necessary to calculate the three required times. Three different test sequences characterized by different ratios between overhead and real processing have been executed at two very different block sizes, for instance 50 and 441, in order to obtain a measurable difference in execution times and a remarkable difference in the number of scheduler cycles and macroinstruction executions. In this way it is possible to solve the following system:

$$\begin{cases} \Delta t_1 = \Delta M_1 * t_M + \Delta S_1 * t_S + \Delta C_1 * t_C \\ \Delta t_2 = \Delta M_2 * t_M + \Delta S_2 * t_S + \Delta C_2 * t_C \\ \Delta t_3 = \Delta M_3 * t_M + \Delta S_3 * t_S + \Delta C_3 * t_C \end{cases} \quad (1)$$

where t_M , t_S and t_C are unknown times for macroinstruction overhead, instance swaps and scheduler cycles respectively, and all the deltas are the overall differences in occurrences measured from the three sequences as explained above. Solution of system (1) in the considered case gave the solution:

$$t_M = 0.88 \mu s, t_S = 11.40 \mu s, t_C = 8.50 \mu s \quad (2)$$

In this way, it has been verified, and in fact it was already arguable from the draft comparison with *sfront* at the end of Chapter 5, that the macroinstruction overhead is of some importance in approximated simulations only in the case of sample-by-sample executions; the cost of typical scheduler operations is in the order of 1 to 3 milliseconds over a second of real processing on the Pentium MMX processor. In the next subsections results will demonstrate how these values are in fact of a low importance.

To conclude, it has to be noted that these estimations are made on real measured differences in normal conditions, and then the extrapolated times already include in some way the time lost in cache latency as a part of the estimated overhead.

3.2.3. Results without Cache Simulation

A first group of simulations has been run without introducing the macro-oriented cache tracer, in order to have a first order estimation of the behavior of the simulator and to understand how much the cache management can potentially influence the overall execution time of a program. Experimental results are reported in Figures 44 to 46 for the Brass sequence (FM + Wavetable synthesis) written in SAOL/SASL for three different block sizes, precisely 50, 100 and 441. A second group of results are reported in Figures 47 to 50 for other test sequences, all of them in SAOL/SASL with a block size of 100. In all these figures the horizontal axis

represents the internal scheduler time, i.e. the performance time, while the vertical axis represents the measured or estimated decoding time on the virtual platform. The solid line in Figures 44 to 50 is the measured decoding time of the program with a time granularity of one performance second. Measures were conducted using SAINT decoder with no other process running on the machine except the operating system; moreover, every known access to the system that could false the measure was disabled in the code: this means that, during measures, writing to files on the hard disk or to stderr/stdout was commented, with the unavoidable exception of functions returning the system clock. This gave the cleanest measure possible. The dashed line in the Figures is the estimated decoding time of the program with a time granularity of one performance second. Cache simulation is disabled and execution overhead has been considered for both macroinstructions and the scheduler, as explained above. During simulation, classes of macroinstructions were incremented and reset each scheduler second and the partial counters multiplied times the measured time for the same class of macroinstructions. A complexity vector of 20 elements has been used (see Appendix 2). Times for eventual scalar operations has been calculated from the measure made on a block of samples, in order to save simplicity and coherency of the simulations; this corresponds to the fact that a single benchmark is normally used or available, for a single operation or for a block of operations. This further introduces a degree of approximation. The dash-dotted line in Figures 44 to 50 is the simulated decoding enabling the cache simulator and will be commented in the next subsection.

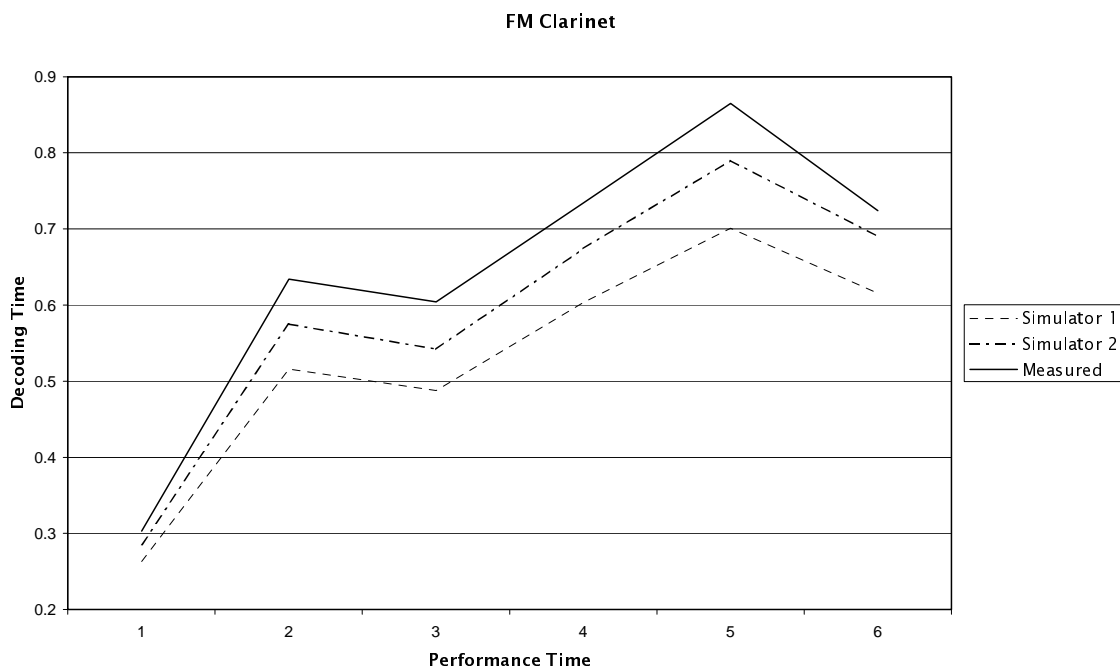


Figure 47 Estimated and measured decoding time for the Clarinet sequence with a block size of 100. Dotted line represents the simulator without cache tracing; dash-dotted line represents the complete simulator. Times are in seconds.

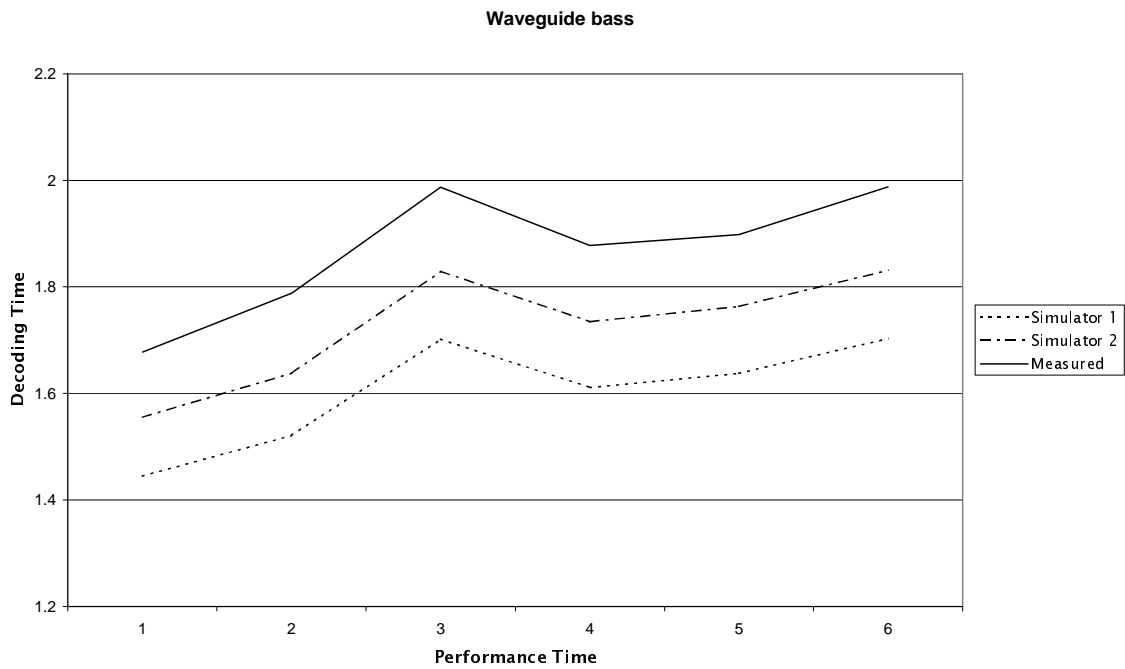


Figure 48 Estimated and measured decoding time for the Bass sequence with a block size of 100. Dotted line represents the simulator without cache tracing; dash-dotted line represents the complete simulator. Times are in seconds.

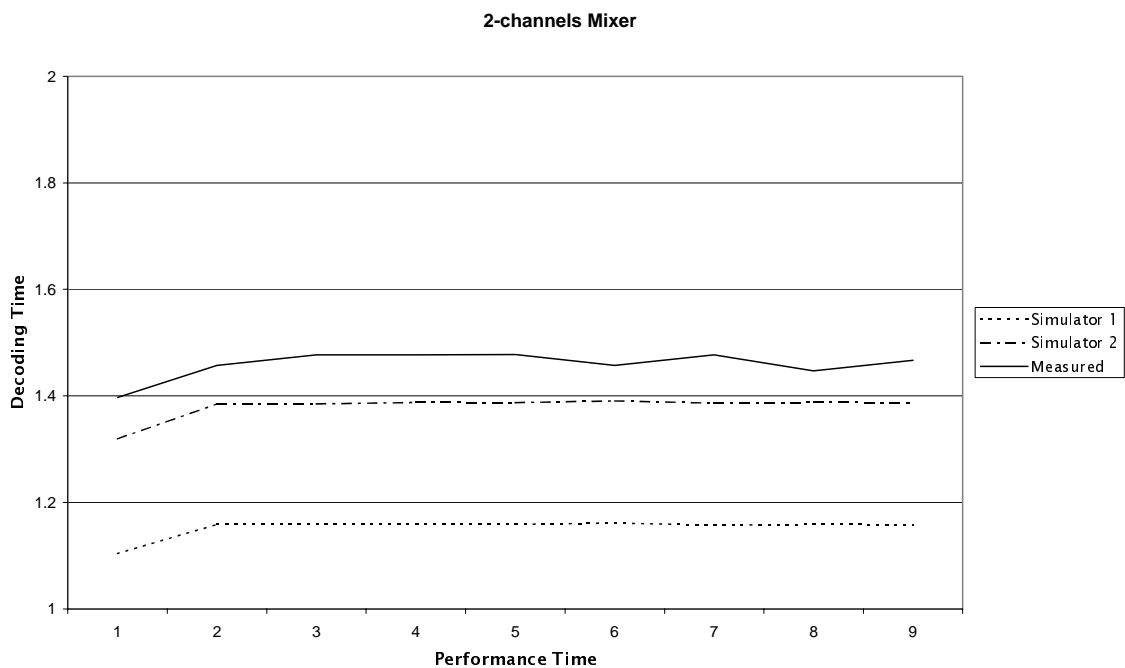


Figure 49 Estimated and measured decoding time for the 2-channels Mixer sequence with a block size of 100. Dotted line represents the simulator without cache tracing; dash-dotted line represents the complete simulator. Times are in seconds.

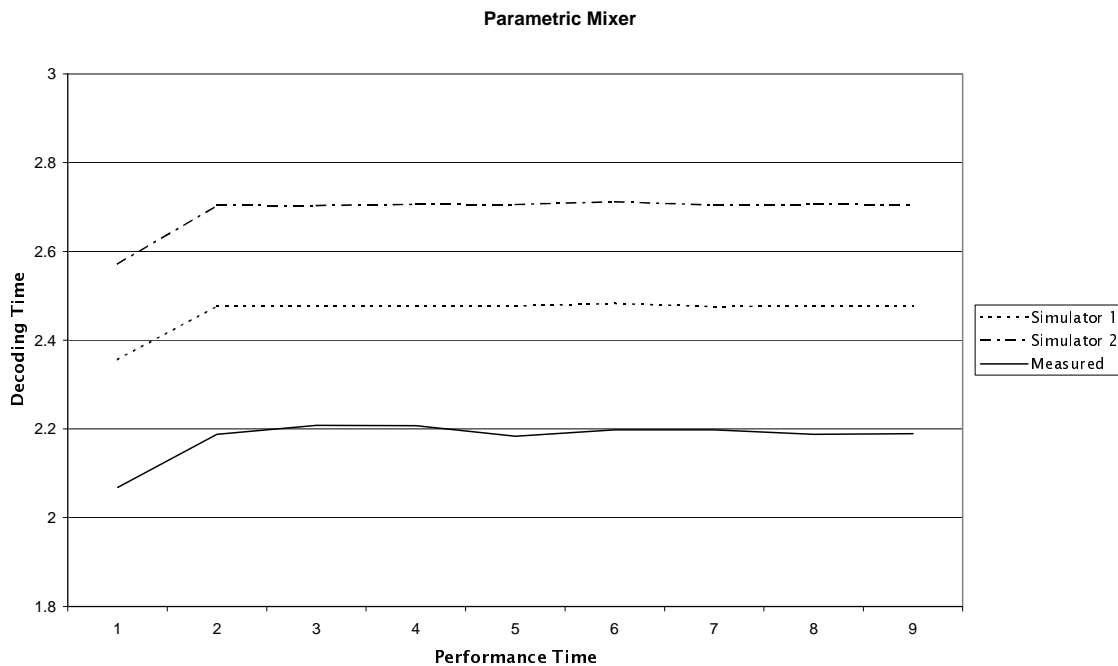


Figure 50 Estimated and measured decoding time for the Parametric Mixer sequence with a block size of 100. Dotted line represents the simulator without cache tracing; dash-dotted line represents the complete simulator. Times are in seconds.

The first comment on the results is about block size. Figures 43 to 45 clearly show that no appreciable difference can be noticed in the simulation fidelity at different meaningful block sizes. For this reason results for other sequences in Figures 46 to 49 have been reported only for a block size of 100.

The Brass sequence is the one giving the best simulation results among all the tested ones; the precision of the estimated time is sometimes even surprising since it sometimes reaches the 95% of the measured time (percentages are plotted in Figure 51). This is somehow justified by the fact that Brass is a sequence containing heavy processing executed in blocks and moreover locality of the program data is remarkable (many instances of the same, short instrument algorithm), a fact that does not make cache memory so influent during execution. The total overhead in this sequence is nearly invaluable in percentage, as it is possible to argue from the comparison with *sfront* of Chapter 5 (SAINT is even slightly faster).

In a "lighter" sequence like the FM Clarinet (Figure 47) the simulated time has an expected, more relevant margin of error over the measured one. The speed of the decoder is enough to guarantee real-time. In this case the approximation of the simulation method is quite evident: grouped classes of operations, extrapolated overhead times in common for all kinds of operations and, moreover, not considered memory allocation times introduce a valuable underestimation of the program execution. However, even without cache simulation, the margin of error is never above 15%; this must be considered an excellent result because it shows that the simplifications of the method did not degrade results too much for many simple but useful purposes like e.g. the definition of MPEG-4 SA conformance.

The same results are confirmed by the waveguide bass sequence in Figure 48, notwithstanding the presence of a considerable block of code executed by samples because of feedback. A heavier underestimation is expected here because, when a macroinstruction in SAINT is executed on a single sample, the real overhead time is often longer than the measured one because of the repeated initializations; most probably this is partially balanced by the fact that processing is heavy and largely predominant in the physical model of the bass algorithm, like in the Brass sequence. Percentage results for the Brass sequence are plotted in Figure 53.

The 2-channels mixer sequence reported in Figure 49 is characterized by a rather low locality since it has many active instances belonging to different instrument algorithms (oscillator, low-shelving, bell filter, high-shelving, top mixer) and then a high margin of error is expected without cache simulation. This is confirmed by the experimental results, even if the error remains inside a 22% band that can still be considered acceptable as for the previous cases.

To test a potential worst case, starting from the 2-channels mixer algorithm an ad-hoc program has been built where the mixing algorithm is written in parametric form using a *while* loop with many simple operations like multiplications, additions and memory accesses to be repeated for every channel in a sample-by-sample fashion, because of the *while* feedback. This sequence breaks the assumptions at several points, because of the s-b-s code, of the unique overhead for the instructions that in this case is largely overestimated (all operations are simple) and of the fact that in this way nearly the 50% of the execution time in SAINT comes from overhead.

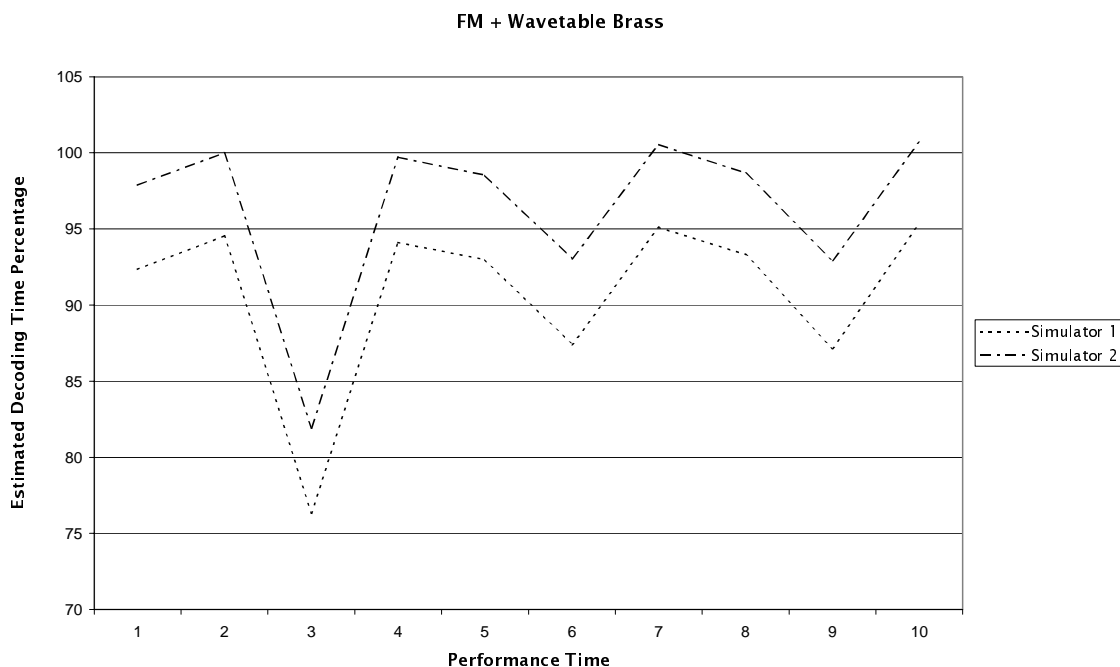


Figure 51 Percentage of the estimated decoding time over the actual measured time for the Brass sequence with a block length of 100. Dotted line represents the simulator without cache tracing; dash-dotted line represents the complete simulator.

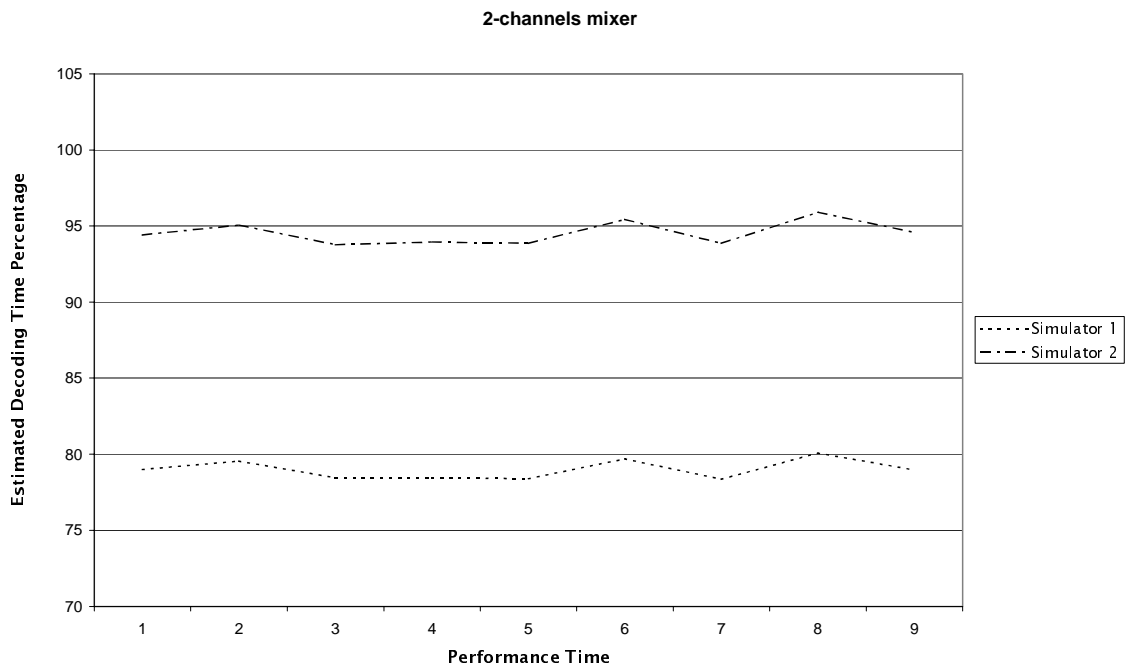


Figure 52 Percentage of the estimated decoding time over the actual measured time for the 2-channels mixer sequence with a block length of 100. Dotted line represents the simulator without cache tracing; dash-dotted line represents the complete simulator.

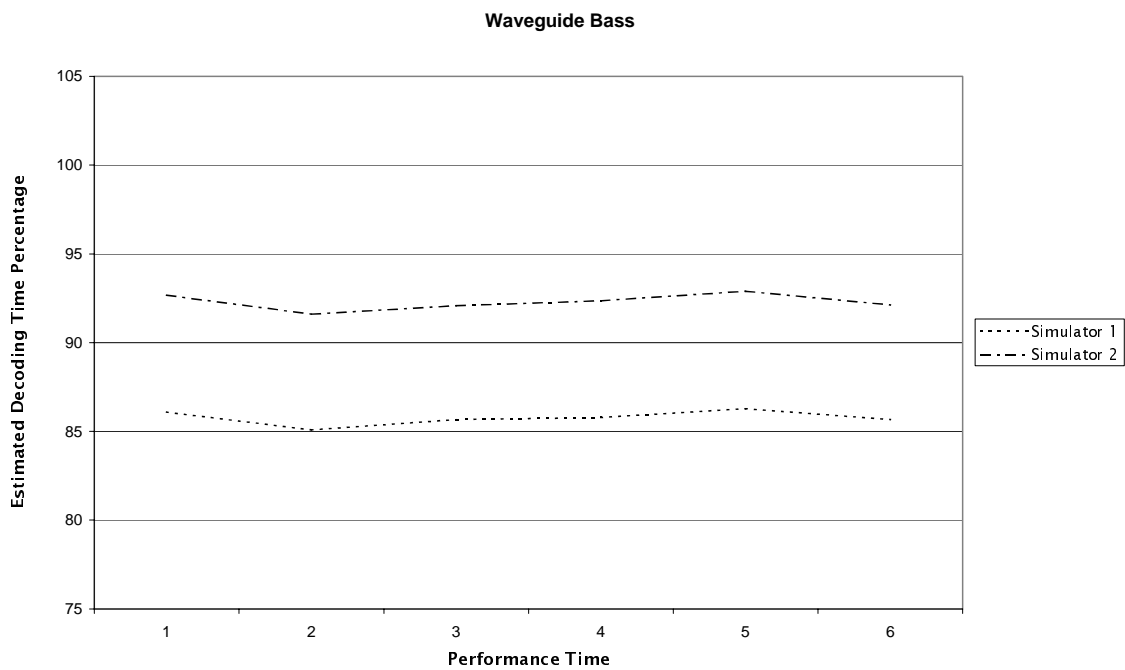


Figure 53 Percentage of the estimated decoding time over the actual measured time for the Bass sequence with a block length of 100. Dotted line represents the simulator without cache tracing; dash-dotted line represents the complete simulator.

In spite of all these challenges the simulated execution time does not surge too much above the measured time (more or less 12% overestimated in Figure 50). This definitely prove the robustness of the simplified proposed method for simulation. In addition, even if with a slightly higher degree of approximation (the complexity vector is shorter and then parameter estimations are rougher), this also prove that the proposed method for MPEG-4 SA Conformance test has a good degree of consistency; a programmer is indeed in a position to judge, starting from Levels and relating complexity figures (which are "virtual" sequences), if the SA decoder under development will be capable of supporting the necessary computation in real-time. It is true that this can be achieved with an error that can be around the 25%, but in any case this is a small fraction of the dynamic excursion of the decoding complexity (i.e. the potential error in the case of having no indication at all).

3.2.4. Results with Cache Simulation

In order to better simulate real architectures and to improve the margin of error presented in the previous subsection it is fundamental to introduce cache simulation to estimate traffic through the system memories.

In order to match coherently the approximation assumptions used for computation, as explained in section 2, cache simulation has been introduced in the simulator with macro-oriented criteria, i.e. essentially loading and discarding data according to processed blocks of samples, completely ignoring any physical address tracing that many other tools of this kind use. This method is supposed to be able to keep track of the basic data traffic to and from the cache memory, with the main, limited, purpose to estimate the impact of cache on the execution time and to track as well as possible the locality of the programs under consideration.

The simulation of cache memories in modern processors like the Intel Pentium, or the TriMedia in the following section, presents some challenges. First of all, more than one cache memory is present; in the case of the Pentium MMX processor, 16 kB of data cache and 16 kB of instruction cache are available on-chip, and moreover and additional 512 kB of Level 2 cache are provided by most common motherboards. A first draft estimation was then made on the overall program size; this estimation shows that in a majority of typical cases, if extremely long delay lines or improbable block sizes are not required⁹, code and data for tested programs were completely contained in the L2 cache memory; it was then decided to focus the simulation on the Level 1 data cache, being the instruction cache essentially considered in the extrapolated overhead times in (2).

Secondly, and even more important, it is not possible to estimate precisely the impact of a cache miss on the executed program unless a byte-exact simulation of the whole system is done. To give a better idea, even the reference manual of the MS Visual C++ compiler [132] does not give a better indication than saying that a cache miss cannot be exactly profiled and could cost something between 10 and 20

⁹ As a side note, it is interesting to report that L2 cache simulation was roughly able to preview at which block size inversion in the decoding time curve occurred for the Brass and 2-channel mixer sequences.

processor clock cycles for L1 caches and between 20 and 40 processor clock cycles for L2 cache. It is obvious at this point that the approximated criteria used for the proposed simulator are more than adequate to the indications provided by much more complex and well known compilers and profilers. In the results presented in this section the best case of 10 cycles per cache miss has been considered.

Experimental results are presented again in Figures from 44 to 50, and 51 to 53 for percentages, by the dash-dotted line labeled "Simulator 2".

Simulations for the Brass sequence astonishingly report estimated times near to the 100% of the measured time when processing is sustained; anyway this must honestly be considered as a particularly favorable case.

Apart from results for the overall simulation on Brass, a comparative analysis of Figure 45, Figure 48 and Figure 49 (or similarly Figure 51, Figure 52 and Figure 53 for percentages) shows two good points and a less good, expected one.

The first good point is that the approximated cache simulator provides a substantially correct view of the locality of the program. The difference between the Brass program and the 2-channels mixer program is well-detected; in the first case an increase in the execution time of a few percentage points is predicted, while a more considerable 15% increase is estimated in the second case, bringing the complete simulation to reach a remarkable 95% of the measured time. The Bass (and Clarinet) programs present an intermediate locality, which is again sufficiently well detected even if only the half or a little more of the underestimation is filled.

The second good point is that, at least in the tested common cases where cache misses represent a fraction of the complete execution time (it is not excluded that the contrary could happen in particularly memory-dependent programs), the order of magnitude of the slowdown due to memory accesses is well estimated.

The bad point is that in some cases the cache simulation is not capable to completely justify the simulation underestimations, or, on the contrary, in some other cases like the parametric mixer example, things are getting even worse than before (as expected in case of overestimation).

In any case, it has to be concluded that the results provided by the virtual simulator are good ones and mostly usable for consistent estimations of the SAINT decoding time on the Pentium MMX processor.

3.3. Results for the TriMedia Platform

It is very important that a tool conceived for simulation and design of architectures is capable of achieving good results for platforms other than the one on which it is developed, otherwise the usefulness of the tool would be hardly limited and practically reduced to a software optimization tool. To further verify the tool, some programs were then run on the simulator, still working on the PC, to try to preview the behavior of SAINT on completely different physical architectures. The TriMedia processor, mounted on the Mykerinos board used in ThreeDSPACE, represented the ideal device for such an experiment.

Being the TriMedia not a general-purpose processor, but rather a multimedia oriented DSP for real-time applications, the development utilities made available by the system are not so flexible as those for PCs or workstation, but this indeed

represents a practical problem that in many common cases must be faced, above all in the DSP field.

In particular, the major issues that were encountered and that do not permit to have detailed results as for the PC case are the following:

- the SAINT decoder used to measure actual decoding time is spread between the PC and the TriMedia, which runs the execution engine. It is not possible then to correctly measure decoding time unless the sequence can be run in real-time (thing that in the end corresponds to the only useful case)
- the SAINT decoder, for practical reasons of implementation complexity and time, is run as a plug-in effect of the Mykerinos API, which considerably interacts with the decoder itself making almost impossible the separation of real decoding time from parasitic offset
- for the same reason of the Mykerinos API, no block length other than 64 can be consistently simulated, since the interface with the real-time TriMedia AudioOut unit at the moment only works effectively with this value. This constraint is a hard one to the generality of SA and its removal is under study at the moment.
- as measuring tool, only a displayed figure indicating the CPU load is available with the Mykerinos SDK, and this makes measure of the actual decoding time nearer to a guess than to a true estimation
- if it is hard to consistently simulate the influence of the cache memory for a modern processor like the Pentium, it is even harder to do that for a processor like the TriMedia, a VLIW unit where a cache miss could break more than a pipeline flow and for which the exact behavior in complex situations is hard to predict. No concrete hints to cache simulation or cache slowdowns were found in literature for such a processor.

On the other hand, these issues make the simulation problem an actual problem. In order to proceed to simulations the following decisions have been taken:

- the TriMedia SDK provides time measures with a resolution of a microsecond: in any case, due to the speed of some optimized vectorial macroinstructions (see Chapter 6), it has been necessary to measure them with a looping technique like that used for PC: this also avoid the data cache overhead as explained in section 3.1. It has been verified that some of the macroinstructions are much faster than the corresponding ones running on a Pentium with a double clock speed, thing that confirms the effectiveness of the work done for the TriMedia (see Appendix B).
- being more a guess than an estimation to extrapolate overhead like done with system (1), it has been decided to calculate it from the PC values instead; since the clock speed of the tested TriMedia is the half and that no practical parallelization is expected on the scheduler linear tasks, PC values in (2) has been doubled. Rough direct measures during decoding did not confute this assumption
- for cache simulation a similar assumption has been made: a cache miss in the 16 kB data cache has a cost that is proportional to that of the Pentium processor; this assumption comes from the remark that most

macroinstructions has only one "streaming" input, while all other operations are performed on static "local" parameters

- given the simulation environment and its purposes, a sequence run in real-time can be evaluated against the displayed CPU load, while a sequence exceeding real-time capabilities cannot be evaluated, since the Mykerinos API engine reduces load artificially to approach real-time.

Results for the Clarinet sequence are reported in Figure 54.

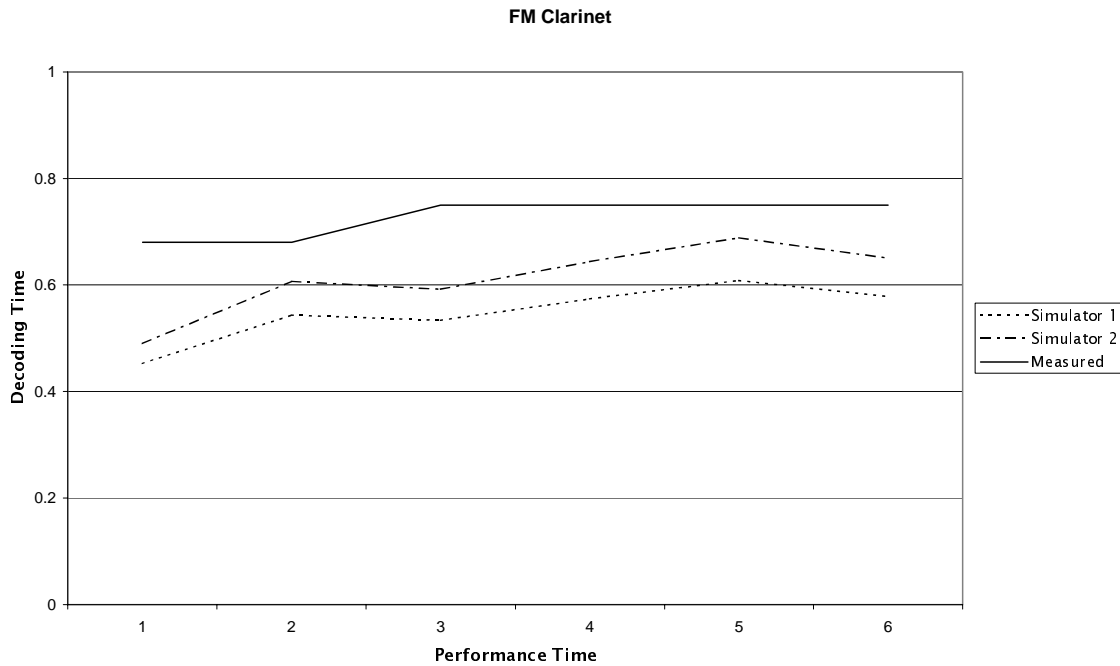


Figure 54 Estimated and measured decoding time for the Clarinet sequence on the TriMedia processor with a block size of 64. Dotted line represents the simulator without cache tracing; dash-dotted line represents the complete simulator. Times are in seconds.

Even if the measured figures must be considered with some caution, it can be considered that the estimated values are good ones. The Simulator 2 line including the cache simulator underestimates the CPU load by 10-15% in the mean; this value increases to 20-25% if the contribution due to overhead of the API (an offset) is removed, but this should be evaluated against a more precise study of the API behavior and it cannot be done at the moment.

The 2-channels mixer is estimated to run out of real-time capabilities of the system of some small percentage; but the 130% value CPU load value cannot be considered a valid reference for what said above. The Brass and Bass sequences are too much complex for a 100MHz TriMedia alone, and indeed a near-to-white noise is the only output of the SAINT decoder.

In general, results for the TriMedia can be considered satisfactory. It is clear that much more knowledge of the architecture is necessary to better tune the simulator to truly parallel devices, but at least the hard division between real- and non real-

time sequences is estimated with a good degree of approximation and can constitute a valid indication for the programmer and the developer.

4. Qualitative Comparison with Other Simulators

In sections 3 and 4 of Chapter 2 the most representative instruction set and architecture simulators reviewed in literature were introduced. It is now time to proceed to a brief comparison with them, in order to evaluate the obtained results. The simulator presented in this dissertation has an original conception concerning the *approach* to simulation and some of the assumptions behind it are also innovative; moreover, the more similar tools in literature are based on quite different language approaches: it follows that a quantitative comparison with them can roughly be done in terms of speed of simulation and approximation of the results, but a direct comparison on the same test sequences and programs is not possible. This is a consequence of the choice of SAOL as a starting point, which yield a tool more suitable for real-time multimedia-oriented applications at the cost of a somewhat diminished generality of the supported programs (only algorithms that can be efficiently implemented in SAOL can be tested).

First of all, it is important to remind that the proposed simulator differs from many other simulation tools in the fact that it is an approximated simulator and not a precise one. Some tools like SHADE offer the possibility to choose a certain degree of approximation to speed up performance time, but this approximation is done reducing stochastically the number of evaluated instructions, while the remaining are simulated exactly. In the proposed case, approximation comes instead from the higher level of abstraction, which hides machine details in change of a better generality of the tool.

This approach to approximation is rather innovative, being Mermaid and Zhu/Gajski the only two other methods based on similar approaches. A comparison with them is consequently more interesting.

4.1. Comparison with Similar Simulators

4.1.1. The SAINT Simulator and Mermaid

Mermaid, briefly introduced in section 3.2.2 of Chapter 2, is inspired by simulation criteria similar to those presented in this dissertation; the work presented here was inspired more by MPEG-related complexity analyses described in [130] for natural video, also because Mermaid appeared in literature well after the beginning of the discussion of the implementation issues inside MPEG. It is undeniable however that Mermaid augments a c-written program in order to simulate a kind of virtual "c-machine" in the same manner of what have been done with SAOL.

The main difference between the two methods lies in the fact that Mermaid implements a simulator for the complete set of C operators, while the SAINT simulator is based on the concept of virtual class of operations; the first approach can guarantee a better precision in simulation (5% in the mean, 30% in the worst case), while the second permits to better decompose, in a simple way, more complex tasks based on language libraries in a platform-independent manner.

Consider, in fact, that a tool that only deals with language operators (and nothing is said about Mermaid and compiler-dependent C libraries) is strongly limited in generality, since not only it requires a program written in C, but most probably any practical application needs, should it be possible, a custom low-level translation of any standard function call.

Moreover, if a one-to-one map is used between functions and virtual instructions, this can result in such a heavy set, above all in the case of C, that any practical benchmarking of actual platforms could become a really hard task. The concept of virtual instruction classes used for SAINT, if it introduces some more approximation, it also contributes to keeping the simulation time limited, never above ten times the execution time, while slowdowns in the order 60-to-700 are reported for Mermaid.

The other context-related advantage of the SAINT simulator over Mermaid, is the choice of SA as a supporting language; if this limit in some way the range of applicability, it allows an analysis of any application in function of the performance time; this is an invaluable benefit for real-time multimedia applications, since in this field only a time-dependent analysis can be considered meaningful.

On the other side, Mermaid is less dependent on a well-done benchmarking of the operations, since simple instructions are less variant than virtual classes and more related to the low-level hardware device.

4.1.2. The SAINT Simulator and the Zhu/Gajski Simulator

The instruction set simulator recently proposed by Zhu/Gajski, described in section 3.2.3 of Chapter 2, makes a step further in the direction of the virtualization of the simulation: the compiler becomes retargetable, and any language can be theoretically cross-compiled to the virtual instruction set. In this tool the concept of virtual host is finally introduced, while in Mermaid a precise definition of the concept of virtual analysis was still not present. Moreover this new simulator is based on a static cross-compiler, and this limits the slowdown to the double of the execution time in the mean (but memory simulation is not included: in this situation the SAINT simulator can be considered as fast as this one). Indeed, in this case a precise relationship between simulation and hardware design is evident.

In comparison with the SAINT simulator, what is still not clear in the Zhu/Gajski simulator is how to deal with the language libraries, and then once more with the problem of algorithms eventually developed and optimized elsewhere. Moreover, the simulator still seems to be a general-purpose one based on a generic virtual instruction set, thing that still makes a simulation in function of time much more complicate.

4.2. Possibility of Application to Other Programming Languages

The proposed virtual model for platform-independent complexity measurements and simulation is based on a programming language; this permits its flexibility and portability, but at the same time it constitutes its limit, in the sense that source code of the application is necessary to run the simulation and to track the *time* of the program. Indeed this limit is such in the context of software simulation, but if

the target is also hardware design or system hardware/software codesign, the code of the application is a requirement and a starting point.

Theoretically, the described method of abstraction can be applied through any other standard imperative programming language. Some remarks on the matter have been already presented in the third section of Chapter 4. As explained there, the simulation in function of the performance time is something that can not be done easily, and above all it is not possible through some mean of automatic code instrumentation, like done in [39] or [130]. In fact, the variable describing time must be searched and manipulated in order to make it tracked during execution of the program. Moreover, the simulation of a program in a high level language makes necessary to deal with library functions and to decompose them in a set of meaningful classes of operations, and this also can require a certain amount of "intelligent", handcrafted tuning. Finally, the manipulation of an application source code can often be tricky and special attention is needed.

The solution proposed by Zhu/Gajski is interesting in the sense that any programming language is cross-compiled to a fixed virtual machine bytecode, not to the one "suggested" by the language itself. In this case either any standard library is cross-compiled to a specific lower-level block of code, or the source code itself needs to be manipulated to eliminate standard library calls. In any case, consider that the elimination of library calls could hardly affect the optimization of an application, since an important, programmer-conceived level of abstraction that could favor the design of specific solutions is removed and replaced by a generic and flat "complexity vector".

From this point of view, the situation definitely improves when the language is compiled by default to a specific bytecode and the virtual machine source code is available; this is what have been exploited in the case of SAINT, and above all what is available in the case of a popular general purpose language like JAVA. In fact, consistent multimedia applications are necessarily implemented using some specific calls to standard functions that deal with time of the system; in this case it is then possible to imagine an easy scenario where the "time" variable can be automatically detected; moreover, instrumentation of the source code for complexity estimation is not necessary, since the execution engine and functions accessing the system time can be manipulated once forever, in order to transform the virtual machine into a simulator.

It is however to be expected that a tuning of the application source code for the simulator will be required, since real-time programs have normally critical behavior in presence of a slowdown (due, for instance, to the "modified" virtual machine); in these cases it is necessary to rework time checks and I/O procedures in order to allow the correct (as in normal speed) execution of the several instructions.

In both situations, compiled languages and interpreted virtual machines, the process of definition of the complexity vector is the critical point. When a device is available, the vector must be carefully biased to stay in between the high level source code and the actual instruction set of the hardware device, in order to be able to effectively map the classes of operations into precise benchmarks for the device; this is what have been done for the MPEG-4 SA conformance, where the

complexity vector corresponds to the general features of state-of-the-art processors and multimedia DSPs. If this has been done in the particular case of audio, it is not difficult to imagine the same done with a special bias to other fields, and actually other researchers in MPEG moved in a similar direction for video (see [130]).

When a device is to be designed, and then not existing yet, the definition of a complexity vector, i.e. a high level instruction set, is easier, but at the same time its definition must be iterated several times in order to decompose at best the application into a homogeneous and balanced set of operations.

5. Conceptual Extension to the Automatic Design of Architectures

Once an architecture simulator is implemented, and its features really makes it such and not a simple instruction-set simulator for a specified platform, it is evident that it can be exploited not only for software optimization, but also as a support for high-level system design. The results provided by the SAINT simulator can be exploited by the designer to improve critical parts of the architecture, to properly measure the dimension of internal memories and to determine required I/O bandwidth.

5.1. Automatic High-level Design

Apart from the "static" exploitation of a simulator for design purposes, it is also interesting to understand if and how the same tool can be extended to provide automatic design of a high-level multimedia system.

In practice, a utility for automatic design is a tool that repeatedly loops over possible known optimization strategies and find the best solution according to some design guidelines or constraints. As described in Chapter 2, hardware/software co-design and some other more specific approaches start from the bare application and refine from scratch the complete system design.

In the case of the SAINT simulator, or tools that may be based on e.g. a JAVA-like virtual machine, once again the interpreter structure can result very helpful in the eventuality of such an extension to automatic design, since it provides the possibility of a straightforward insertion of counters and design hooks and it permits to easily manipulate the execution of an interpreted process.

For instance, it is not difficult to imagine an extension of the actual tool to become a feedback-driven looping simulator able to automatically redefine some important parameters in order to tune them and generate a high-level architecture. This architecture will be of course based on some well-defined "building blocks" that must be considered as "units to be configured" by the simulator. The main parameters that are right now exposed to potential manipulation are the following:

- required processing power and ALU operations: the great flexibility of the complexity vector permits to make it variable and tunable automatically in order to make e.g. a certain block of code executable in a certain maximum amount of cycles, assigning to each parameter of the vector a fixed amount of required or expected clock cycles.

- cache memory size and associated throughput: the cache memory simulator, even if approximated, has demonstrated to be capable to estimate correctly the locality of the application and relating cache throughput, and then the data cache size can be tuned according to the simulation results.
- basic blocks (s-b-s, b-b-b) and main data flows: since the application code is divided in blocks characterized by straight flows of data (b-b-b code blocks) and blocks containing feedback loops, this can be very useful to detect basic blocks of the architecture and precise points where is possible or not to insert buses and internal control paths. Operations and data accesses for each sub-block can be measured independently. Moreover data flows across the defined buses can be measured.
- SIMD parallelism: since the execution keeps tracks of simultaneously active instruments and instances, it is possible to know the mean and worst case for these values and then estimate if a SIMD-like structure would bring a benefit or not to the speed of execution. Consider in fact that two simultaneously active instances of an instrument naturally induce a SIMD program structure.
- I/O channels: input and output to and from instruments and orchestra are easily measured, being them just special buses, and then indications about input/output ports can be derived.
- data variance: it is very important to be able to monitor variance in data flows and processing power along the execution. In fact, it is possible to argue that heavy sub-blocks with a low variance are more suitable for hardware implementation, while highly variant sub-blocks, characterized by different complexity when control parameters change, are more suitable for a software implementation. In this case the automation of the design is not straightforward, even if possible: the design goals are increasing in detail and complexity and they are approaching those of hardware/software co-design.

5.2. *Block-based System Design*

In Chapter 2 a new trend in SoC design, globally identified as block-based system design, has been introduced; inside this trend many different solutions are spreading on the market, but all of them are characterized by a similar goal.

Block-based system design aims at reducing design complexity and time-to-market providing the designer with some prepared and possibly reconfigurable hardware basic blocks (from adders to almost complete cores) that the designer can manipulate specifying the required system with some kind of high-level description language. This language can very well be an HDL language, but other object-oriented solutions have been proposed, where standard or extended programming languages are used to describe the chip functionality and interconnections. For example the solution proposed by Improv systems is interesting, where a specific JAVA compiler is used to reconfigure a VLIW processing unit in an FPGA-like fashion. Seen from another point of view, the reconfigurable VLIW processor, or the building blocks proposed by Extensa, defines a machine with an open instruction set that can be specified through a higher-level description. In such a design style, the minimal

granularity allowed for customization is that of the smallest unit that can be accessed by the designer via the programming language.

The design tool that could be potentially derived from the SAINT simulator is not far from this design approach. Basic building blocks corresponding to the complete instruction set must be made available and their characteristics known. Once this is accomplished, it is possible to analyze the application with defined complexity vectors that could exactly correspond to different granularities; for instance, if any instruction is considered on its own, with the maximum detail permitted by the instruction set, this means that adders, multipliers, noise generators etc. can be configured independently in number and connections; if, on the other side, operations are grouped into more abstract classes, like done for the simulations reported in this chapter or in the case of the MPEG conformance test, this means that design requirements can be detailed at the level of e.g. ALUs, divisors, effect block coprocessors and so on. In this sense there is here a much larger allowed flexibility in design constraints, in spite of the fact that the tool is small and fast.

The main limitation of the approach proposed in this dissertation in comparison with block-based design lies in the fact that the SAINT simulator is intrinsically based on MPEG-4 Structured Audio and then on floating-point data; a program must be written with a clear specification of control and signal variables, being all of them in 32-bit floating-point format. Although this constitutes a severe limitation to the generality of the target system, it covers a large part of currently available audio applications, since the mainstream in this field is rapidly moving toward this format; MPEG-4 is a proof for that.

CHAPTER 8. FUTURE WORK AND CONCLUSION

The exploitation of the work described in this dissertation is not finished yet. A first phase of it is indeed accomplished with the project ThreeDSPACE and the realization of its system. But ThreeDSPACE does not completely fulfill the functionality of MPEG-4, since it still does not include a real implementation of encoding and transmission, things that are fundamental to a multimedia framework. Moreover, the complete extension of BIFSAINT to support the full Advanced Audio BIFS functionality is still under construction because of the unstable situation in MPEG-4 Systems that did not allow this during ThreeDSPACE. All this will be done in the new project CARROUSO, which will be briefly described in the first section of this last chapter.

On the other side, it is interesting to further develop the design potential of the proposed method and in this sense new activities are also envisaged.

Final remarks and a brief summary of the overall work and its applications are contained in the concluding section.

1. A System for Advanced Applications: CARROUSO

As already discussed in Chapter 6, today's state of the art is that multichannel audio systems are capable to produce a good spatial impression and immersion for a very small region of the listening area, the so called "sweet spot". Even in a professional listening environment, the size of the sweet spot is a problem if several people are supposed to listen at the same time. The project ThreeDSPACE intends to break these problems merging a flexible and powerful coding technique such as MPEG-4 with the new and powerful WFS rendering technique. But if ThreeDSPACE aims at *demonstrating* the rendering potential of such a system, it is necessary to move fast towards an *application* of the complete MPEG-4 chain.

The key objective of the CARROUSO (for Creating, Assessing and Rendering in Real-time Of High-quality Audio-Visual Environments in MPEG-4 Context) project is to provide a new technology that enables to transfer a sound field, generated at a certain real or virtual space, to another remote located space; this will be possible with full interactive control of perceptually relevant natural, temporal and spatial properties of the sound space, especially in combination with the transmission of visual data. The basis for this technology is again WFS combined with MPEG-4, and then BIFSAINT, which opens windows to a wide range of new audio applications.

CARROUSO is a European collaboration assembling 11 industrial and academic partners from 5 different countries and will last for a period of 30 months.

1.1. A Complete MPEG-4 Coding Chain

To fulfill the CARROUSO objective, modeling, recording, encoding, decoding and rendering techniques have to be specified, developed, implemented, evaluated and validated. An experimental system incorporating the developed key building blocks

will be implemented during the project to validate the new technology on two specific, adequately chosen applications, among which a high-quality application for videoconferencing. If the outcome of ThreeDSPACE can be used as solid starting point for the rendering system, and some available transmission protocols can be exploited, all the rest must be developed, implemented and tested in the project.

1.1.1. Recording

In the space where the original sound field is generated, which can be called the recording space, microphone array technology will be applied to record the source signals and to determine the position of the sources, which may be fixed or moving. The source localization procedure will be enhanced by a video-based source tracking system. Microphone arrays will be used to gather relevant information on the acoustical conditions in the recording room. Acoustic models will be developed enabling to parameterize the acoustic data set, thus making it suitable for transmission. It is to be noted that these models will be able to treat data recorded in a real space as well as data simulated in a virtual space. All this information can be coded by the MPEG-4 Audio and Systems standards to compose a scene.

1.1.2. Transmission

Transmission will take place according to the existing MPEG standards. Encoding of audio objects will be operated by the MPEG-4 standard and encapsulated into specific data streams. For broadcasting applications, digital video broadcasting (DVB) streams will be adopted. The multiplexed streams will be transmitted through a real digital television-broadcasting platform. An alternate solution envisaged is the use of telecommunication networks.

1.1.3. Rendering

In the space where the sound field is to be reproduced, i.e. the rendering space, the transmitted data must be demultiplexed, decoded and processed by a compositor, from which it is fed to a WFS system, which as in ThreeDSPACE will consist of arrays of DSP-driven loudspeakers which reproduce the sound field of the recording space inside the rendering space.

The compositor will be equipped with a user interface much more powerful than the ThreeDSPACE one, such that the rendered result can be adapted interactively in all the aspects made available by MPEG-4 version 2 AABIFS.

The main contribution from BIFSAINT and its virtual approach will be of course in this end terminal block. It is possible to say that the work done during these years will constitute the heart of the CARROUSO rendering system for a couple of reasons:

- BIFSAINT will be finally extended to become a complete decoder for version 2 of Audio and Systems. Pre-processing support for the perceptual approach will be included and the geometrical approach will be better defined. Interface with streaming data flows will also be inserted and a better user application will be developed for satisfactory interaction
- the first phase of the project will redefine requirements and target platforms for the complete system. An eventual recognized need for semi-custom

solutions (custom board and/or FPGA programming for part of the rendering system) would benefit from the SAINT simulator for a high-level design of the final platform.

1.1.4. The CARROUSO System

The foreseen final outcome of the project will be the implementation, probably in a mixed software/hardware form, of a modularly designed experimental system for evaluation and broadcasting purposes. The experimental system will consist of the following basic parts, summarized for better clarity in Figure 55:

- a recording system consisting of a microphone array to measure the acoustical characteristics of the recording space and a camera to record and track moving acoustical sources.
- an MPEG-4 terminal containing encoding software, a graphical user interface, and interfaces to the recording system and the transmission channel, respectively.
- a transmission system consisting in a DVB channel carrying combined MPEG-4 and MPEG-2 audio and video bitstreams, and a local server containing multiplexing management tools.

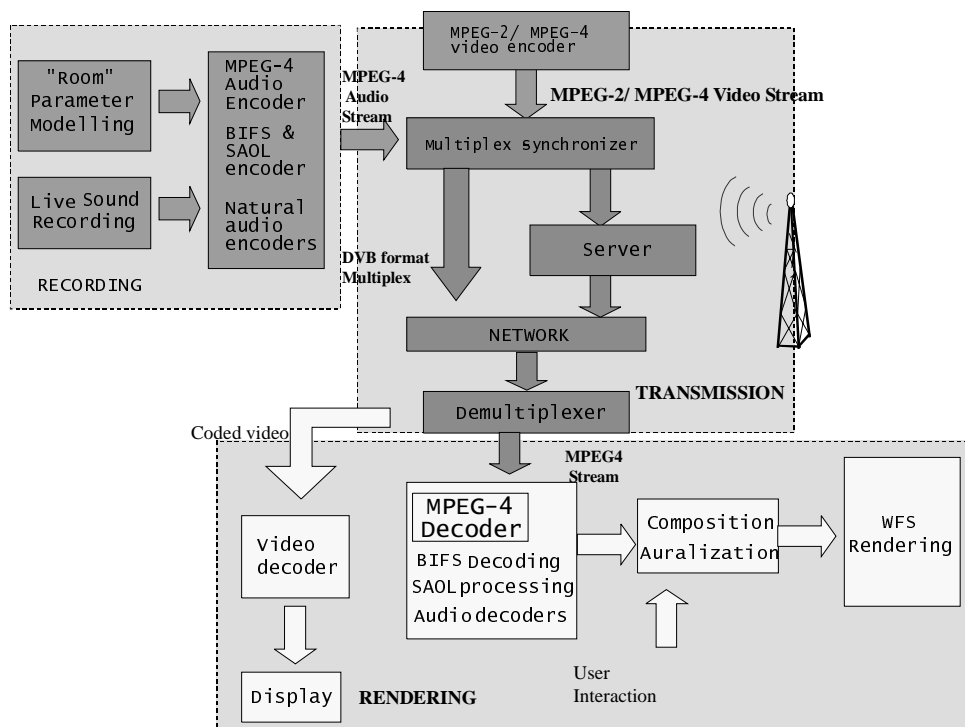


Figure 55 The high-level block diagram of the complete CARROUSO system

- an MPEG-4 end terminal including a demultiplexer, an audio/video decoder, an MPEG-4 scene information decoder, a compositor hardware and software, and a standardized interface to a wave field synthesis system (an enhancements of the ThreeDSPACE one described in Table 7), controlled by a graphical user interface. BIFS SAINT will constitute the core of this block.

- a modular rendering system of loudspeaker arrays as in ThreeDSPACE, with further enhancements for better response to dynamic interaction, enabling sound field reconstruction by wave field synthesis.

The CARROUSO experimental system will also incorporate a platform composed of a set of communication and management tools to allow easy setup and evaluation of the applications and validation of the complete experimental system chain.

1.2. Practical Application Scenarios

Two applications will be targeted into the CARRUSO project.

- the first one is a one-to-many configuration. It concerns 3D high quality audio with associated video for broadcasting. It is oriented towards the electronic consumer market.
- the second application is a one-to-one configuration for cooperative and interactive work on 3D audio objects. It is oriented towards the professional market.

The first one-to-many application aims at a broadcasting situation where the end user can choose different acoustical setups for a given content. This will allow him to have a local and limited interactivity changing the conditions of signal delivery. The service will be delivered as a digital television service with a high quality 3D audio.

For the second one-to-one application, a distant access to the control parameters of 3D audio will be allowed by exchanging information over a telecommunication network. This will enable to remotely change the rendering at the distant location. Simple parameters such as localization or reverberation parameters can be manipulated both at the local and distant location so that the WFS scene is shared even with some delays.

The project will generate and deliver after transmission specific audio signals processed by the complete chain developed in the project. This will allow demonstrating the technical feasibility and will help to achieve the first step of the introduction of new applications and services, such as cooperative work, many-to-many teleconferencing facilities, cyber teaching, science museums etc. Upon success, CARROUSO will be the final exploitation on the market of the BIFSAINT approach.

1.3. Further Future Applications of the SAINT Simulator

Other than a potential use in CARROUSO, at the same time other activities are envisaged to further develop and exploit the final part of this dissertation, i.e. the SAINT simulator.

Thanks to ThreeDSPACE and other projects, the LSI Laboratory grew in experience on the implementation of advanced 3-D Audio processing algorithms. To proceed further in this direction, some more activities are under study in particular with the company producing the Mykerinos sound board. The effective implementation of surround reverberators and tools for enhanced sound listening is the main target of this previewed activity.

3-D Audio algorithms have special requirements in memory and processing that are somewhat different from common audio effects, as already underlined. In this sense the SAINT simulator may be used to efficiently analyze the new algorithms, thanks to the fast modeling possible with MPEG-4 SA. Starting from this analysis, goal of the activity could also be that of extending the simulator to a high-level designer of the main elements of an enhanced board and/or of ad-hoc modifications to the default software API setup.

2. Conclusion

In this dissertation a virtual model has been presented for design-oriented simulation and analysis of Audio architectures in a multimedia context. The model is conceived for platform-independent complexity analyses of algorithms and is based on the concept of abstract classes of operations, where each class derives from decomposition of complex functions and grouping by successive refinement of common instruction sets of state-of-the-art multimedia-oriented devices.

Tools for simulation and analysis of architectures have spread from academic and industrial research laboratories above all in the last decade. Many of them are conceived to provide low-level exact simulation of the supported devices, at the price of heavy slowdowns in simulation times and in size of generated data reports. Some others, in the last years, introduced some degrees of approximation in simulations, in order to speed up execution time and to increase the flexibility of the tools to a real multi-architecture support. The resulting more or less abstract models are anyway not capable to analyze actual multimedia-oriented applications, where programs are usually available in some language including libraries and meaningful results are only those achieved in function of time.

On another level, tools for hardware/software codesign or for block-based system design provides more concrete results, but to be effective they must be based on rigid building blocks that may allow only some degrees of reconfigurability; some very recent tools are conceived to design complex systems by modeling blocks through a high level description language.

The virtual model and deriving tool described in this dissertation moves from an approach to the analysis of complexity that is as far as possible platform independent. Media applications are commonly programmed by imperative or object-oriented languages, which are composed by many different statements, operators and above all standard libraries. A careful profiling of typical applications permits to detect key operations and functions and to define a virtual instruction set composed by grouping of more or less similar operators and by decomposition of functions into similar building blocks. The virtual instruction set does not correspond to any actual one but it has as purpose to be easily mapped on a large number of existing ones. Simulation of an architecture requires then the availability of measures, benchmarks or estimation of at least one member of each abstract class. The number of classes, i.e. the complexity vector, can be adapted in length and detail to the needed degree of precision and to the available set of actual measures and benchmarks.

The proposed simulation tool derived from the virtual model is based on the MPEG-4 Structured Audio Orchestra Language, a language conceived for Audio synthesis and processing; the virtual instruction set is partially reconfigurable and permits different levels of detail and consequent abstraction.

The process of abstraction of the concept of complexity of a program led to two main practical results:

- the proposed method has been used to define complexity levels for Structured Audio in the MPEG-4 Standard, and consequently the implemented platform independent profiler has become the MPEG-4 reference software for the MPEG-4 SA Conformance test.
- a virtual instruction set has been conceived that has permitted the implementation of an optimized Structured Audio decoder, SAINT, based on a virtual interpreted DSP and corresponding bytecode. This decoder is in the mean 20 times faster than the MPEG reference software and can compete with desktop-oriented solutions based on SAOL-to-C cross compilation.

The flexibility of the SAINT virtual DSP approach has permitted a fast porting of its execution engine on a superscalar VLIW DSP, making it one of the building blocks of the ThreeDSPACE system, a framework for advanced rendering of 3-D Audio based on MPEG-4 scene descriptions.

A reduced, more abstract set, than the SAINT DSP one has been finally measured and used to simulate the SAINT decoder on two quite different platforms, based on a Pentium processor and on the Philips TriMedia VLIW processor, respectively; further, a block-based approximated cache simulator has been implemented. The resulting simulation tool has shown excellent results, achieving estimations of the SAINT execution time with an approximation of the 10% in the mean for the Pentium processor, and of the 20% for the very complex and not well-documented VLIW processor. Estimated programs have sometimes dynamic excursions of a factor 10 in their complexity along the time axis.

In general, the experimental results can be considered in line with those of the most recent, state-of-the-art simulators presented in literature. A drawback of the language adopted to model the application is that its generality is limited to one-dimensional floating-point computation; the major advantage is that simulation in function of the performance time is straightforward and provides the time-dependent results that are fundamental for the implementation of real-time applications. With the resulting tool, complex applications can be quickly modeled thanks to specific libraries and also quickly analyzed; moreover the tool can be easily extended to become a tool for automatic generation and configuration of the main building blocks of a system implementing the application, or the class of applications, under consideration.

The good acceptance of the proposed approach and of its results in both international conferences and consortia for international projects lets suppose that they are indeed of some value and the hope is that they will be exploited to their potential. In the next future the SAINT platform will be used for an important project on European scale. At the same time the acceptance of the tool for analysis and design is also tied to the acceptance of the MPEG-4 standard by the market, so

that its SAOL language can really become the internationally adopted language that will transform the SAINT simulator in a good reference for application modeling. Being the guess of the future always risky if not gambling, the author of this work is happy, for the moment, of feeling the taste of a good work accomplished but potentially not over.

APPENDIX A. THE SAINT VIRTUAL DSP INSTRUCTION SET

This Appendix provides a formal description of the vectorial instruction set implemented by the SAINT virtual DSP (vDSP). Instruction execution and general issues are further described in the vDSP specification in Chapter 5. Since this instruction set is not stored in a specific file format, it will not be described here in terms of bytes or words, rather in terms of functionality and interaction with memory; instructions will be identified by a textual symbolic opcode. Potential exceptions generated by the several instructions are not included in this appendix.

1. The SAINT Instruction Set

The SAINT instruction set is designed as a vectorial one; all instructions that supports vectorial execution will be called from here on *macroinstructions*, while the term *instruction* will be used for simple scalar operators, as described in Chapter 5. All the formal opcodes for macroinstructions have an *m* prefix.

Macroinstructions do not contain information about their width in the formal definition; rather all of them have access to special registers *start* and *stop*. These two registers contain the first and (last+1) offset that must be considered for the execution of an instruction. Offsets relate to a base address, which is the one contained in the parameters following the instruction formal code.

For instance:

```
madd x, a, b
```

will add two vectors of data whose locations in memory begin respectively in *a* and *b*. The result will be stored in a vector starting at *x*. During execution, if *start* is equal to 0 and *stop* is equal to 1, this means that the instruction is scalar and only two values are added and the single result stored. The same is valid if e.g. *start* is 32 and *stop* is 33, but this time operands are taken, and result is stored, with an offset of 32 positions starting from *a*, *b* and *x*. If *start* is equal to 0 and *stop* is equal to e.g. 100, this means that a vector of 100 samples starting from *a* must be added, sample-per-sample, with the one starting in *b*, and the result stored starting from *x*. In C language this can be written as:

```
for(i=0; i<100; i++)  
    x[i] = a[i] + b[i];
```

All macroinstructions operate on both variables and registers, since there is no difference between the two from a conceptual point of view. The same is valid for the target vector. The only difference between variables and registers is that the former are defined by the programmer and the latter by the compiler. All variables and registers are in 32-bit, simple precision floating-point format.

Instructions are different from macroinstruction in two ways: first of all they have access to the machine registers *start*, *end*, *length*, *program_counter* (*PC*) and can communicate with the scheduler; secondly, they do not produce processing or modify variables and vectors of the instance space, and normally do not operate on vectors (except if and else).

A final remark about SAINT instruction space; since opcodes in SAOL are static and they preserve their state from a call to the other, the same is necessary, for full support, for many of the macroinstructions of SAINT. For this purpose, each instruction in the dynamically allocated code to be executed has a dedicated static memory space where data can be saved from a call to the other (for instance coefficients of filters, internal phases and so on). Every time in the following description a reference is made to a state of an instruction, the information must be considered as stored in the static macroinstruction space.

1.1. Macroinstructions

1.1.1. *muminus*

muminus *x, a*

The **muminus** macroinstruction changes the sign of a vector of samples *a*, storing the result in a second vector *x*. The operation is executed in a sample-per-sample fashion on the operand vector, so that each element of the operand vector with the same offset is changed of sign and the result is stored in the target vector in the position characterized by the same offset.

1.1.2. *munot*

munot *x, a*

The **munot** macroinstruction negates a vector of samples *a*, storing the result in a second vector *x*. *x* will contain 1 for each element that is equal to zero, and 0 otherwise. The operation is executed in a sample-per-sample fashion on the operand vector, so that each element of the operand vector with the same offset is negated and the result is stored in the target vector in the position characterized by the same offset.

1.1.3. *minc*

minc *x, in, a*

The **minc** macroinstruction increments the elements of a vector of samples *in* by a quantity *a*, storing the result in a third vector *x*. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each element of the operand vector with a certain offset is incremented and the results stored in the target vector in the position characterized by the same offset.

1.1.4. *madd*

madd *x, a, b*

The **madd** macroinstruction adds two vectors of samples, *a* and *b*, storing the result in a third vector *x*. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each couple of elements of the operand vectors with the same offset is added and the results stored in the target vector in the position characterized by the same offset.

1.1.5. *msub*

msub *x, a, b*

The **msub** macroinstruction subtracts vector *b* from vector *a*, storing the result in a third vector *x*. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each couple of elements of the vectors with the same offset is used for subtraction and the results stored in the target vector in the position characterized by the same offset.

1.1.6. *mmul*

mmul *x, a, b*

The **mmul** macroinstruction multiplies vectors *a* and *b*, storing the result in a third vector *x*. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each couple of elements of the vectors with the same offset is used for multiplication and the results stored in the target vector in the position characterized by the same offset.

1.1.7. *mdiv*

mdiv *x, a, b*

The **mdiv** macroinstruction divides vector *a* by vector *b*, storing the result in a third vector *x*. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each couple of elements of the vectors with the same offset is used for division and the results stored in the target vector in the position characterized by the same offset.

1.1.8. *meqeq*

meqeq *x, a, b*

The **meqeq** macroinstruction executes a comparison for equality on vectors *a* and *b*, storing the result in a third vector *x*. *x* will contain 1 for each couple of elements that are equal in *a* and *b*, and 0 otherwise. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each couple of elements of the vectors with the same offset is used for comparison and the results stored in the target vector in the position characterized by the same offset.

1.1.9. *mneq*

mneq x,a,b

The **mneq** macroinstruction executes a comparison for inequality on vectors a and b, storing the result in a third vector x. x will contain 1 for each couple of elements that are not equal in a and b, and 0 otherwise. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each couple of elements of the vectors with the same offset is used for comparison and the results stored in the target vector in the position characterized by the same offset.

1.1.10. *mgeq*

mgeq x,a,b

The **mgeq** macroinstruction executes a comparison on vectors a and b, checking elements of a that are greater than or equal to elements of b and storing the result in a third vector x. x will contain 1 for each element of a that is greater than or equal to the corresponding element of b, and 0 otherwise. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each couple of elements with the same offset is used for comparison and the results stored in the target vector in the position characterized by the same offset.

1.1.11. *mleq*

mleq x,a,b

The **mleq** macroinstruction executes a comparison on vectors a and b, checking elements of a that are less than or equal to elements of b and storing the result in a third vector x. x will contain 1 for each element of a that is less than or equal to the corresponding element of b, and 0 otherwise. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each couple of elements with the same offset is used for comparison and the results stored in the target vector in the position characterized by the same offset.

1.1.12. *mgt*

mgt x,a,b

The **mgt** macroinstruction executes a comparison on vectors a and b, checking elements of a that are strictly greater than elements of b and storing the result in a third vector x. x will contain 1 for each element of a that is greater than the corresponding element of b, and 0 otherwise. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each couple of elements with the same offset is used for comparison and the results stored in the target vector in the position characterized by the same offset.

1.1.13. *mlt*

mlt x,a,b

The **mlt** macroinstruction executes a comparison on vectors a and b, checking elements of a that are strictly less than elements of b and storing the result in a third vector x. x will contain 1 for each element of a that is less than the corresponding element of b, and 0 otherwise. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each couple of elements with the same offset is used for comparison and the results stored in the target vector in the position characterized by the same offset.

1.1.14. *mint*

mint x,a

The **mint** macroinstruction calculates the integer part of vector a, storing the result in a second vector x. x will contain an integer value, in floating-point format, for each element of a. The operation is executed in a sample-per-sample fashion on the operand vector, so that each element is used for the calculation and the result is stored in the target vector in the position characterized by the same offset.

1.1.15. *mln*

mln x,a

The **mln** macroinstruction calculates the natural logarithm of vector a, storing the result in a second vector x. x will contain a result for each element of a. The operation is executed in a sample-per-sample fashion on the operand vector, so that each element is used for the calculation and the result is stored in the target vector in the position characterized by the same offset.

1.1.16. *mpow*

mpow x,a,b

The **mpow** macroinstruction calculates vector a elevated to the bth power, storing the result in a third vector x. The operation is executed in a sample-per-sample fashion on the operand vectors, so that each couple of elements with the same offset is used to calculate the results $a[i]b[i]$, which are stored in the target vector in the position characterized by the same offset.

1.1.17. *msgn*

msgn x,a

The **msgn** macroinstruction calculates the sign of vector a, storing the result in a second vector x. x will contain an integer value, in floating-point format, for each element of a; x will contain 1 for each element that is strictly greater than 0, -1 for each element that is strictly less than 0, 0 if the element is 0. The operation is executed in a sample-per-sample fashion on the operand vector, so that each

element is used for the calculation and the result is stored in the target vector in the position characterized by the same offset.

1.1.18. *msqrt*

msqrt *x, a*

The **msqrt** macroinstruction calculates the square root of vector *a*, storing the result in a second vector *x*. *x* will contain a result for each element of *a*. The operation is executed in a sample-per-sample fashion on the operand vector, so that each element is used for the calculation of the square root and the result is stored in the target vector in the position characterized by the same offset.

1.1.19. *msin*

msin *x, a*

The **msin** macroinstruction calculates the sinus of vector *a* expressed in radians, storing the result in a second vector *x*. *x* will contain a result for each element of *a*. The operation is executed in a sample-per-sample fashion on the operand vector, so that each element is used for the calculation of the sinus and the result is stored in the target vector in the position characterized by the same offset.

1.1.20. *mcos*

mcos *x, a*

The **mcos** macroinstruction calculates the cosinus of vector *a* expressed in radians, storing the result in a second vector *x*. *x* will contain a result for each element of *a*. The operation is executed in a sample-per-sample fashion on the operand vector, so that each element is used for the calculation of the cosinus and the result is stored in the target vector in the position characterized by the same offset.

1.1.21. *masin*

masin *x, a*

The **masin** macroinstruction calculates the inverse sinus of vector *a*, storing the result, expressed in radians, in a second vector *x*. *x* will contain a result for each element of *a*. The operation is executed in a sample-per-sample fashion on the operand vector, so that each element is used for the calculation of the inverse sinus and the result is stored in the target vector in the position characterized by the same offset.

1.1.22. *macos*

macos *x, a*

The **macos** macroinstruction calculates the inverse cosinus of vector *a*, storing the result, expressed in radians, in a second vector *x*. *x* will contain a result for each

element of *a*. The operation is executed in a sample-per-sample fashion on the operand vector, so that each element is used for the calculation of the inverse cosine and the result is stored in the target vector in the position characterized by the same offset.

1.1.23. *mmin*

mmin *x, a, b*

The **mmin** macroinstruction calculates the minimum of two vectors *a* and *b*, storing the result in a vector *x*. The operation is executed in a sample-per-sample fashion on the operand vectors, so that all the *n* elements with the same offset are used and their minimum is calculated, which is stored in the target vector in the position characterized by the same offset.

1.1.24. *mmax*

mmax *x, a, b*

The **mmax** macroinstruction calculates the maximum of two vectors *a* and *b*, storing the result in a vector *x*. The operation is executed in a sample-per-sample fashion on the operand vectors, so that all the *n* elements with the same offset are used and their maximum is calculated, which is stored in the target vector in the position characterized by the same offset.

1.1.25. *mgettune*

mgettune *x*

The **mgettune** macroinstruction reads the value of the special register *mtune*, containing the master tune of the orchestra, and write this value into the elements of a vector *x*. For further information about the master tune, see ISO/IEC 14496/3-sub5 [69].

1.1.26. *msettune*

msettune *a*

The **msettune** macroinstruction sets the value of the special register *mtune*, containing the master tune of the orchestra, according to the value of the variable *a*. For further information about the master tune, see ISO/IEC 14496/3-sub5 [69].

1.1.27. *mcheck*

mcheck *x, a*

The **mcheck** macroinstruction checks the values in the vector *x* against a fixed value and saturates them; according to the value of the first parameter from *a*, values of *x* are checked one by one for being greater than the second parameter, inside the interval specified by the second and third parameter, less than the second

parameter. In the three cases, values not verifying the condition are saturated to the value of the condition.

1.1.28. *malloc*

malloc *t,n*

The **malloc** macroinstruction allocates a memory structure for a table *t*, composed by five parameters and *n* single-precision floating-point samples. The five parameters are, in order: length, loop start point, loop end point, sampling rate and base frequency of the table.

1.1.29. *mshift*

mshift *t,n*

The **mshift** macroinstruction shifts all the elements of a table *t* by *n* samples. The first *n* samples are discarded while the last *n* samples are filled with zeros.

1.1.30. *mget_tab_parms*

mget_tab_parms *x,t,off*

The **mget_tab_parms** macroinstruction reads the parameter of the table *t* with offset *off* starting from *t*, storing the result into the elements of a vector *x*. According to the format of the table memory structure, allowed offsets from 1 to 5 correspond to the length, loop start point, loop end point, sampling rate and base frequency of the table respectively.

1.1.31. *mset_tab_parms*

mset_tab_parms *a,t,off*

The **mget_tab_parms** macroinstruction sets the parameter of the table *t* with offset *off* starting from *t*, according to the value contained in the variable *a*. According to the format of the table memory structure, allowed offsets from 2 to 5 correspond to the loop start point, loop end point, sampling rate and base frequency of the table respectively.

1.1.32. *minterp*

minterp *x,t,ind*

The **minterp** macroinstruction executes an interpolated reading of the table *t* at points contained in vector *ind*, storing the result into the elements of a vector *x*. The interpolation factor depends on the value of the special register *interp*, which is set only at the beginning of the orchestra by the scheduler and never modified again. The operation is executed in a sample-per-sample fashion on the operand vectors; if the value of *interp* is equal to 0 values written in each element of *x* are linearly interpolated from the two nearest points corresponding to the value of *ind*

with the same offset; if the value of `interp` is equal to 1 values written in each element of `x` are interpolated with a factor 3 from the nearest points corresponding to the value of `ind` with the same offset.

1.1.33. *mtablewrite*

```
mtablewrite a,t,ind
```

The **mtablewrite** macroinstruction writes into the table `t` at points contained in a vector `ind`, reading the values from a vector `a`. The exact point where the value is written is given by rounding each element of `ind` to the nearest integer value. The operation is executed in a sample-per-sample fashion on the operand vectors; from each element of `a`, values are written in the table at a position calculated rounding the value of `ind` with the same offset to the nearest integer value.

1.1.34. *mphasor*

```
mphasor x,inc
```

The **mphasor** macroinstruction produces moving phase values looping from 0 to 1 repeatedly with an increment `inc`. The increment is executed with modulo 1. The first time that the macroinstruction is called with regard to a particular state the internal phase starts from 0, while on subsequent calls the phase is saved from previous calls. At each execution of the macroinstruction a vector of phases is produced and stored in the vector `x`.

1.1.35. *mload*

```
mload t,f,n
```

The **mload** macroinstruction loads into a table or a vector `t` the values of the `n` scalar variables listed starting from `f`. Each time the macroinstruction is executed a vector of values is loaded in the vector `x`.

1.1.36. *mstep*

```
mstep x,f,n
```

The **mstep** macroinstruction produces a line-segmented step signal according with the `n` parameters listed starting from `f`, storing the result in a table or vector `t`. The `n` parameters are structured as a sequence of signal values and segment durations, so that the result is the value of the first parameter for a number of samples contained in the second parameter; once this is done the result is the value of the third parameter for a number of samples contained in the fourth parameter, and so on. When the `n` parameters have been used the result is always 0. If `n` is even only `n-1` parameters are used before setting the result to 0.

1.1.37. *mline*

mline *x, f, n*

The **mline** macroinstruction produces a line-segmented ramp signal according with the *n* parameters listed starting from *f*, storing the result in a table or vector *x*. The *n* parameters are structured as a sequence of signal values and ramp durations, so that the result moves linearly from the value of the first parameter to the value of the third parameter in a number of samples contained in the second parameter; once this is done the result moves linearly to the value of the fifth parameter in a number of samples contained in the fourth parameter, and so on. When the *n* parameters have been used the result is always 0. If *n* is even only *n*-1 parameters are used before setting the result to 0.

Each time the macroinstruction is executed a vector of values is calculated and stored in the vector *x*.

1.1.38. *mexpon*

mexpon *r, f, n*

The **mexpon** macroinstruction produces a segmented signal made out of exponential curves according with the *n* parameters listed starting from *f*, storing the result in a table or vector *r*. The *n* parameters are structured as a sequence of signal values and segment durations, so that the result moves exponentially from the value of the first parameter to the value of the third parameter in a number of samples contained in the second parameter; once this is done the result moves exponentially to the value of the fifth parameter in a number of samples contained in the fourth parameter, and so on. When the *n* parameters have been used the result is always 0. If *n* is even only *n*-1 parameters are used before setting the result to 0. The exponential law of the result is:

$$r = y_k (y_{k+1} / y_k) \exp((x - x_k) / (x_{k+1} - x_k))$$

where *x* is the axis of samples and *y* is the axis of the result signal; *k* and *k*+1 indicate the first and the second point of the segment respectively.

Each time the macroinstruction is executed a vector of values is calculated and stored in the vector *x*.

1.1.39. *mcubicseg*

mcubicseg *r, f, n*

The **mcubicseg** macroinstruction produces a segmented signal made out of segment of cubic polynomials according with the *n* parameters listed starting from *f*, storing the result in a table or vector *r*. The *n* parameters are structured as a sequence of signal values, segment durations and inflection points; the result moves with a cubic law from the value of the second parameter to the value of the fourth parameter in a number of samples contained in the third parameter with inflection

point at the value of the first and fifth parameter, and so on. When the n parameters have been used the result is always 0. If n is not multiple of 2 the last parameters are discarded and only n-1 parameters are used before setting the result to 0. The cubic polynomial law of the result from infl_k to infl_{k+1} is:

$$r = ax^3 + bx^2 + cx + d$$

where a, b, c and d are the coefficients of a cubic polynomial that passes through (infl_k, y_k) , (x_k, y_{k+1}) and $(\text{infl}_{k+1}, y_{k+2})$ and that has 0 derivative at x_k . x is the axis of samples and y is the axis of the result signal; k and k+1 indicate the first and the second point of the segment respectively.

Each time the macroinstruction is executed a vector of values is calculated and stored in the vector x.

1.1.40. *mspline*

mspline r, f, n

The **mspline** macroinstruction produces a smooth spline signal according with the n parameters listed starting from f, storing the result in a table or vector r. The n parameters are structured as a sequence of signal values, segment durations and derivative values; the result moves with a cubic law from the value of the first parameter to the value of the fourth parameter in a number of samples contained in the second parameter with a derivative at the second extreme equal to the third parameter, and so on. When the n parameters have been used the result is always 0. If n is not multiple of 3 the last parameters are discarded and only n-1 or n-2 parameters are used before setting the result to 0. The cubic polynomial law of the result from x_k to x_{k+1} is:

$$r = ax^3 + bx^2 + cx + d$$

where a, b, c and d are the coefficients of a cubic polynomial that passes through (x_k, y_k) and (x_{k+1}, y_{k+1}) and that has z_k derivative at x_k and z_{k+1} at x_{k+1} . x is the axis of samples and y is the axis of the result signal r; k and k+1 indicate the first and the second point of the segment respectively. z are the derivative values.

Each time the macroinstruction is executed a vector of values is calculated and stored in the vector x.

1.1.41. *mpluck*

mpluck x, f, par

The **mpluck** macroinstruction calculates a signal according to the Karplus-Strong algorithm at the frequency in vector f, using the four parameters contained in par and storing the result in a vector x. The exact algorithm is the same as described in ISO/IEC 14996/3sub5, subclause 5.9.7.7. Each time the macroinstruction is executed a vector of values is calculated and stored in the vector x.

1.1.42. *mbuzz*

mbuzz *x, f, par*

The **mbuzz** macroinstruction calculates a band-limited pulse train signal at the frequency in vector *f*, using the three parameters contained in *par* and storing the result in a vector *x*. The exact algorithm is the same as described in ISO/IEC 14996/3sub5, subclause 5.9.7.8. Each time the macroinstruction is executed a vector of values is calculated and stored in the vector *x*.

1.1.43. *mgrain*

mgrain *x, n, par*

The **mgrain** macroinstruction calculates granular synthesis signals using the *n* parameters contained starting from *par* and storing the result in a vector *x*. The exact algorithm is the same as described in ISO/IEC 14996/3sub5, subclause 5.9.7.9. Each time the macroinstruction is executed a vector of values is calculated and stored in the vector *x*.

1.1.44. *mrnd*

mrnd *x, p*

The **mrnd** macroinstruction generates a vector of random numbers according to a linear distribution between $-p$ and p . Each time the macroinstruction is executed a vector of values is calculated and stored in the vector *x* according to the probability density function:

$$p(x) = 1/2p : x \in [-p, p]$$

and 0 elsewhere.

1.1.45. *mport*

mport *x, ctrl, par*

The **mport** macroinstruction converts a step-valued control signal into a portamento signal, storing the result in a vector *x*. The exact algorithm is the same as described in ISO/IEC 14996/3sub5, subclause 5.9.9.1. Each time the macroinstruction is executed a value is calculated and stored in the vector *x*.

1.1.46. *mhipass*

mhipass *x, in, cut*

The **mhipass** macroinstruction high-pass filters its input signal contained in vector *in*, storing the result in a vector *x*. *cut* is the -6 dB cut-off point of the filter in Hz. The frequency mask of the filter has a -20dB/octave slope in the transition band and a maximum 3 dB ripple in the passband. Each time the macroinstruction is executed a vector of values is calculated and stored in the result vector *x*.

1.1.47. *mlopass*

mlopass x,in,cut

The **mlopass** macroinstruction low-pass filters its input signal contained in vector in, storing the result in a vector x. cut is the -6 dB cut-off point of the filter in Hz. The frequency mask of the filter has a -20dB/octave slope in the transition band and a maximum 3 dB ripple in the passband. Each time the macroinstruction is executed a vector of values is calculated and stored in the result vector x.

1.1.48. *mbandpass*

mbandpass x,in,bn

The **mbandpass** macroinstruction band-pass filters its input signal contained in vector in, storing the result in a vector x. bn is the address of a 2-value structure containing center frequency and bandwidth (-6 dB cut-off points) of the filter passband in Hz. The frequency mask of the filter has a -20dB/octave slope in the transition bands and a maximum 3 dB ripple in the passband. Each time the macroinstruction is executed a vector of values is calculated and stored in the result vector x.

1.1.49. *mbandstop*

mbandstop x,in,bn

The **mbandpass** macroinstruction band-stop filters its input signal contained in vector in, storing the result in a vector x. bn is the address of a 2-value structure containing center frequency and bandwidth (-6 dB cut-off points) of the filter stopband in Hz. The frequency mask of the filter has a -20dB/octave slope in the transition bands and a maximum 3 dB ripple in the passbands. Each time the macroinstruction is executed a vector of values is calculated and stored in the result vector x.

1.1.50. *mallpcomb*

mallpcomb x,in,par

The **mallpcomb** macroinstruction performs allpass or comb filtering on its input signal contained in vector in, storing the result in a vector x. par is the address of a memory space containing three parameters specifying if allpass or comb is to be performed, feedback delay and gain of the filter. The exact algorithm is the same as described in ISO/IEC 14996/3sub5, subclauses 5.9.9.7 and 5.9.9.8. Each time the macroinstruction is executed a vector of values is calculated and stored in the result vector x.

1.1.51. *mbiquad*

mbiquad x,in,par

The **mbiquad** macroinstruction performs normative canonical second-order filtering (Transposed Direct Form II) on its input signal contained in vector in, storing the result in a vector x. par is the adress of a memory space containing the five parameters specifying the biquad filter. Each time the macroinstruction is executed a vector of values is calculated and stored in the result vector x.

1.1.52. *mfilter*

mfilter n,x,in,par

The **mfilter** macroinstruction performs nth order filtering on its input signal contained in vector in, storing the result in a vector x. par is the address of a memory space containing the 2*n parameters specifying the filter. The **mfilter** macroinstruction is a Transposed Direct Form II implementation of the standard difference equation. Each time the macroinstruction is executed a vector of values is calculated and stored in the result vector x.

1.1.53. *mcompress*

mcompress x,in,comp,par

The **mcompress** macroinstruction operates a compressor/limiter/expander function on its input in by the control signal comp, using the seven parameters stored starting from par and storing the result in a vector x. The exact algorithm is the same as described in ISO/IEC 14996/3sub5, subclause 5.9.11.4. Each time the macroinstruction is executed a vector of values is calculated and stored in the vector x.

1.1.54. *mrms*

mrms x,in

The **mrms** macroinstruction calculates the power in the elements of its input vector in, storing the result in a variable x. Each time the macroinstruction is executed a value is calculated as follows:

$$x = \sqrt{\frac{\sum_{j=0}^{I-1} in[j]^2}{I}}$$

where I is the length of a vector of samples, B₁. If a block of code is executed in a sample-by-sample fashion, the value of x is updated only after the last execution of the sample-by-sample loop.

1.1.55. *mupsamp*

mupsamp x,k

The **mupsamp** macroinstruction upsamples a scalar variable (an initialization or control variable with width equal to 1) to a vector x, where all the elements of x contain the same value k.

1.1.56. *mscalar_prod*

mscalar_prod x,in,t

The **mscalar_prod** macroinstruction multiplies vector in and the first B_i elements of table t, storing the result in a second vector x. The operation is executed in a sample-per-sample fashion on the operand vectors and table, so that each couple of elements of the vector and of the table with the same offset is used for the multiplication and the results are stored in the target vector in the position characterized by the same offset.

1.1.57. *mdline*

mdline x,in,t

The **mdline** macroinstruction implements a fixed-length delay line of length $\text{floor}(t*s_rate)$ samples, inserting at the beginning of the delay line the values of the input vector in and storing the output values (in FIFO management fashion) into the vector x.

1.1.58. *mchorus*

mchorus x,in,par

The **mchorus** macroinstruction creates a sound with a chorusing effect on the input vector in, using the two parameters (rate and depth) at the address starting from par and storing the result in a vector x. The exact algorithm is a two-phase chorus with triangular delay law. Each time the macroinstruction is executed a vector of values is calculated and stored in the vector x.

1.1.59. *mflange*

mflange x,in,par

The **mflange** macroinstruction creates a sound with a flanged effect on the input vector in, using the two parameters (rate and depth) at the address starting from par and storing the result in a vector x. The exact algorithm is a two-phase flange with triangular delay law. Each time the macroinstruction is executed a vector of values is calculated and stored in the vector x.

1.1.60. *mreverb*

mreverb *x, in, n, par*

The **mreverb** macroinstruction creates a sound with a reverberation effect on the input vector *in*, using the *n* parameters at the address starting from *par* and storing the result in a vector *x*. The exact meaning of the *n* parameters is explained in ISO/IEC 14996/3sub5. Reverberation is calculated by means of feedback delay networks. Each time the macroinstruction is executed a vector of values is calculated and stored in the vector *x*.

1.1.61. *mget_tempo*

mget_tempo *x*

The **mget_tempo** macroinstruction reads the parameter of the global variable *tempo*, storing the result into the elements of a vector *x*.

1.1.62. *mset_tempo*

mset_tempo *a*

The **mset_tempo** macroinstruction sets the parameter of the global variable *tempo*, according to the value contained in the control variable *a*.

1.1.63. *mwindow*

mwindow *r, ty, p*

The **mwindow** macroinstruction produces a windowing signal according with the type *ty* and to the parameter *p*, storing the result in a vector *r*. The exact algorithm is the same as described in ISO/IEC 14996/3sub5, subclause 5.10.10. Each time the macroinstruction is executed a vector of values is calculated and stored in the vector *x*.

1.1.64. *mpoly, mharm, mharm_phase, mperiod*

m* *r, f, n*

The **mpoly**, **mharm**, **mharm_phase**, **mperiod** macroinstruction produce special signals according to the *n* parameters at the address starting from *f*, storing the result in a vector *r*. The exact algorithms are the same as described in ISO/IEC 14996/3sub5, subclause 5.10.11 to 5.10.14. These special functionality is used to rapidly fill vectors with functions used in audio synthesis and in particular an arbitrary polynomial function, a linear combination of harmonics, a linear combination of harmonics with initial phase and an arbitrary periodic function composed by sinusoids at arbitrary functions. Each time the macroinstruction is executed a vector of values is calculated and stored in the vector *x*.

1.1.65. *moutput*

moutput a

The **moutput** macroinstruction adds the vector a to the current values of the instrument output bus. The operation is executed in a sample-per-sample fashion on the operand vector, so that each element of the operand vector with the same offset is changed of sign and the result is stored in the target vector in the position characterized by the same offset.

1.2. *Instructions*

1.2.1. *if*

if x, sub

The **if** instruction executes a sub-block of code according to the value of a vector x. The elements of x are evaluated in order. When an element equal to one is encountered, the *return* register is set to the current value of the PC, the PC is set to sub, the *start* register is set to the index of the current element of x and the *end* register is set to the first element further encountered that is equal to zero. The block of code starting at sub is then executed. This procedure is repeated until the whole vector x has been scanned.

1.2.2. *else*

else x, sub1, sub2

The **else** instruction executes one of two sub-blocks of code according to the value of a vector x. The elements of x are evaluated in order.

If an element equal to one is encountered, the *return* register is set to the current value of the PC, the PC is set to sub1, the *start* register is set to the index of the current element of x and the *end* register is set to the first element further encountered that is equal to zero. The block of code starting at sub is then executed.

If an element equal to zero is encountered, the *return* register is set to the current value of the PC, the PC is set to sub2, the *start* register is set to the index of the current element of x and the *end* register is set to the first element further encountered that is equal to one. The block of code starting at sub is then executed.

This procedure is repeated until the whole vector x has been scanned.

1.2.3. *p_set*

p_set reg, a

The **p_set** instruction sets the reg register to a. reg must be one of the registers *start*, *end* or *length*.

1.2.4. *p_inc*

p_inc reg

The **p_set** instruction increments the reg register by one. reg must be one of the registers *start*, *end* or *length*.

1.2.5. *p_return*

p_jump sub

The **p_jump** instruction sets PC to the value of the *return* register.

1.2.6. *p_jump*

p_jump sub

The **p_jump** instruction sets the *return* register to the next value of the PC and set the PC to sub.

1.2.7. *extend*

extend ti

The **extend** instruction signals to the scheduler that the currently active instance must be lengthened by a time ti in seconds at the next control cycle, i.e. its scheduled termination time T will become T+ti.

1.2.8. *turnoff*

turnoff

The **turnoff** instruction signals to the scheduler that the currently active instance must be scheduled for termination at the next control cycle.

1.2.9. *speedc*

speedc f

The **speedc** instruction modify the output of the orchestra with a speed change effect according to the speed change factor f. This instruction directly operates on the special bus input_bus of the orchestra. The exact way in which this instruction is executed and the exact algorithm are described in ISO/IEC 14996/3sub5, subclause 5.9.14.4 and Annex 5.D.

APPENDIX B. THE SAINT SIMULATOR COMPLEXITY VECTOR

The following table reports the complexity vector that has been used in the SAINT simulator to emulate the SAINT decoder as reported in Chapter 7. The first column contains the 20 elements of the vector, the second contains the measured times in microseconds for the vector on the Intel-based platform (BorlandC++ 5.02 compiler), the third contains the measured times in microseconds for the vector on the TriMedia-based platform.

Vector element	Time on PC platform (microseconds)	Time on TM platform (microseconds)
Macroinstruction calls	0.88	1.76
Arithmetic operations	0.07	0.04
Logic operations	0.07	0.04
Divisions	0.075	0.18
Tests	0.07	0.04
Trigonometric functions	1.85	0.24
Math functions	0.46	0.18
Noise generators	0.25	n.a.
Interpolations	2.06	0.28
Phasors	0.33	0.27
AllpCombs	0.38	0.031
MACs	0.165	0.031
Filters	0.64	0.42
Effects	n.a.	n.a.
Round-Int-Abs	0.15	0.14
Summing Bus	0.11	0.08
Memory reads	0.015	0.25
Memory writes	0.015	0.125
Allocated memory	0	0
Tables memory	0	0

REFERENCES

- [1] Semiconductor Industry Association "The International Roadmap for Semiconductors". San Jose', U.S.A., 1999.
- [2] Pierfrancesco Bellini, Fabrizio Fioravanti, Paolo Nesi "Managing Music in Orchestras". IEEE Computer, vol. 32, no. 9, pages 26-34, September 1999.
- [3] Christoforos Kozyrakis, David A. Patterson "A New Direction for Computer Architecture Design". IEEE Computer, vol. 31, no. 11, pages 24-32, November 1998.
- [4] Michael Barr "Developing Embedded Software in Java". Proceedings of the Embedded Systems Conference, San Jose, U.S.A., September 1999.
- [5] R. K. Gupta "Co-Synthesis of Hardware and Software for Digital Embedded Systems". Kluwer Academic Publishers, 1995.
- [6] Barbara Tuck "SoC design: Hardware/Software Codesign or a Java-based Approach ?". Computer Design, vol. 4, no. 4, April 1998.
- [7] David Bursky "Optimized Processor Blocks Eliminate The Gamble with RISC for SoC Designs". Electronic Design, May 1, 2000.
- [8] Keith Diefendorff, Pradeep Dubey "How Multimedia Workloads Will Change Processor Design". IEEE Computer, vol. 30, no. 9, pages 43-45, September 1997.
- [9] Grant Erickson "RISC for Graphics: A Survey and Analysis of Multimedia Extended Instruction Set Architectures". Internal Report, University of Minnesota, December 1996.
- [10] Alex Peleg, Sam Wilkie, Uri Weiser "Intel MMX for Multimedia PCs". Communications of the ACM, vol. 40, no. 1, January 1997.
- [11] Michael J. Flynn "Computer Architecture: Pipelined and Parallel Processor Design". Jones and Bartlett Publishers, Sudbury, U.S.A., 1997.
- [12] David Skillicorn "Foundations of Parallel Programming". Cambridge University Press, pages 1-48, Cambridge, U.K., 1994.
- [13] Cade Metz "500-MHz Pentium III PCs - Extending the Pentium III". PC Magazine, March 23, 1999.
- [14] Intel Corporation "Itanium Processor Microarchitecture Reference for Software Optimization". Made available at the Intel Web site <http://developer.intel.com>, March 2000.
- [15] Texas Instruments "TMS320C62xx Technical Brief". Made available at the Texas Instrument Web site <http://dspvillage.ti.com>, October 1997.

- [16] Ralf Steinmetz "Analyzing the Multimedia Operating System" IEEE Multimedia, vol. 2, no. 1, Spring 1995.
- [17] J. Montanaro et al. "A 160 MHz 32b 0.5W CMOS RISC Processor". IEEE Journal of Solid-State Circuits, vol. 31, no. 11, November 1996.
- [18] J. Panfil "The TinyRISC CPU - An Area Efficient CPU Core Optimized for Embedded Applications". Proceedings of the European Microprocessor and Microcontroller Conference, Heathrow, U.K., November 1996.
- [19] Yiannakis Sazeides, James E. Smith "Modeling Program Predictability". Proceedings of the 25th International Symposium on Computer Architecture, Barcelona, Spain, July 1998.
- [20] Yale Patt et al. "One Billion Transistor, One Uniprocessor, One Chip". IEEE Computer, vol. 30, no. 9, pages 51-57, September 1997.
- [21] Kunle Olukotun et al. "The Case for a Single-Chip Multiprocessor". Proceedings of the 7th ACM International Conference On Architectural Support for Languages and Operating Systems, pages 2-11, New York, U.S.A., 1996.
- [22] David Patterson et al. "A Case for Intelligent DRAM: IRAM". IEEE Micro, vol. 17, no. 2, April 1997.
- [23] Elliot Waingold et al. "Baring It All to Software: Raw Machines". IEEE Computer, vol. 30, no. 9, pages 86-93, September 1997.
- [24] Ken Arnold, James Gosling, David Holmes "The Java Programming Language, Third Edition". Addison Wesley, 2000.
- [25] Sun Microsystems Incorporated "JINI Architectural Overview". Technical White Paper, Palo Alto, U.S.A., January 1999.
- [26] Bruce Schneier "Why Cryptography is Harder than it Looks". Information Security Bulletin, vol. 2, no. 2, pages 31-36, March 1997.
- [27] Linda Geppert, T. Perry "Transmeta's Magic Show". IEEE Spectrum, vol. 37, no. 5, pages 27-33, May 2000.
- [28] Jennifer Anderson et al. "Transparent, Low-Overhead Profiling on Modern Processors". 1998 Workshop on Profile and Feedback-Directed Compilation, Paris, France, October 1998.
- [29] James Larus "Efficient Program Tracing". IEEE Computer, vol. 26, no. 5, May 1993.
- [30] Alvin R. Lebeck, David A. Wood "Active Memory: A New Abstraction for Memory-System Simulation". Proceedings of the 1995 ACM SIGMETRICS Conference, May 1995.
- [31] Shubhendu Mukherjee et al. "Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator". Workshop on Performance Analysis and Its Impact on Design - PAID, June 1997.

- [32] Thomas Ball, James Larus "Optimally Profiling and Tracing Programs". ACM Transactions on Programming Languages and Systems, vol. 16, no. 4, pages 1319-1360, July 1994.
- [33] Robert Cmelik, David Keppel "SHADE: A Fast Instruction-Set Simulator for Execution Profiling". 1994 ACM Conference on Measurement and Modeling of Computer Systems, 1994.
- [34] Robert Cmelik "The Shade User's Manual". Sun Microsystems Laboratories, Inc., February 1993.
- [35] Emmet Witchel, Mendel Rosenblum "Embra: Fast and Flexible Machine Simulation". Proceedings of the 1996 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems, pages 128-137, May 1994.
- [36] M.Rosenblum, S. Herrod, A. Gupta "Complete Computer System Simulation: The SimOS Approach". IEEE Parallel & Distributed Technology, vol. 4, no. 3, 1995.
- [37] M. Rosenblum, E. Bugnion, S. Devine, S. Herrod "Using the SimOS Machine Simulator to Study Complex Computer Systems". ACM Transactions on Modeling and Computer Simulation, vol. 7, no. 1, pages 78-103, January 1997.
- [38] Peter S. Magnusson et al. "SimICS/sun4m: A Virtual Workstation". USENIX Annual Technical Conference, New Orleans, U.S.A., June 1998.
- [39] A.D.Pimentel, L.O.Hertzberger "Abstract Workload Modeling in Computer Architecture Simulation". Proceedings of the 24th ACM/IEEE International Symposium on Computer Architecture, Denver, U.S.A., June 1997.
- [40] Jianwen Zhu, Daniel Gajski "A Retargetable, Ultra-Fast Instruction Set Simulator". Proceedings of the Design, Automation and Test in Europe Conference, DATE '99.
- [41] J. Rowson "Hardware/Software Co-simulation". Proceedings of the 31st ACM/IEEE Design Automation Conference, 1994.
- [42] Peter Voigdt Knudsen, Jan Madsen "Aspects of System Modelling in Hardware/Software Partitioning". Proceedings of the 7th IEEE International Workshop on Rapid Systems Prototyping, pages 18-23, Thessaloniki, Greece, June 1996.
- [43] B. Tabbara, L. Lavagno, and A. Sangiovanni-Vincentelli "Fast Hardware-Software Co-Simulation using Software Synthesis and Estimation". IEEE International High Level Design Validation and Test Workshop, 1997.
- [44] Stephen Edwards et al. "Design of Embedded Systems: Formal Models, Validation and Synthesis". Proceedings of the IEEE, vol. 85, no. 3, March 1997.
- [45] Synopsys Corporation "Designing Complex Digital Communications for Systems on a Chip". Available on the Internet at www.synopsys.com/products/dsp/digital_br.html.
- [46] CoWare Corporation "CoWare N2C Design System". Available on the Internet at www.CoWare.com.
- [47] Barbara Tuck "The hardware/software coverification challenge". Computer Design, vol. 4, no. 4, April 1998.

- [48] Dave Bursky "Tool Suite Enables Designers To Craft Customized Embedded Processors" *Electronic Design*, Feb. 8, 1999.
- [49] Cary Ussery "Configurable VLIW Uniquely Suited for Today's SoC Designs". Improv Systems internal report, U.S.A. 1999.
- [50] Jacob Greidinger "Managing Complex IC Design Challenges Requires Block-Level Design Plan Synthesis". Aristo Technology, Inc., Technical Report, <http://www.aristotech.com>, 1999.
- [51] Atul Puri, Tsuhan Chen (editors) "Multimedia Systems, Standards, and Networks". Marcel Dekker Inc., New York, 2000.
- [52] Rob Koenen "MPEG-4. Multimedia for Our Times". *IEEE Spectrum*, vol. 36, no. 2, pages 26-33, February 1999.
- [53] ISO/IEC SC29WG11 "The Virtual Reality Modeling Language Specification". Official Draft #3, Document no. 14772, July 1996.
- [54] Giorgio Zoia "New Audio Applications for Multimedia and MPEG-4: Complexity and Hardware". Proceedings of the European Conference on Multimedia Applications, Services and Techniques - ECMAST '98 - pages 518-530. Springer, Berlin, Germany, May 1998.
- [55] ISO/IEC JTC1/SC29/WG11 "Information Technology - Coding of Audio-Visual objects. Part 1: Systems". MPEG-4 Systems International Standard, document MPEG98/N2501.
- [56] Laurent Le Bourhis, Giorgio Zoia, Marco Mattavelli, Daniel J. Mlynek "An efficient Host/Co-Processor Solution for MPEG-4 Audio Composition". Digest of technical papers of the International Conference on Consumer Electronics - ICCE '99 - pages 26-27. Los Angeles, California, June 1999.
- [57] Laurent Le Bourhis, Giorgio Zoia, Marco Mattavelli, Daniel J. Mlynek "An efficient Host/Co-Processor Solution for MPEG-4 Audio Composition". *IEEE Transactions on Consumer Electronics*, vol.45, no.4 - pages 1290-1300. November 1999.
- [58] Eric D. Scheirer, Riitta Väänänen and Jyri Huopaniemi "AudioBIFS: Describing audio Scenes with the MPEG-4 Multimedia Standard". *IEEE Transactions on Multimedia*, vol. 1, no. 3, pages 237- 250, 1999.
- [59] Tapio Takala et al. "An Integrated System for Virtual Audio Reality". 100th AES Convention of the Acoustic Engineering Society, Copenhagen, May 1996.
- [60] Jean-Marc Jot, Olivier Warusfel "Spat: A Spatial Processor for Musicians and Sound Engineers". Proceedings of the International Conference on Acoustics and Musical Research, CIARM '95, Ferrara, Italy, May 1995.
- [61] Roger Moog "MIDI: Musical Instrument Digital Interface". *Journal of the Audio Engineering Society*, pages 394-404, May 1986.
- [62] The MIDI Manufacturer Association "The Complete Detailed MIDI 1.0 Specification". Available by order at the MMA web site <http://www.midi.org>, 1996.

- [63] Curtis Roads "The computer music tutorial". Part II, Sound Synthesis. MIT Press, Cambridge, U.S.A., 1996.
- [64] Barry Vercoe, William Gardner, Eric Scheirer "Structured Audio: Creation, Transmission and Rendering of Parametric Sound Representations". Proceedings of the IEEE, vol. 86, no. 5, May 1998.
- [65] G. E. Garnett "Music, Signals and Representations: A Survey". In Representations of Musical Signals, G. De Poli, A. Piccialli, C. Roads Editors, pages 325-370, MIT Press, Cambridge, U.S.A., 1991.
- [66] Ted Painter, Andreas Spaniar "Perceptual Coding of Digital Audio". Proceedings of the IEEE, vol. 88, no. 4, April 2000.
- [67] Max V. Matthews "The Technology of Computer Music", MIT Press, Cambridge, U.S.A., 1969.
- [68] Steven T. Pope "Machine Tongues XV: Three Packages for Software Sound Synthesis". Computer Music Journal, vol. 17, no. 2, pages 23-54. MIT Press, 1993.
- [69] Center for Computer Research in Music and Acoustics (CCRMA), Stanford University "Music Kit Documentation". Available via Internet download at the CCRMA web site (<http://ccrma-www.stanford.edu/>), Stanford, U.S.A., 1993.
- [70] Eric D. Scheirer "SAOL: the MPEG-4 Structured Audio Orchestra Language". Proceedings of the 1998 International Computer Music Conference, Ann Arbor, U.S.A., October 1998.
- [71] ISO/IEC JTC1/SC29/WG11 "Information Technology - Coding of Audio-Visual objects. Part 3: Audio. Subpart 5: Structured Audio". MPEG-4 Audio International Standard, document MPEG98/N2503-sub5.
- [72] Barry Vercoe "CSound: a Manual for the Audio Processing System". MIT Media Laboratory, Cambridge, U.S.A., 1993.
- [73] Eric D. Scheirer, Youngmoo E. Kim "Generalized Audio Coding with MPEG-4 Structured Audio". Proceedings of the Audio Engineering Society 17th International Conference on High-Quality Audio Coding, Florence, Italy, September 1999
- [74] Karlheinz Brandenburg et al. "ISO/IEC MPEG-2 advanced audio coding". Journal of the Audio Engineering Society, vol. 45, no. 10, pages 789-814, October 1997.
- [75] R. Giladi and N. Ahituv "SPEC as a performance evaluation measure". IEEE Computer, vol. 28, no. 8, pages 33-42, August 1995.
- [76] Standard Performance Evaluation Corporation. "SPECcpu95". Warrenton USA 1995. (<http://www.spec.org>).
- [77] Standard Performance Evaluation Corporation. "SPECcpu2k". Warrenton USA 2000. (<http://www.spec.org>).
- [78] Ulf Schünemann "Architecture of Programming Languages". Computer Science Department homepage, Memorial University, Canada, <http://www.cs.mun.ca/~ulf>

- [79] Bjarne Stroustrup "What is Object-Oriented Programming". Internal Report, AT&T Bell Laboratories, Murray Hill, U.S.A., 1991.
- [80] John Lazzaro and John Wavrzynek "MPEG-4 Structured Audio: developer tools". CS Division, UC Berkeley, <http://www.cs.berkeley.edu/~lazzaro/sa/index.html>.
- [81] Giorgio Zoia, Claudio Alberti "An Efficient Block-Based Interpreter for MPEG-4 Structured Audio". Proceedings of the IEEE International Symposium on Circuits and Systems - ISCAS 2000, Geneva, Switzerland, May 2000.
- [82] ISO/IEC JTC1/SC29/WG11 "Information Technology - Coding of Audio-Visual objects. Part 4: Conformance. Subpart 3: Audio Conformance". MPEG-4 Conformance International Standard, document MPEG99/N3067-sub3.
- [83] Giorgio Zoia "A method for complexity measurements in Structured Audio". Contribution MPEG98/M3602, Dublin, Ireland, July 1998.
- [84] Giorgio Zoia "Complexity measurement tool for level definitions of Algorithmic synthesis and AudioFX object type". Contribution MPEG99/M5031, Melbourne, Australia, October 1999.
- [85] Giorgio Zoia, Claudio Alberti "A Virtual DSP Architecture for MPEG-4 Structured Audio". Proceedings of the COST-G6 Conference on Digital Audio Effects - DAFX'00, Verona, Italy, December 2000.
- [86] Durand R. Begault, "3-D Sound for Virtual Reality and Multimedia". Academic Press, Boston, U.S.A., 1994.
- [87] Jens Blauert, "Spatial hearing. The Psychophysics of Human Sound Localization". MIT Press, Cambridge, U.S.A., 1983.
- [88] Phil Lapsley, J. Bier, A. Shoham, E.A. Lee "DSP processor fundamentals". IEEE Press, New York, 1997.
- [89] MIT Media Laboratory - MPEG-4 SA Homepage - <http://sound.media.mit.edu/mpeg4>.
- [90] M.R. Schroeder "Natural sounding artificial reverberation". Journal of the Audio Engineering Society, vol. 10, no. 3, 1962.
- [91] William G. Gardner, "Efficient convolution without input/output delay". Proceedings of the 97th AES Convention, San Francisco, U.S.A., 1994.
- [92] Giorgio Zoia "A possible Structured Audio subset for the AudioFX node". Contribution MPEG98/M4012, Atlantic City, U.S.A., October 1998.
- [93] Giorgio Zoia, L.LeBourhis, U.Horbach, A.Karamustafaoglu "Proposed revision of Systems and Audio profiles and levels from an analysis of audio composition". Contribution MPEG98/M3604, Dublin, Ireland, July 1998.
- [94] Giorgio Zoia "Level definition examples for Structured Audio main profile". Contribution MPEG98/M4110, Atlantic City, U.S.A., October 1998.
- [95] Richard B. Dannenberg, N. Thompson "Real-Time Software Synthesis on Superscalar Architectures". Computer Music Journal, vol. 21, no.3, pages 83-94, MIT Press 1997.

- [96] M. Tremblay, D. Greenley, K. Normoyle "The design of the microarchitecture UltraSPARC-I". Proceedings of the IEEE, vol. 83, no. 12, pages 1653-1663, December 1995.
- [97] Roger Espasa, Mateo Valero "Exploiting Instruction- and Data-Level Parallelism". IEEE Micro, vol. 17, no. 5, September/October 1997, pages 20-27, 1997.
- [98] Tim Lindholm, Frank Yellin "The JAVA Virtual Machine Specification". 2nd Edition (JAVA series), Addison Wesley, 1999.
- [99] Jon Meyer, Troy Downing "JAVA Virtual Machine". O'Reilly & Associates, 1st edition, March 1997
- [100] John R. Levine, Tony Mason, Doug Brown "Lex and Yacc". O'Reilly & Associates, 2nd revised edition, October 1992.
- [101] Ronald Mak "Writing Compilers and Interpreters. An applied approach". John Wiley and Sons, New York 1991.
- [102] Gerald J. Popek, Robert Goldberg "Formal Requirements for Virtualizable Third Generation Architectures". Communications of the ACM, vol.17, no. 7, July 1974.
- [103] Judith Hall, Paul Robinson "Virtualizing the VAX Architecture" Proceedings of the 18th Annual International Symposium on Computer Architecture, pages 380-389, Toronto, Canada, May 1991.
- [104] Matthias Rosenthal "Concept and Design of a Reconfigurable Parallel Processing System for Digital Audio", PhD dissertation No. 11936, ETHZ, Zürich, 1997.
- [105] Jesus Corbal, Roger Espasa "Adding a Vector Unit to a Superscalar Processor". 1999 International Conference on Supercomputing, Rhodes, Greece, June 1999.
- [106] Optivec vectorial libraries. Available on the Internet at <http://www.optivec.com>
- [107] Chao-Ju Hou, Kang Shin "Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed Real-Time Systems". IEEE Transactions on Computers, vol. 46, no. 12, December 1997.
- [108] Sanjoy K. Baruah, Jayant R. Haritsa "Scheduling for Overload in Real-time Systems", IEEE Transactions on Computers, vol. 46, no. 9, September 1997.
- [109] J. Sandvad, D.Hammershoi, "Binaural auralization. Comparison of FIR and IIR filter in representation of HIRs". Proceedings of the 96th AES Convention, Amsterdam, The Netherlands, 1994.
- [110] Jean-Marc Jot, Vincent Larcher, Olivier Warusfel, "Digital signal processing issues in the context of binaural and transaural stereophony". Proceedings of the 98th AES Convention, Paris, France, 1995.
- [111] Ulrich Horbach "Implementation of Audio Compositing Functions: Algorithms and Hardware requirements". Contribution MPEG97/M1931, Bristol, U.K., April 1997.
- [112] Elizabeth Cohen, John Eargle "Audio in a 5.1 Channel Environment". Proceedings of the 99th AES Convention, New York, U.S.A., 1995.

- [113] Roger Dressler "A Step Toward Improved Surround Sound: Making the 5.1-channel Format a Reality". Proceedings of the 1996 Audio Engineering Society Workshop, Copenhagen, Denmark, May 1996.
- [114] J. S. Bamford, J. Vanderkooy "Ambisonic Sound for Us". Proceedings of the 99th AES Convention, New York, U.S.A., 1995.
- [115] M. A. Gerzon "Ambisonics in Multichannel Broadcasting and Video". Journal of the Audio Engineering Society, vol. 33, pp. 859-871, November 1995.
- [116] C. J. McCabe, D. J. Furlong "Virtual Imaging Capabilities of Surround Sound Systems". Proceedings of the 93rd AES Convention, San Francisco, U.S.A., 1992.
- [117] Ulrich Horbach "New Techniques for the Production of Multichannel Sound" Proceedings of the 103rd AES Convention, New York, U.S.A., 1997.
- [118] W. G. Gardner "Reverberation algorithms". in Applications of Digital Signal Processing to Audio and Acoustics, M. Kahrs and K. Brandenburg, Eds., pages 85-132, Kluwer Academic, 1998.
- [119] Jean-Marc Jot "Efficient models for reverberation and distance rendering in computer music and virtual audio reality". Proceedings of the International Computer Music Conference, pages 236-243, Thessaloniki, Greece, 1997.
- [120] A.J. Berkhout. "A holographic approach to acoustic control". Journal of the Audio Engineering Society, vol. 36, pages 977-995, 1988.
- [121] A.J. Berkhout, D. de Vries, P. Vogel "Acoustic Control by Wave Field Synthesis". Journal of the Acoustic Society, vol. 93, pages 2764-2778, 1993.
- [122] M. Boone, E. Verheijen, G. Jansen "Virtual Reality by Sound Reproduction Based on Wave Field Synthesis". Proceedings of the 100th AES Convention, Copenhagen, Denmark, 1996.
- [123] Ulrich Horbach et al. "Numerical Simulation of Wave Field Synthesis Created by Loudspeaker Arrays". Proceedings of the 107th AES Convention, New York, U.S.A., 1999.
- [124] Ulrich Horbach, Marinus Boone "Future Transmission and Rendering Formats for Multichannel Sound". Proceedings of the AES 16th International Conference on Spatial Sound Reproduction, Rovaniemi, Finland, 1999.
- [125] Ulrich Horbach and Attila Karamustafaoglu "Practical Implementation of a Data-Based Wave Field Reproduction System". Proceedings of the 108th AES Convention, Paris, France, 2000.
- [126] CreamWare "Pulsar - DSP Powered Music Production Environment". Available at the Internet address: <http://www.creamware.de/Products>
- [127] Gerritt Slavenburg et al. "TM1100 Preliminary Data Book". Philips Electronics NAC, December 1998.

- [128] K. Ebcioglu et al. "VLIW compilation techniques in a superscalar environment". Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation,
- [129] Giorgio Zoia, Claudio Alberti "An Audio Virtual DSP for Multimedia Frameworks". Proceedings of the International Conference on Acoustics, Speech and Sound Processing, ICASSP2001, Salt Lake City, U.S.A., May 2001.
- [130] Marco Mattavelli, Sylvain Brunetton "Implementing Real-time Video Decoding on Multimedia Processors by Complexity Prediction Techniques". IEEE Transactions on Consumer Electronics, vol.44, no.3 - pages 760-767, August 1998.
- [131] David Solomon "Inside Windows NT". Microsoft Press, second edition, May 1998.
- [132] David Kruglinski, George Shepherd and Scott Wingo "Programming with Microsoft Visual C++". Fifth edition, Microsoft Press, U.S.A. 1998.

Integrated Systems Laboratory
DE-LSI / EPFL
1015 LAUSANNE Switzerland

Giorgio Zoia

Phone: +41 21 693 6979
Fax: +41 21 693 4663
E-mail: Giorgio.Zoia@epfl.ch

Education and professional experience

Giorgio Zoia was born in Arluno, not far from Milano, Italy, on the 21st of September 1968.

He received his Diploma of Electronic Engineering ("Laurea in Ingegneria Elettronica") from the Politecnico di Milano in April 1996. Preparing his diploma final thesis, from May to November 1995 he was a visiting student at the Signal Processing Laboratory of the Swiss Federal Institute of Technology (EPFL) in Lausanne, Switzerland. There he worked on the design and optimization of the motion estimation algorithm for an MPEG-2 encoder, based on evolutionary algorithms.

In June 1996 he joined the Integrated System Laboratory, where his first activity was the development of a PCI interface architecture for the PUMA audio chip, a collaboration with Studer Professional Audio AG.

In September 1996 he started working in the PUMA group; his main activities have been digital design and CAD synthesis optimization in submicron technology.

From summer 1997 until fall 1998, while starting his PhD, he was involved in the European project EMPHASIS, with specific tasks in MPEG-4 synthetic and 3-D audio. Since then he is still actively collaborating to the MPEG-4 standardization process, to which he submitted several contributions concerning Structured Audio, Audio composition (Systems) and analysis of computational complexity.

In 1999 he began to work on efficient solutions for the implementation of MPEG-4 Structured Audio and AudioBIFS decoders. He was active promoter of the ThreeDSPACE project, where he is greatly developing his skill in writing compilers, in design of virtual architectures and in fast execution engines for digital Audio.

From January to April 1999 he attended at the EPFL the Postgrade Course on "Intelligent Interfaces & Systems".

In 2000 he was again an active promoter of the CARROUSO project, an important effort in the IST fifth framework recently approved for funding. CARROUSO aims at developing a complete MPEG-4 encoding/transmission/decoding-and-rendering chain for advanced 3-d Audio environments.

Throughout the last two years he participated to the coordination of the Audio activities of his Laboratory, building an experience in submission and management of national and international research projects.

Knowledge

- Research: Audio and Video compression, VLSI design, compilers, computer simulation, computer music, 3-D Audio processing.
- Systems: UNIX, Linux, Windows98, WindowsNT.
- Programming: C, C++, SAOL, HTML (major skills)
Java, Pascal, Csound (basic skills)

Memberships and Service in Professional Organizations

IEEE Student Member, Swiss National Body Delegate in MPEG.

Languages

Italian: native
English: fluent written and spoken
French: fluent written and spoken

Extra curriculum activities

In 1994 he worked in social activities as conscience objector.

Playing piano since 1978 (eight years of courses)

Playing pipe organ since 1985 (currently at the Italian parish in Lausanne)

Played in a basketball team for 10 years; currently trekker and climber as a hobby.

Publications

Marco Mattavelli, Giorgio Zoia

"Vector Tracing Techniques for Motion Estimation Algorithms in Video Coding", Proceedings of the EUSIPCO-98 Conference vol. IV pages 2097-2100. Island of Rhodes, Greece, September 1998.

Marco Mattavelli, Giorgio Zoia

"An Efficient Motion Estimation Algorithm Based on Tracing Techniques on Large Search Windows", Proceedings of the IEEE International Conference on Image Processing - ICIP '98, vol. III. Chicago, Illinois, October 1998.

Giorgio Zoia

"New Audio Applications for Multimedia and MPEG-4: Complexity and Hardware", Multimedia Applications, Services and Techniques - ECMAST '98 - pages 518-530 Springer. Berlin, Germany, May 1998 .

Laurent Le Bourhis, Giorgio Zoia, Marco Mattavelli, Daniel J. Mlynek
"An efficient Host/Co-Processor Solution for MPEG-4 Audio Composition", Digest of technical papers of the International Conference on Consumer Electronics - ICCE '99, pages 26-27. Los Angeles, California, June 1999.

Laurent Le Bourhis, Giorgio Zoia, Marco Mattavelli, Daniel J. Mlynek
"An efficient Host/Co-Processor Solution for MPEG-4 Audio Composition", IEEE Transactions on Consumer Electronics, Vol.45, No.4, November 1999, pages 1290-1300.

Giorgio Zoia, Claudio Alberti
"An Efficient Block-Based Interpreter for MPEG-4 Structured Audio", Proceedings of the IEEE International Symposium on Circuits and Systems - ISCAS 2000, Geneva, Switzerland, May 2000.

Marco Mattavelli, Giorgio Zoia
"Vector Tracing Algorithms for Motion Estimation in Large Search Windows", IEEE Transactions on Circuits and Systems for Video Technology, Vol .10, No. 8, pages 1426-1438, December 2000.

Giorgio Zoia, Claudio Alberti
" A Virtual DSP Architecture for MPEG-4 Structured Audio ", Proceedings of the COST-G6 Conference on Digital Audio Effects – DAFX'00, Verona, Italy, December 2000.

Giorgio Zoia, Claudio Alberti
"An Audio Virtual DSP for Multimedia Frameworks", Proceedings of the IEEE International Conference on Audio, Sound and Speech Processing - ICASSP 2001, Salt Lake City, U.S.A., May 2001.

Giorgio Zoia, Claudio Alberti
"An MPEG-oriented platform for Wave Field Synthesis arrays of loudspeakers", Proceedings of the AES 19th International Conference on Surround Sound, Schloss Elmau, Germany, June 2001.

More than 15 contributions, mainly accepted, to the MPEG-4 standardization process from 1997 to 2000.