Koen Bertels *Editor*

Hardware/Software Co-design for Heterogeneous Multi-core Platforms

The hArtes Toolchain

Hardware/Software Co-design for Heterogeneous Multi-core Platforms

Koen Bertels
Editor

Hardware/Software Co-design for Heterogeneous Multi-core Platforms

The hArtes Toolchain

Springer

Editor
Koen Bertels
Fac. Electrical Engineering
Mathematics & Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands
k.l.m.bertels@tudelft.nl

ISBN 978-94-007-1405-2 e-ISBN 978-94-007-1406-9
DOI 10.1007/978-94-007-1406-9
Springer Dordrecht Heidelberg London New York

Library of Congress Control Number: 2011941860

© Springer Science+Business Media B.V. 2012
No part of this work may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, microfilming, recording or otherwise, without written permission from the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work.

Cover design: VTeX UAB, Lithuania

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The hArtes project[1] was started as an innovative European project (funded by European Union) aiming at laying the foundations of a new holistic approach for the design of complex and heterogeneous embedded solutions (hardware and software), from the concept to the silicon (or B2B, from the brain to bits). The hArtes stands for "holistic Approach to reconfigurable real time embedded systems". As defined in the Embedded Systems Chapter of the IST 2005-06 Work Programme the objective of the hArtes project is to "develop the next generation of technologies, methods and tools for modeling, design, implementation and operation of hardware/software systems embedded in intelligent devices. An end-to-end systems (holistic) vision should allow building cost-efficient ambient intelligence systems with optimal performance, high confidence, reduced time to market and faster deployment".

The hArtes project aims to lay the foundation for a new holistic (end-to-end) approach for complex real-time embedded system design, with the latest algorithm exploration tools and reconfigurable hardware technologies. The proposed approach will address, for the first time, optimal and rapid design of embedded systems from high-level descriptions, targeting a combination of embedded processors, digital signal processing and reconfigurable hardware. The project ended with an important scientific and technical contribution that resulted in more than 150 international publications as well as a spin-off company, BlueBee.[2]

From the application point of view, the complexity of future multimedia devices is becoming too big to design monolithic processing platforms. This is where the hArtes approach with reconfigurable heterogeneous systems becomes vital. As a part of the project, a modular and scalable hardware platforms will be developed that can be reused and re-targeted by the tool chain to produce optimized real-time embedded products. The results obtained will be evaluated using advanced audio and video systems that support next-generation communication and entertainment facilities, such as immersive audio and mobile video processing. Innovations of the hArtes approach include: (a) support for both diagrammatic and textual formats

[1] www.hartes.org.

[2] www.bluebee-tech.com.

in algorithm description and exploration, (b) a framework that allows novel algorithms for design space exploration, which aims to automate design partitioning, task transformation, choice of data representation, and metric evaluation for both hardware and software components, (c) a system synthesis tool producing near optimal implementations that best exploits the capability of each type of processing element; for instance, dynamic reconfigurability of hardware can be exploited to support function upgrade or adaptation to operating conditions.

Delft, The Netherlands K. Bertels

Acknowledgement

This book describes the outcome of the hArtes project (IST-035143) supported by the Sixth Framework Programme of the European Community under the thematic area Embedded Systems.

Contents

1 Introduction 1
Koen Bertels

2 The hArtes Tool Chain 9
Koen Bertels, Ariano Lattanzi, Emanuele Ciavattini, Ferruccio Bettarelli, Maria Teresa Chiaradia, Raffaele Nutricato, Alberto Morea, Anna Antola, Fabrizio Ferrandi, Marco Lattuada, Christian Pilato, Donatella Sciuto, Roel J. Meeuws, Yana Yankova, Vlad Mihai Sima, Kamana Sigdel, Wayne Luk, Jose Gabriel de Figueiredo Coutinho, Yuet Ming Lam, Tim Todman, Andrea Michelotti, and Antonio Cerruto

3 The hArtes Platform 111
Georgi Kuzmanov, Raffaele Tripiccione, Giacomo Marchiori, and Immacolata Colacicco

4 Audio Array Processing for Telepresence 125
Gregor Heinrich, Fabian Logemann, Volker Hahn, Christoph Jung, Jose Gabriel de Figueiredo Coutinho, and Wayne Luk

5 In Car Audio 155
Stefania Cecchi, Lorenzo Palestini, Paolo Peretti, Andrea Primavera, Francesco Piazza, Francois Capman, Simon Thabuteau, Christophe Levy, Jean-Francois Bonastre, Ariano Lattanzi, Emanuele Ciavattini, Ferruccio Bettarelli, Romolo Toppi, Emiliano Capucci, Fabrizio Ferrandi, Marco Lattuada, Christian Pilato, Donatella Sciuto, Wayne Luk, and Jose Gabriel de Figueiredo Coutinho

6 Extensions of the hArtes Tool Chain 193
Ferruccio Bettarelli, Emanuele Ciavattini, Ariano Lattanzi, Giovanni Beltrame, Fabrizio Ferrandi, Luca Fossati, Christian Pilato, Donatella Sciuto, Roel J. Meeuws, S. Arash Ostadzadeh, Zubair Nawaz, Yi Lu, Thomas Marconi, Mojtaba Sabeghi, Vlad Mihai Sima, and Kamana Sigdel

7 Conclusion: Multi-core Processor Architectures Are Here to Stay . . 229
Koen Bertels

Contributors

Anna Antola Politecnico di Milano, Via Ponzio 34/5, 20133 Milan, Italy

Giovanni Beltrame Politecnico di Milano, Via Ponzio 34/5, 20133 Milan, Italy

Koen Bertels Fac. Electrical Engineering, Mathematics & Computer Science, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands

Ferruccio Bettarelli Leaff Engineering, Via Puccini 75, 62016 Porto Potenza Picena, MC, Italy

Jean-Francois Bonastre Université d'Avignon et des Pays de Vaucluse, 339 Chemin des Meinajaries, 84911 Avignon, France

Francois Capman Thales Communications, 146 Bd de Valmy, 92704 Colombes, France

Emiliano Capucci Faital Spa, Via B. Buozzi 12, 20097 San Donato Milanese, MI, Italy

Stefania Cecchi DIBET—Universitá Politecnica delle Marche, Via Brecce Bianche 1, 60131 Ancona, Italy

Antonio Cerruto Atmel Corp., Milan, Italy

Maria Teresa Chiaradia Politecnico di Bari, Bari, Italy

Emanuele Ciavattini Leaff Engineering, Via Puccini 75, 62016 Porto Potenza Picena, MC, Italy

Immacolata Colacicco Università di Ferrara, via Saragat 1, 44100 Ferrara, Italy

Fabrizio Ferrandi Politecnico di Milano, Via Ponzio 34/5, 20133 Milan, Italy

Jose Gabriel de Figueiredo Coutinho Imperial College, 180 Queen's Gate, London SW72A2, UK

Luca Fossati Politecnico di Milano, Via Ponzio 34/5, 20133 Milan, Italy

Volker Hahn Fraunhofer IGD, Fraunhoferstraße 5, 64283 Darmstadt, Germany

Gregor Heinrich vsonix GmbH, Fraunhoferstraße 5, 64283 Darmstadt, Germany; Fraunhofer IGD, Fraunhoferstraße 5, 64283 Darmstadt, Germany

Christoph Jung Fraunhofer IGD, Fraunhoferstraße 5, 64283 Darmstadt, Germany

Georgi Kuzmanov TU Delft, Delft, The Netherlands

Ariano Lattanzi Leaff Engineering, Via Puccini 75, 62016 Porto Potenza Picena, MC, Italy

Marco Lattuada Politecnico di Milano, Via Ponzio 34/5, 20133 Milan, Italy

Christophe Levy Université d'Avignon et des Pays de Vaucluse, 339 Chemin des Meinajaries, 84911 Avignon, France

Fabian Logemann Fraunhofer IGD, Fraunhoferstraße 5, 64283 Darmstadt, Germany

Yi Lu TU Delft, Delft, The Netherlands

Wayne Luk Imperial College, 180 Queen's Gate, London SW72A2, UK

Giacomo Marchiori Università di Ferrara, via Saragat 1, 44100 Ferrara, Italy

Thomas Marconi TU Delft, Delft, The Netherlands

Roel J. Meeuws TU Delft, Delft, The Netherlands

Andrea Michelotti Atmel Corp., Milan, Italy

Yuet Ming Lam Imperial College, 180 Queen's Gate, London SW72A2, UK

Alberto Morea Politecnico di Bari, Bari, Italy

Zubair Nawaz TU Delft, Delft, The Netherlands

Raffaele Nutricato Politecnico di Bari, Bari, Italy

S. Arash Ostadzadeh TU Delft, Delft, The Netherlands

Lorenzo Palestini DIBET—Universitá Politecnica delle Marche, Via Brecce Bianche 1, 60131 Ancona, Italy

Paolo Peretti DIBET—Universitá Politecnica delle Marche, Via Brecce Bianche 1, 60131 Ancona, Italy

Francesco Piazza DIBET—Universitá Politecnica delle Marche, Via Brecce Bianche 1, 60131 Ancona, Italy

Christian Pilato Politecnico di Milano, Via Ponzio 34/5, 20133 Milan, Italy

Andrea Primavera DIBET—Universitá Politecnica delle Marche, Via Brecce Bianche 1, 60131 Ancona, Italy

Mojtaba Sabeghi TU Delft, Delft, The Netherlands

Donatella Sciuto Politecnico di Milano, Via Ponzio 34/5, 20133 Milan, Italy

Kamana Sigdel TU Delft, Delft, The Netherlands

Vlad Mihai Sima TU Delft, Delft, The Netherlands

Simon Thabuteau Thales Communications, 146 Bd de Valmy, 92704 Colombes, France

Tim Todman Imperial College, 180 Queen's Gate, London SW72A2, UK

Romolo Toppi Faital Spa, Via B. Buozzi 12, 20097 San Donato Milanese, MI, Italy

Raffaele Tripiccione Università di Ferrara, via Saragat 1, 44100 Ferrara, Italy

Yana Yankova TU Delft, Delft, The Netherlands

List of Figures

Fig. 1.1 Growth of the ES industry (EU report) 2
Fig. 1.2 Proportion of software cost in total cost of end product (EU report) 3
Fig. 1.3 Market for multi-core CPU's in ES 3
Fig. 1.4 The hArtes tool chain 5
Fig. 1.5 Mapping Code on various platforms 6
Fig. 2.1 The hArtes toolchain 11
Fig. 2.2 NU-Tech graphical user interface 36
Fig. 2.3 NU-Tech RealTime Watch 37
Fig. 2.4 NU-Tech Profiling information 38
Fig. 2.5 GAETool 39
Fig. 2.6 GAETool Eclipse Wizard 40
Fig. 2.7 Hgprof Tool Flow 42
Fig. 2.8 Interaction of tools, scripts, and intermediate results in the Quipu modeling approach 44
Fig. 2.9 Predicted versus measured number of flip-flops on a Virtex-4 LX200 using the Quipu prediction model. These are validated predictions using leave-one-out validation 47
Fig. 2.10 PandA framework 50
Fig. 2.11 PandA automatic parallelization 51
Fig. 2.12 Example of program in original format (**a**) and in the corresponding SSA-form (**b**) 52
Fig. 2.13 Example of Control Flow Graph of basic blocks 54
Fig. 2.14 Control dependence graph (**a**), data dependence graph (**b**), anti-dependence graph (**c**) 57
Fig. 2.15 Program dependence graph 59
Fig. 2.16 Nodes belonging to different loops are clustered in different subgraphs 60
Fig. 2.17 Phases of the parallelism extraction algorithm 61
Fig. 2.18 A task graph compliant with the fork/join programming model (on the *left*) and one not compliant 62
Fig. 2.19 Fork (**a**) and Join (**b**) node sharing 65

Fig. 2.20 Parallelization of JPEG encoder 66
Fig. 2.21 Parallelization of DRM decoder 67
Fig. 2.22 Validation of the FPGA area model 70
Fig. 2.23 Distributions of the benchmarks dataset with respect to the number of instructions and the number of cycles 73
Fig. 2.24 Example of unprofitable parallelism 73
Fig. 2.25 The task mapping approach is comprised by four main modules: (1) an automatic C guidelines verification process that automatically determines if the code conforms to the restrictions of the task mapping process, (2) a filtering engine that determines which processing elements can support each task, (3) a task transformation engine which produces one or more implementations of a given task that can potentially exploit the capabilities of the processing elements in the system, and (4) the mapping selection process which assigns a processing element implementation to each task of the application 76
Fig. 2.26 An overview of the task transformation engine. The task transformation engine receives as input a task and additional parameters such as the processing element that we wish to target, and generates a set of implementations. The set of transformations to be applied to each processing element is provided by the user. The implementations of transformations are stored as shared libraries for ROSE transformations, and as text files for CML-based transformations. A CML description consists of three sections: the pattern to match, the matching conditions, and the resulting pattern (Listing 2.1) 79
Fig. 2.27 Starting with the original code for the application that models the vibration of a guitar string, we explore ways of using seven different transformations to attempt to improve the run time and memory usage. Much of the speedup comes from simplifying the code and making iteration variables integer, while the remainder comes from caching to prevent repeat memory access and removing a constant assignment from the loop body. The caching also enables one array to be eliminated (about 33% reduction in memory usage), possibly at the expense of performance 81
Fig. 2.28 An overview of the mapping selection process 82
Fig. 2.29 Searching for the best mapping and scheduling solution using multiple neighborhood functions. The *solid arrows* show the moves generated by different neighborhood functions. The *dotted arrows* denote the best move in each iteration of the search. PE: processing element, tk: task 82
Fig. 2.30 A description of C guidelines automatically detected by hArmonic, and a screenshot of its graphical user-interface 85
Fig. 2.31 Example of MOLEN 89
Fig. 2.32 Molen without parallelism 91

Fig. 2.33 Molen with parallelism expressed with OpenMP syntax 91
Fig. 2.34 The DWARV toolset 92
Fig. 2.35 Representation of the linking process in the hArtes toolchain . . . 99
Fig. 2.36 Hartes framework 101
Fig. 3.1 The Molen machine organization 113
Fig. 3.2 The Molen programming paradigm 114
Fig. 3.3 Top level architectural structure of the hArtes Hardware Platform. The system has two independent heterogeneous and configurable processors that communicate among each other and with an audio I/O subsystems; the latter supports several ADAT channels . 117
Fig. 3.4 Detailed block diagram of the Basic Configurable Element (BCE) of the hArtes Hardware Platform. The BCE is the basic building block of the platform, supporting several processing architectures. One or more BCEs work in parallel to support an hArtes application 118
Fig. 3.5 Picture of the hArtes Hardware Platform. The two BCEs use two daughter boards each, one for the D940HF processor and one for the FPGA based infrastructure. These daughter-boards are at the center of the mother board. The ADAT interfaces and several standard I/O connectors are clearly visible at the *top* and at the *bottom* of the picture, respectively 120
Fig. 4.1 Prototype setup used for the telepresence application 127
Fig. 4.2 Telepresence scenario: functional overview 128
Fig. 4.3 Multi-channel beamforming: Listening in two directions 129
Fig. 4.4 Wave-field synthesis: Matching the original and synthesized fields 130
Fig. 4.5 Generic structure of the array processing algorithms 131
Fig. 4.6 Structure of the beamforming algorithm 132
Fig. 4.7 Structure of the wave-field synthesis algorithm 133
Fig. 4.8 Overlap-Save fast convolution for a filter block $H(z)$. *Square brackets* indicate frame indices, interleaved signals are denoted x_{12} etc., sizes of data blocks are to scale 135
Fig. 4.9 Multitap ring buffer with wrap-around output for a delay line $D(z) = z^{-M}$ 136
Fig. 4.10 Main loop C-language structure of the generic algorithm in Fig. 4.5 137
Fig. 4.11 Implementations of audio array processing algorithms 139
Fig. 4.12 Exemplary benchmark of memory transfer (hHP DMA read from shared memory to DSP-internal memory) 141
Fig. 4.13 Embedded platform benchmark based on the generic array processing algorithm in Fig. 4.5. *Left*: optimal path (D-MEM represents shared memory, FPGA used for hHP audio I/O), *top-right*: memory transfers on DEB and hHP (linear coefficients: DSP cycles at 80MHz $= a + Nb$). *Bottom-right*:

Performance figures for filter operation (cycles vs. frame-length, internal/external overlap refers to storing the previous OLS frame in DSP internal memory or shared memory) and multiply-add (cycles $= a + Nb$) . . . 142
Fig. 4.14 Main results of the hArmonic tool in combination with the synthesis toolbox and the hHP: (**a**) beamformer and (**b**) wave-field synthesis . . . 149
Fig. 4.15 A description of some of the hArtes constraints automatically derived by the hArmonic tool to ensure that the mapping solutions are feasible on the hArtes platform. This figure shows part of the log generated by hArmonic when processing the beamformer application. A total of 677 mapping constraints are identified for this application . . . 150
Fig. 5.1 Enhanced listening experience algorithms . . . 162
Fig. 5.2 General approach for speech recognition . . . 166
Fig. 5.3 General approach for speaker recognition . . . 167
Fig. 5.4 NU-Tech Framework . . . 168
Fig. 5.5 NU-Tech integration of single-channel speech enhancement . . . 169
Fig. 5.6 Effects of partitioning and mapping on the kernel application . . . 170
Fig. 5.7 Main results of the hArmonic tool in combination with the synthesis toolbox and the hArtes hardware platform: (**a**) Audio Enhancement application and (**b**) Stationary Noise Filter . . . 172
Fig. 5.8 Main features of the code generated by the hArmonic tool for each backend compiler in the synthesis toolbox. In this figure, we focus on the code generated for the ARM processor . . . 173
Fig. 5.9 PowerServer final setup in the car trunk . . . 174
Fig. 5.10 Loudspeakers position . . . 175
Fig. 5.11 Example of a midrange loudspeaker . . . 175
Fig. 5.12 Effect of adaptive stereo equalization on car . . . 177
Fig. 5.13 Smoothed car impulse response prototype and equalization weights during adaptive equalization . . . 178
Fig. 5.14 Stereophonic Echo Cancellation of a music sequence . . . 179
Fig. 5.15 Finally, the multi-channel speech enhancement module has been implemented in NU-Tech, using the SRP-PHAT criterion for on-line speaker localization . . . 179
Fig. 5.16 Limited search using region-constrained estimation of SRP-PHAT criterion . . . 180
Fig. 5.17 Scheme of the Power Server System considering the hartes platform 181
Fig. 5.18 Scheme of the overall Audio Enhancement In-Car application developed for the hHp platform . . . 183
Fig. 5.19 In Car application performances considering different approaches 184
Fig. 5.20 Workload of the overall Audio Enhancement system considering a parallelization of the applications . . . 185
Fig. 5.21 (**a**) Devices workload and (**b**) Devices memory requirements for the Audio Enhancement system in case of parallelized applications 185

Fig. 5.22 In-Car communication performances in terms of time elapseds . . 188
Fig. 6.1 Run-time environment 194
Fig. 6.2 A sample CCG 195
Fig. 6.3 Kernels and memories 197
Fig. 6.4 AMMA postcode 199
Fig. 6.5 Memory access 201
Fig. 6.6 The Molen structure with interrupt support 202
Fig. 6.7 Simple general operation of Molen 203
Fig. 6.8 An example of runtime partial reconfiguration instructions 204
Fig. 6.9 Integration of the ICAP controller to Molen 205
Fig. 6.10 Models 206
Fig. 6.11 Simple example 210
Fig. 6.12 Generic expression and its pipeline circuit for example in Fig. 6.11 210
Fig. 6.13 Graph to show speedup and area overhead w.r.t. dataflow 212
Fig. 6.14 Application profiling and system level architecture exploration in the context of hartes toolchain 213
Fig. 6.15 rSesame framework 215
Fig. 6.16 Dynamic profiling famework overview 216
Fig. 6.17 Sample profiling elements stored by QUAD in an architecture description file 217
Fig. 6.18 The architecture of the ReSP system simulation platform 219
Fig. 6.19 Organization of the simulated environment including the Routine-Emulation module 221
Fig. 6.20 Internal structure and working mechanisms of the function routine emulator 222
Fig. 6.21 Average execution time on all the 1344 runs for 8 benchmarks and for all the tested number of processors 223
Fig. 6.22 Execution Time in front of different System Call latencies: 2 PE 2 threads 223
Fig. 6.23 Execution Time in front of different System Call latencies: 4 PE 4 threads 223
Fig. 6.24 Execution Time in front of different System Call latencies: 8 PE 8 threads 224
Fig. 6.25 Execution Time in front of different System Call latencies: 16 PE 16 threads 224
Fig. 7.1 Persistence of legacy platforms and applications 230

List of Tables

Table 2.1 Number of functions in each domain with the main algorithmic characteristics present in each application domain 46
Table 2.2 Comparison of results of Ant Colony Optimization (ACO), Simulated Annealing (SA) and the Tabu Search (TS), along with a dynamic policy 64
Table 2.3 Speed-ups for the parallel versions of JPEG and DRM applications as estimated by Zebu and measured on SimIt-ARM and ReSP 68
Table 2.4 Performance estimation error on single processing element . . . 72
Table 2.5 Average and Maximum Speed-up Estimation Error on whole task graph compared with techniques based on Worst Case and Average Case 74
Table 2.6 List of filters employed to find mappings 78
Table 2.7 Evaluation of the main hArtes applications 88
Table 2.8 C language support—data types 95
Table 2.9 C language support—storage 95
Table 2.10 C language support—expressions 95
Table 2.11 C language support—statements 96
Table 2.12 hArtes library components specifications 98
Table 2.13 hArtes library components specifications 98
Table 4.1 Reference implementations of the BF and WFS algorithms . . . 140
Table 4.2 Parameters of hArtes implementations of the BF and WFS algorithms 143
Table 4.3 Evaluation of the automatic mapping approach on the beamformer and wave-field synthesis applications. The hArtes constraints correspond to the number of automatically derived constraints to make the mapping solution compatible with the hArtes platform. The speedup achieved corresponds to the ratio between the performance of the application running on the ARM and the mapping solution produced by the hArmonic tool . 148
Table 4.4 Beamformer application performance (parameters see Table 4.2) 151

Table 4.5 WFS application performance (parameters see Table 4.2) 151
Table 5.1 Evaluation of the automatic mapping approach on the Audio Enhancement application and related kernels (Xfir, PEQ, FracShift, Octave), and the Stationary Noise Filter. The speedup corresponds to the ratio between the performance of the application running on the ARM and the mapping solution produced by the hArmonic tool 171
Table 5.2 Average execution time of direct and inverse 1024-pts complex FFT for different optimization levels 176
Table 5.3 Performances of the martes and DSE approach in terms of workload for each algorithm with relation to the direct implementation 183
Table 6.1 Memory block scores (in ms) 199
Table 6.2 Execution overheads of tracking the memory 200
Table 6.3 Status of CCU and the microcode 201
Table 6.4 Time and hardware to compute DCT 211

Chapter 1
Introduction

Koen Bertels

The following text fragment is taken from the original proposal becoming ultimately the accepted description of work that constituted the basis for the 3-year technical work that was performed. We leave it up to the reader of this book to assess whether the reviewers of the hArtes project were right in judging that we have completely achieved the stated objectives of the project. The entire consortium that has worked on this project is convinced that an important scientific and technical contribution was made resulting in more than 150 international publications as well as a spin off company, BlueBee.[1]

hArtes aims to lay the foundation for a new holistic (end-to-end) approach for complex real-time embedded system design, with the latest algorithm exploration tools and reconfigurable hardware technologies. The proposed approach will address, for the first time, optimal and rapid design of embedded systems from high-level descriptions, targeting a combination of embedded processors, digital signal processing and reconfigurable hardware. We will develop modular and scalable hardware platforms that can be reused and re-targeted by the tool chain to produce optimized real-time embedded products. The results will be evaluated using advanced audio and video systems that support next-generation communication and entertainment facilities, such as immersive audio and mobile video processing. Innovations of our approach include: (a) *support for both diagrammatic and textual formats in algorithm description and exploration,* (b) *a framework that allows novel algorithms for design space exploration, which aims to automate design partitioning, task transformation, choice of data representation, and metric evaluation for both hardware and software components,* (c) *a system synthesis tool producing near-optimal implementations that best exploits the capability of each type of processing*

[1] www.hartes.org and www.bluebee-tech.com.

K. Bertels (✉)
Fac. Electrical Engineering, Mathematics & Computer Science, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: k.l.m.bertels@tudelft.nl

K. Bertels (ed.), *Hardware/Software Co-design for Heterogeneous Multi-core Platforms*,
DOI 10.1007/978-94-007-1406-9_1, © Springer Science+Business Media B.V. 2012

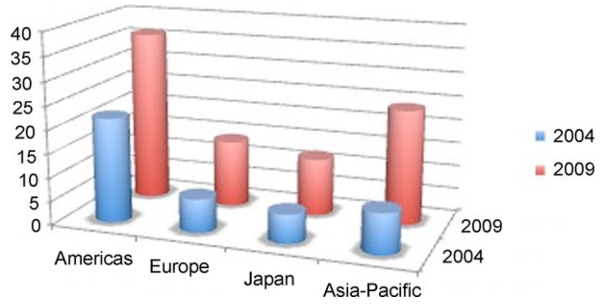


Fig. 1.1 Growth of the ES industry (EU report)

element; for instance, dynamic reconfigurability of hardware can be exploited to support function upgrade or adaptation to operating conditions. From the application point of view, the complexity of future multimedia devices is becoming too big to design monolithic processing platforms. This is where the hArtes approach with reconfigurable heterogeneous system becomes vital.

In this introductory chapter, we try to sketch the reasons justifying why the hArtes project was started and to what extent the technical results contribute to the main challenges of the Embedded Systems industry. We also briefly describe what constitutes the essence of the hArtes approach and what the anticipated benefits are for its users.

1.1 The Need for Advanced HW/SW Co-design

Europe has a strong leadership position in the Embedded Systems (ES) market over the US and China, which is highly segmented and delivers technology to end products in telecommunication, health, avionics, automotive, etcetera. A report commissioned by the EU, stipulates that the Embedded Systems market is highly competitive and rapidly evolving as new technologies are being introduced [1]. As shown in Fig. 1.1, it also grows at a higher rate than other ICT markets.

Furthermore, as shown in Fig. 1.2, the report indicates that it is expected that embedded components, such as advanced multimedia systems in a car, will represent an increasingly larger part of the total value of the end product.

Combined with the observation that ES are becoming ever more performing and richer in functionality, a number of challenges emerge for the industry. Increased complexity of applications Embedded Systems become increasingly software oriented but today's ES development assumes a close interaction between hardware and software decisions, generally termed as HW/SW co-design. The increased functionality of ES leads to increased complexity of the applications. The highly competitive nature of the ES market requires very efficient design strategies to minimise the non-recurring engineering costs.

Not only are the applications becoming increasingly complex, the computing power needed to provide the necessary real time performance can only be given by

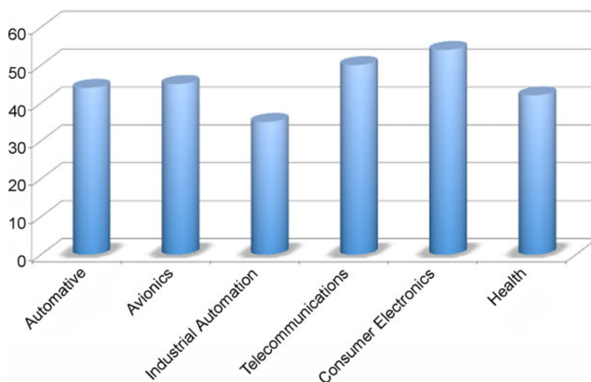


Fig. 1.2 Proportion of software cost in total cost of end product (EU report)

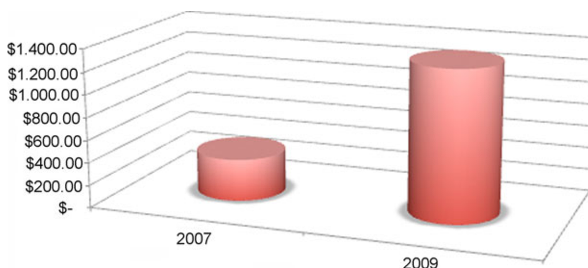


Fig. 1.3 Market for multi-core CPU's in ES

multi-core platforms. The increasing market of multi-core CPU's in ES is shown in Fig. 1.3.

The increased complexity poses enormous productivity challenges for the industry as they do not have the knowledge nor expertise sufficiently available to adopt in a smooth way such platforms. According to a study by VDC the military and communications industries are the most likely to adopt multi-core technologies as their applications require this kind of performance [2]. In addition, these sectors have been using multi-processor systems for a long time which makes the transition multi-core platforms more easy. In addition, they think that openMP will be increasingly adopted to make legacy code ready for execution on such multi-core platforms. However, enhancement and adjustment of available commercial embedded software to enable use with processors having two or more cores are primary concerns (VDC p. 6).

One important limitation, which is imposed on any technology, the ES industry may adopt, relates to the existing code base. As pointed out by the HIPEAC roadmap [3], software has become the rigid component in modern computer and embedded systems. Hardware tends to evolve much faster than software and ES companies do not want to loose that intellectual property nor go through expensive re-design processes with uncertain outcomes. When software companies were asked whether the Embedded Systems industry was prepared to address the issues for multi-core computing, the overall rating was 2.06 on a scale of 1 (not prepared) to 5 (well prepared) [2, p. 7]. This indicates that the ES does not consider itself to be well-prepared to address the challenges sketched above. Recent announcements by ARM

and Xilinx are further indications that the Embedded Industry is indeed going into the direction of (heterogeneous) multi-core systems. Contacts with large industrial corporations such as ABB also confirm that e.g. FPGAs are becoming increasingly important but only in combination with a general-purpose processor. Other kinds of heterogeneous platforms such as the TI Omap family are also heavily used in the ES industry.

Even for the entry-level multi-core platforms, mapping applications on them is considered problematic. Leading platform providers such as TI and STM who both have announced advanced multi-core platforms to be launched in 2010 are lacking good tools to ease this entire effort. One of the main reasons IBM no longer supports the Cell processor is the difficulty to program it. The hArtes technology relieves the system developer of this technical challenge by automating this process. If needed, the developer can always override mapping decisions proposed by the hArtes tools. It also bridges the gap between hardware and software design. In case reconfigurable components are used, the hArtes tools automatically generates the hardware kernels and will make the necessary modifications to the original source code while maintaining the functional equivalence. We define in more detail in the following section the advantages of our technology.

1.2 The hArtes Approach: A Unified View to Develop High Performance Applications

Embedded systems applications require increasingly more processing power for which single processor platforms are no longer sufficient. On the other hand, multi-core platforms not only find their way into the desktop and server markets but also in the embedded systems domain. Such platforms can contain any number of computing nodes, ranging from RISC processors to DSPs and FPGAs. Such platforms provide the necessary computing power industry needs for the most demanding applications. Increased computing power is also needed in a wide variety of markets that belong to the high-performance computing domain such as bio-informatics, oil and gas exploration and finance. Recently, Convey introduced a hybrid supercomputer on the market. It is one of the first commercial platforms to combine general purposed processors (Intel Xeon) with high performance FPGAs (Xilinx Virtex 6). The hybrid architecture is clearly the direction in which research and industry is heading. This trend poses specific challenges for both the embedded and the supercomputing domain. The complexity of such hardware requires detailed knowledge of the hardware platform and in the case where FPGAs are available, one needs to have hardware design knowledge to be able to use them. The hArtes approach aims to simplify the use of heterogeneous multi-core platforms by providing an integrated tool chain that will transform existing and new applications to allow execution on heterogeneous computing platforms.

Entry point for the tool chain, given in Fig. 1.4, is C-code that can be generated by (open or proprietary) tools such as matlab and scilab, manually coded or existing code. Information regarding the target platform is available in the form of

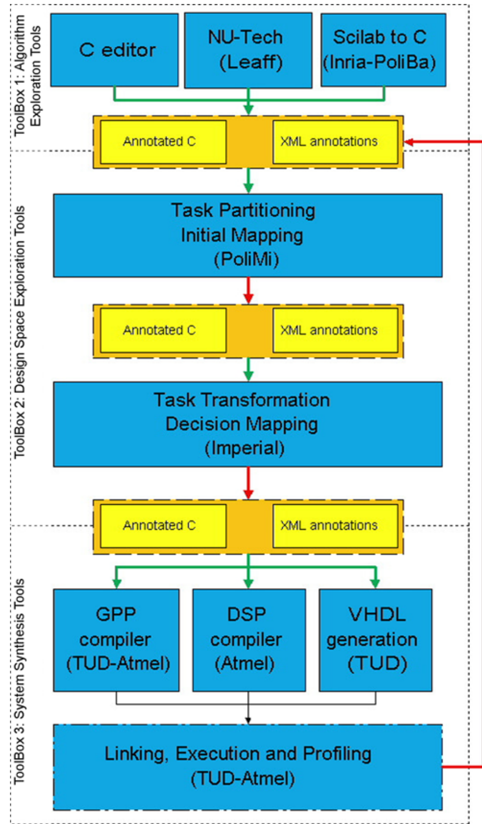


Fig. 1.4 The hArtes tool chain

XML files. If the developer has some prior knowledge with respect to the mapping of the application, simple code annotations can used to express that knowledge. The Task Transformation Engine then analyses the entire application and will decide on a mapping. This may involve that certain parts stay on the general-purpose processor; other parts should be executed by a DSP or can be best executed by an FPGA. These decisions are simply represented by additional and automatically inserted code annotations. The next set of compiler tools then take the annotated code and generate the appropriate binaries that are linked together. The generated executable can then be immediately executed on the HW platform. If needed, this process can be repeated until the design objectives are achieved. In order to better illustrate the power of our technology, we include some code samples that need to be written when mapping parts of an application on, in this case, a DSP accelerator. In the example, we assume that an FFT-routine will be executed on the accelerator. Figure 1.5 shows 3 different kinds of code snippets. The code examples on the left are respectively targeting either the Atmel Diopsis DSP platform or the TI Omap L138. The smaller code sample on the right provides exactly the same functionality as either examples on the left but then using the hArtes approach where one can abstract away from the lower level platform details as this is taken care of by the back-end tools of the hArtes tool chain.

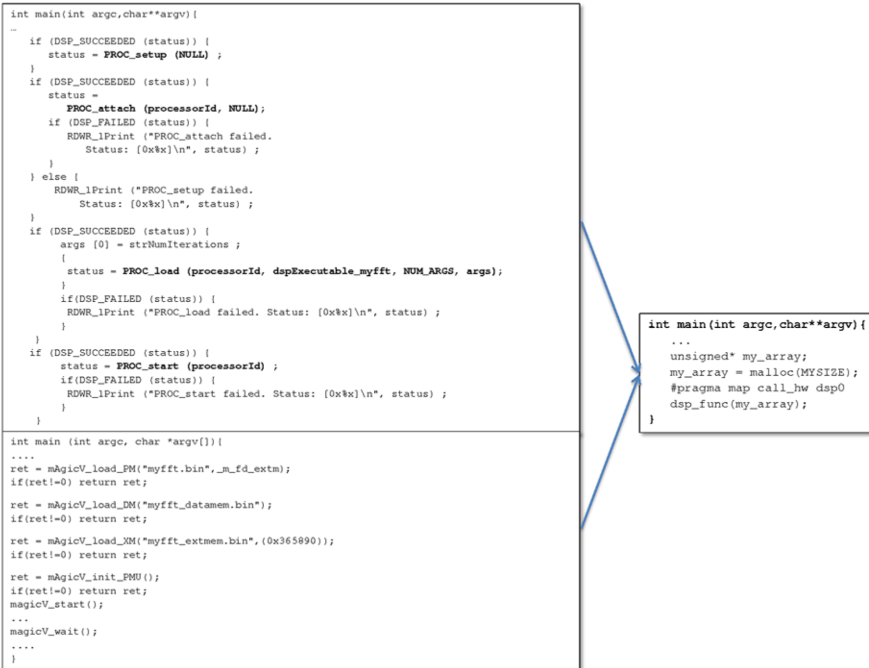


Fig. 1.5 Mapping Code on various platforms

1.3 Benefits of the hArtes Approach

In view of the above brief discussion, we have identified the following benefits of the approach described in the book.

Use of a Familiar Programming Paradigm One of the key challenges when adopting such platforms is that one is forced to use programming tools and languages that are very platform specific and require substantial code rewriting to port an existing application. Also, the learning process for using these tools is long and needs to be repeated each time when adopting another technology. hArtes proposes a familiar programming paradigm which is compatible with the widely used programming practice, irrespective of the target platform. In essence: conventional source code such as ANSI-C, will be annotated after some (semi) automatic transformations have been applied. These transformations are driven by performance requirements and real time constraints. The hArtes tool chain can automatically make the necessary transformations, mappings and subsequent code annotations. It also allows the developer to make simple annotations to the source code indicating which parts of the code will be accelerated.

See Multiple Cores as a Single Processor One of the distinguishing features of the hArtes approach is that it abstracts away the heterogeneity as well as the multicore aspect of the underlying hardware. The developer can view the platform as

consisting of a single general-purpose processor. This view is completely consistent with the programming paradigm that was presented above. The hArtes toolboxes take care of the mapping of parts of the application on the respective hardware components.

Easily Port Existing Applications Due to the high cost, embedded systems companies are conservative regarding the need to re-implement their applications when targeting new platforms. The cost for Validation and Verification and possibly certification is too high to be repeated when the underlying technology changes. hArtes provides a migration path where either through manual annotation or through the use of the tool chain in applying the necessary modifications. In a matter of days/hours, one can test on the real platform how the application behaves and, when necessary, the process can be repeated if the design objective has not been met.

Develop New Applications using Powerful Toolboxes Evidently, new applications will also be developed and possibly mapped on a variety of hardware platforms. In order to reduce this effort, the hArtes tool chain provides both high level algorithm exploration tools with subsequent automatic code generation which can then be fed to other toolboxes in the chain.

Easily Retargetable to New Hardware Platforms The tool chain can be easily retargeted to different hardware platforms. Again, this brings the benefit to keep using the same development tools and environments no matter what hardware platform one targets.

From Fully Automatic to Fully Manual From fully automatic to fully manual The developer has the choice to opt for either a full automatic mapping, a semi-automatic or even a fully manual one. At each step, decisions can be evaluated and overruled if considered inadequate.

1.4 Conclusion and Outline of the Book

In this chapter, we have tried to outline the transformation the Embedded Systems industry is currently undergoing. The introduction and necessity to adopt multi-core platforms have emerged as the most important challenge the Embedded Systems industry will be facing over the next years. It is crucial for the industry not to loose the investments made in the past and thus be able to port the legacy applications to the latest generation hardware platforms. In addition, the transition and creation of experience of and expertise in using multi-core platforms must be achieved as smooth as possible. We believe that hArtes like tools are one example of technology that can help in making this transition.

The book is organised as follows. We first introduce the hArtes tool chain as it was demonstrated towards the end of the project. In the subsequent chapters, we

describe the hardware platform that was developed as the main test and development platform of the project. The final chapters in the book present the different use cases that allowed to validate the entire hArtes technology and to identify also missing features and opportunities for further extensions, along with the features that were investigated or adopted for preliminary evaluation but not included in the final release of the tool chain.

References

1. Helmerich, A., et al.: Study of worldwide trends and research and development programmes in embedded systems in view of maximising the impact of a technology platform in the area. FAST GmbH, Munich, Germany, November 2005
2. Venture Development Capital: Multi-core computing in Embedded Systems. White paper (2007)
3. HiPEAC Roadmap, Duranton, M., et al.: Available at: http://www.hipeac.net/roadmap

Chapter 2
The hArtes Tool Chain

Koen Bertels, Ariano Lattanzi, Emanuele Ciavattini, Ferruccio Bettarelli, Maria Teresa Chiaradia, Raffaele Nutricato, Alberto Morea, Anna Antola, Fabrizio Ferrandi, Marco Lattuada, Christian Pilato, Donatella Sciuto, Roel J. Meeuws, Yana Yankova, Vlad Mihai Sima, Kamana Sigdel, Wayne Luk, Jose Gabriel de Figueiredo Coutinho, Yuet Ming Lam, Tim Todman, Andrea Michelotti, and Antonio Cerruto

This chapter describes the different design steps needed to go from legacy code to a transformed application that can be efficiently mapped on the hArtes platform.

2.1 Introduction

The technology trend continues to increase the computational power by enabling the incorporation of sophisticated functions in ever-smaller devices. However, power and heat dissipation, difficulties in increasing the clock frequency, and the need for technology reuse to reduce time-to-market push towards different solutions from the classic single-core or custom technology. A solution that is gaining widespread momentum consists in exploiting the inherent parallelism of applications, executing them on multiple off-the-shelf processor cores. Unfortunately, the development of parallel applications is a complex task. In fact, it largely depends on the availability of suitable software tools and environments, and developers must face with problems not encountered during sequential programming, namely: non-determinism, communication, synchronization, data partitioning and distribution, load-balancing, heterogeneity, shared or distributed memory, deadlocks, and race conditions.

Since standard languages do not provide any support for parallel programming some effort has been devoted to the definition of new languages or to the extension of the existing ones. One of the most interesting approaches is the OpenMP standard based on pragma code annotations added to standard languages like C, C++ and Fortran [45].

K. Bertels (✉)
Fac. Electrical Engineering, Mathematics & Computer Science, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: k.l.m.bertels@tudelft.nl

K. Bertels (ed.), *Hardware/Software Co-design for Heterogeneous Multi-core Platforms*,
DOI 10.1007/978-94-007-1406-9_2, © Springer Science+Business Media B.V. 2012

The aim of the hArtes toolchain is to have a new way for programming heterogeneous embedded architectures, dramatically minimizing the learning curve for novice and simultaneously speed up computations by statically and transparently allocating tasks to different Processing Elements.

Currently the hArtes toolchain addresses three platforms, based on Atmel Diopsis SOC. This SOC embeds an ARM9EJS and MAGIC, a floating point DSP. The OMAP family will be supported soon. These target platforms support a Fork/Join, non preemptive threading model, where a master processor spawns multiple software and hardware threads on the various processing elements (or on the FPGA) and retakes control after all of them terminate. This model clearly fits well with the OpenMP standard. In fact, the OpenMP standard has been adopted in this project since it is well supported and accepted among parallel application developers and hardware vendors. It is supported by many commercial compilers (Microsoft and IBM) and, in the OpenSource scene, by GCC [13], starting with the 4.2 release. The reasons behind the wide diffusion of OpenMP lie in the fact that it is considered a good mean to achieve portable parallel execution on shared-memory multiprocessor (SMP) systems, it allows the incremental parallelization of existing codes and it has a very powerful and complete syntax to express complex models of parallelism (e.g. for loops parallelism), but, at the same time, it remains simple and effective.

A subset of the OpenMP pragmas is used in the context of the hArtes project: `#pragma omp parallel` is used to express parts of code that potentially runs in parallel and, inside it, the `#pragma omp sections` declares the single parallel parts. Each `#pragma omp parallel` block acts as a fork and it implicitly joins the spawned threads at its end. It is interesting to note that nested `#pragma omp parallel` are allowed, giving the possibility to support fork from children threads. The annotations for the initial guesses on the mapping of the tasks on the target platforms will instead adopt an ad-hoc, independent syntax. Using an independent syntax for mapping makes sense since it is a different problem from thread decomposition and it is tightly coupled with the target platform. In this way the parallel code produced can be tested on different hosts that support OpenMP, simply ignoring the mapping directives.

For these reasons, differently from CUDA or other architectures that address the problem of "parallel computing", the hArtes toolchain doesn't impose to write applications by using new syntaxes or new libraries. In fact, the programmer can express parallelism as he always did on a PC platform (threads, OpenMP), or he may leave the toolchain to extract the parallelism automatically. Then, at the end of the compilation process the harts toolchain produces a single executable with all the symbols and debugging information for each processing element. The programmer has only to execute/debug it on the targeted heterogeneous platform as he was dealing with a single processor architecture (just like a PC).

The following sections of the chapter will first introduce the overall toolchain structure and then present the different tools composing it, concluding with the introduction of the overall framework user interface, showing how it can be used by the toolchain users for mapping their applications.

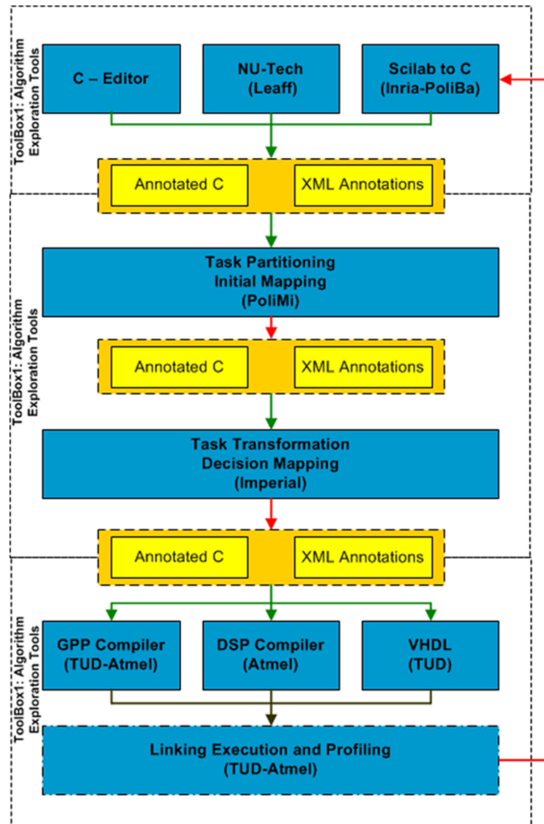


Fig. 2.1 The hArtes toolchain

2.2 Tool Chain Structure

Figure 2.1 illustrates the overall hArtes toolchain flow. The flow is constituted by several stages, and each stage can be iterated more than once. Information produced in the execution of the different phases is transferred from one phase to the next using C pragma annotations added to the application source code and XML annotations contained in a dedicated file that is modified and enriched by each tool. The pragma annotations are used to specify partitioning, mapping and profiling of the application that is currently analyzed by the toolchain. The XML annotations contain information about the target architecture (i.e., the available processing elements and their interconnections) and about the characterization of the implementations for each function to be mapped onto that architecture. In conclusion, all the information about the structure of the application is contained in the application source code, while the architecture description file provides all the information on the characteristics of the execution on the target architecture. The flow is composed of the following three main blocks:

1. the Algorithm Exploration and Translation (AET) toolbox (detailed in Sect. 2.4)
2. the Design Space Exploration (DSE) toolbox (detailed from Sect. 2.5 to Sect. 2.7.6)
3. the System Synthesis (SysSyn) toolbox (detailed from Sect. 2.8 to Sect. 2.10)

The Algorithm Exploration Tools (AET) are used in the first phase for the algorithm design and exploration. Its output is the algorithm C code. The available graphical or textual tools allow the designer to explore different algorithm solutions, until the final correct C code of the application (plus possible notations) has been obtained. The Design Space Exploration (DSE) tools are used in the second phase to manipulate and optimize the C code. The profiled C code is then partitioned in tasks taking into consideration the performance data just obtained. The resulting partitioned code, is further annotated to provide an initial guess on the mapping of each task on the processing elements of the target platform. Each task can then be transformed to optimize it for the specific processing elements on which it has been mapped. To reduce the amount of hardware required for an operation, the number of bits used to represent data needs to be minimized. This goal is addressed by the Data Representation optimization stage. All these stages provide new information and can be repeated several times to optimize the resulting code. Each task is finally committed to each processing element before code generation. Finally, the System Synthesis tools are used in the third (final) phase, to perform the compilation, the linking and loading of the application onto the target hardware. In particular, the code generation is performed by a specific back end for each target unit. The ELF objects for the software parts are merged to generate the executable code, while high level synthesis is performed and synthesizable VHDL is generated for the FPGA part.

The hArtes project includes also the development of the hArtes IDE, which is the human interface to the workspace. The integration in a unified toolchain of a set of tools going from the very high level of the application development to the lowest levels is an original contribution of hArtes and the integration methodology adopted is generic enough to conceive the hArtes framework as the basis on which other toolchain instances for embedded system development can be built. The rest of the chapter describes the functionalities expected from each tool represented in the toolchain.

2.2.1 *The Algorithm Exploration Tools*

The user has three different options to enter the hArtes toolchain:

1. Describe the application with Scilab
2. Describe the application with NU-Tech
3. Describe the application directly in C language

The AET Tools have the aim of translating in C language the application described with NU-Tech or with Scilab. Starting from an application described as

a NU-Tech network of Satellites (NUTSs), NU-Tech is able, thanks to the GAE Tool feature, to produce a C description of the application and pass the results to the toolchain DSE tools. In order to produce a C description of the application the source code of each functional block is needed. The GAE Tool feature handles the task of describing how each block interacts with each other and with the system I/O. The ready-to-use NUTS library allows the user to concentrate on the core of his/her own algorithm and write it down according to the GAE Tool specifications. Specifications are few and very easy to follow:

- fixed sampling frequency for each streaming session: any streaming session should work at a fixed sample rate.
- Init/Deinit/Process architecture: any algorithm should consist of these three functions.

As an alternative option, the application can be described using the scripting language of Scilab. A tool to translate the Scilab description into the C language description is developed and is integrated with a library of functions to support the DSP applications. The initial set of functions is selected according to the application partners requirements. Addition of new library functions is possible if the developer provides the underlying description in C, supporting all the different required implementations associated with the data types supported by Scilab (e.g. implementation for scalar variables, operating on individual items like float variables, or implementation for vectorial variables operating on array of data mono or two dimensional). In order to optimize the implementation on the embedded platform, dynamic memory allocation and de-allocation shall be avoided. The last option is directly writing the C description of the application. The code must respect some coding guidelines and it is possible to add some annotations to control the behaviour of the tools that will be subsequently called. The output of the AET tools is C code with optional pragma annotations added by the user.

2.2.2 The DSE Tools

The Design exploration tools are composed of a set of tools aiming at first at collecting information on the application, then on proposing a task partitioning based on cost estimations and applying transformations to these tasks in order to make them more suitable for their implementation on the given target platform. Finally, a phase of task mapping is performed to identify the most suitable hardware/software assignment of the different tasks, in order to optimize specific figures of merit.

At first a profiling of the application is performed. The Code Profiler provides information necessary for the subsequent tasks since it generates information on the CPU cycles of the different functions to identify computational hotspots and analyzes the memory usage and the access patterns. CPU usage is computed by pro, that has been modified to produce its output in the required format. The memory usage and access patterns provide as output a quantitative usage graph containing

information on the amount of data exchanged between two functions. The tools output their results by annotating the C code of the application. The computation time of a function will be provided as an annotation in the C-code with # pragma_profile directives. The two measures num_calls and time are provided for each function. num_calls indicates the number of times the function is called and the time expresses the running time of the program.

Task Partitioning

The task partitioning tool identifies the parallelism available in the application by analyzing the tree structure of the code and groups the identified independent operation clusters at the appropriate granularity level. Moreover, the task partitioning tool automatically proposes an initial guess on the tasks mapping, defining which parts of the application should be implemented in hardware, or which parts should be executed in software and on which kind of processing element available in the target system. The mapping is based on an initial cost estimation taken into account together with the performance. The requirement analysis shows that the problem can be clearly separated in two distinct sub-problems:

1. in the first phase the tool identifies the parallelism in the application, i.e. shall annotate, using task partitioning C pragma notations, how it can be decomposed in separate threads, presenting the parallel tasks within
2. the annotated code; in the second phase the tool will indicate, using specific code annotations, an initial guess concerning which processing element (GPP, DSP or programmable logic) should execute the identified tasks, i.e. where each task should be mapped in the system. The overview of panadA automatic parallelization process is shown in Fig. 2.11.

The inputs are the C description of the application and an XML file containing information about the architecture and the performance of the application on each processing element. When the XML file does not provide any performance estimation, the task partitioning tool performs an internal estimation of the performance of the application. Feedback from the toolchain can improve the accuracy of the performance estimations and therefore the quality of the partitioning and of the initial mapping identified by the partitioning tool. The result of Task partitioning is a code annotated with Task Partitioning annotations and possibly with HW assignment annotations.

Task Mapping

The task mapping tool (**hArmonic**) has two main inputs: (1) the source code, supporting an arbitrary number of C source files and (2) the XML platform specification. The platform specification describes the main components of the heterogeneous system, including the processing elements, the interconnect, storage components, and system library functions. The output of the task mapping tool is a set of C

sources. Each source is compiled separately for each backend compiler targeting a processing element, and subsequently linked to a single binary. The hArmonic tool is divided in four stages:

- **C Guidelines Verification**. Automatically determines whether the source code complies with the restrictions of the mapping process, allowing developers to revise their source in order to maximize the benefits of the toolchain.
- **Task Filtering**. Determines which processing elements can support each individual task in the application in order to generate feasible mapping solutions on the hArtes platform.
- **Task Transformation**. Generates several versions of the same task in order to exploit the benefits of each individual processing element, and maximize the effectiveness of the mapping selection process.
- **Mapping Selection**. Searches for a mapping solution that minimizes overall execution time.

Cost Estimation

The cost estimation process includes the contribution of Imperial, PoliMi and TUD. Each contribution solves a different problem in the hArtes toolchain. Imperial's cost estimator is used for the task mapping process (**hArmonic**) to determine the cost of each task when executed on a particular processing element. In this way, the task mapper is able to decide, for instance, whether there is a benefit to run a particular task on a DSP instead of a CPU. PoliMi's cost estimator is a module integrated in the partition tool (**zebu**) which assists in deriving a better partition solution and reduces the design exploration time. Finally, the cost estimator from TUD is used to aid early design exploration by predicting the number of resources required to map a particular kernel into hardware. Its output is a set of information added to the XML file.

2.2.3 The System Synthesis Tools

The System Synthesis tools purpose is to create a unified executable file to be loaded on the HW platform starting from the C codes produced by the task mapping tool. The System Synthesis tools are:

- a customized version of the C compiler for the GPP (**hgcc**),
- a DSP compiler enriched with tools to generate information required by the hArtes toolchain,
- an RTL generator from the C code provided as input,
- a linker to generate the unified executable image for the hArtes HW platform, containing the executable images associated with the GPP, DSP and FPGA,
- a loader to load the executable image sections into the associated HW platform memory sections,

- additional tools and libraries for format conversion, metrics extractions, interaction with the operating system running on the platform.

The **hgcc Compiler** for the target GPP is extended to support pragma annotations to call the selected HW implementation. The effect of these pragmas is to expand functions into a sequence of Molen APIs (see later). This customized version of the C compiler for the GPP is called hgcc since it is a customization for hArtes (h) of the gcc compiler. The **DSP Compiler** is the compiler available for the DSP. It must be completed by adding an executable format conversion in order to create a unified elf file for the entire application (see later mex2elf) and a tool to add the cost information to the XML file (see later DSP2XML). The **VHDL Generator** tool (C2VHDL) generates an RTL HDL from the C code provided as input. This tool include two steps. First, the input C-description of the algorithm shall be presented as a graph, which shall be then transformed into a set of equivalent graphs, introducing different graph collapsing techniques. Second, a metric base decision on the optimal graph representation shall be made and this graph representation shall be translated into RTL level and then to a selected HDL. The **DSP2XML** tool collects information about the code compiled on the DSP. The collected information concerns the optimization achieved by the task allocated on the DSP, such as code size and execution speed. The **mex2elf:** tool converts the executable format generated by the DSP C compiler into the object format used by the GPP linker, i.e. ARM little endian elf format. The **MASTER GPP Linker** creates an elf executable ready to be loaded by the target OS (Linux). It links Molen Libraries, GPP and DSP codes. The FPGA bitstream is included as a binary section. It is composed of a set of customization of the scripts controlling the code production, and customization to the linker itself is avoided in order to reuse the linker tool from the GNU toolchain. The **MASTER GPP Loader** is in charge of loading DSP and MASTER codes on the Target platform. The FPGA bitstream is loaded into a portion of the shared memory accessible by the FPGA. The loading process is performed in the following steps:

1. Loading of the code sections following the default Linux OS strategy, including the m-mapping of the hArtes memory sections associated with the HW platform components (DSP, FPGA).
2. Customized C Run Time (CRT) performing the additional operations required by the hArtes HW/SW architecture, such as copy of the memory sections to the physical devices using the appropriate Linux drivers developed to access the HW platform components (DSP, FPGA).

Dynamic FPGA reconfiguration is performed by copying the required configuration bitstream to the FPGA from the system shared memory where all the available configuration bitstreams were loaded during the loading phase.

2.3 hArtes Annotations

The high level hArtes APIs are POSIX compliant. Software layers have been added to make uniform C, thread and signal libraries of PEs. The POSIX compliance

makes easy the porting of application across different architectures. Low level and not portable APIs are available to the application in order to access directly particular HW resources like timers, audio interfaces. The hArtes toolchain requires an XML architecture description, where main HW and SW characteristics are described. Beside the XML description (provided by the HW vendors), the toolchain uses source annotations via pragmas. From the developer point of view source annotations are optional because they are generated automatically by the toolchain. Both source and XML annotations can be used by the developer to tune the partitioning and mapping of the application.

2.3.1 XML Architecture Description File

The XML Architecture Description File aims at providing a flexible specification of the target architecture and it is used for information exchange between the tools involved in the hArtes project. First we provide a brief overview of the XML format. Next we present the organization of the XML file for architecture description. Then we describe the interaction of the hArtes tools with the presented XML file.

There are many advantages of using the XML format for the Architecture Description File. The XML format is both human and machine readable, self-documenting, platform independent and its strict syntax and parsing constraints allow using efficient parsing algorithms. In consequence, the XML format is suitable for the structured architecture description needed by the tools involved in the hArtes project.

The structure of an XML document relevant for the Architecture Description File is shortly presented in this section. The first line of the XML file usually specifies the version of the used xml format and additional information such as character encoding and external dependencies (e.g. `<?xml version="1.0" encoding="ISO-8859-1"?>`).

The basic component of an XML file is an element, which is delimited by a start tag and an end tag. For example, the following element:

```
<name>hArtes</name>
```

has `<name>` as the start tag and `</name>` as the end tag of the "name" element. The element content is the text that appears between the start and end tags. Additionally, an element can have attributes, such as:

```
<name id="12">hArtes</name>
```

An attribute is a pair of name-value, where the value must be quoted.

Finally, we mention that every XML document has a tree structure, thus it must have exactly one top-level root element which includes all the other elements of the file.

The root element of the XML description is named `ORGANIZATION`. It contains the following elements:

- HARDWARE: which contains information about the hardware platform;
- OPERATIONS: which contains the list of operations (i.e., C functions) and implementations;
- PROFILES: which contains additional information generated by tools.

```
<ORGANIZATION>
   <HARDWARE>
      ...
   </HARDWARE>
   <OPERATIONS>
      ...
   </OPERATIONS>
   <PROFILES>
       ...
   </PROFILES>
</ORGANIZATION>
```

The HARDWARE XML Element

The HARDWARE element contains the following elements:

- NAME: name of the board/hardware;
- FUNCTIONAL_COMPONENT: which describes each processing element;
- STORAGE_COMPONENT: which describes the storage (memory) elements;
- BUS_COMPONENT: which describes architectural interconnection elements;
- VBUS_COMPONENT: which describes virtual (direct) interconnection elements.

Each FUNCTIONAL_COMPONENT is composed of:

- NAME: the unique identifier for the FUNCTIONAL_COMPONENT, such as *Virtex4* or *ARM*. It should be a valid C identifier as it can be used in pragmas;
- TYPE: the class of the functional component. Valid values are *GPP*, *DSP* and *FPGA*. Based on this information, the proper compiler will be invoked the related stitch code will be generated;
- MODEL: represents the specific model for the processing element (e.g., *XILINX VIRTEX XC2VP30*);
- MASTER: whether the functional component is the master processing element or not. The element contains *YES* if the component is the master or *NO* otherwise. Only one functional component can be defined as master.
- SIZE: which is relevant only for reconfigurable hardware and represents the number of available Configurable Logic Blocks (CLBs).
- FREQUENCY: is the maximum frequency of the functional component, expressed in MHz;
- START_XR: the starting range of the transfer registers associated with that functional component.
- END_XR: the ending of the range of the transfer registers.

- HEADERS: contains the list of all headers to be included when the C code is split to the corresponding backend compilers. It is composed of:
  – NAME: (multiple) filename name of the header.
- DATA: contains C data (storage) specification. In particular, it specifies the following elements:
  – MAXSTACKSIZE: the maximum stack size.
  – DATA_TYPE: is of list of the C basic types. Note that a **typedef** can be treated as a basic type if included here. In this case, it does not matter if the hidden type does not resolve to a basic type.
    * NAME: name of the basic type;
    * PRECISION: is the precision of the data type.

Example:

```
<FUNCTIONAL_COMPONENT>
  <NAME>Arm</NAME>
  <TYPE>GPP</TYPE>
  <MODEL>ARMv9</MODEL>
  <MASTER>YES</MASTER>
  <START_XR>1</START_XR>
  <END_XR>512</END_XR>
  <SIZE>0</SIZE>
  <FREQUENCY>250</FREQUENCY>
  <HEADERS>
    <NAME>my_header.h</NAME>
    <NAME>my_header2.h</NAME>
  </HEADERS>
  <DATA>
    <MAXSTACKSIZE>1000</MAXSTACKSIZE>
    <DATA_TYPE>
      <NAME>int</NAME>
      <PRECISION>32</PRECISION>
    </DATA_TYPE>
  </DATA>
</FUNCTIONAL_COMPONENT>
```

Similarly, each STORAGE_COMPONENT contains the following elements:

- NAME: for the name of the component. E.g. MEM1. One storage component, connected to FPGA must be named XREG.
- TYPE: type of the memory. E.g., SDRAM.
- SIZE: the size of the memory in kilobytes. E.g., 16.
- START_ADDRESS: is the starting range of memory addresses in the shared memory. These should be hexadecimal number and thus you must use “0x” prefix.
- END_ADDRESS: is the ending range of memory addresses in the shared memory. These should be hexadecimal number and thus you must use “0x” prefix.

Example:

```
<STORAGE_COMPONENT>
  <NAME>MEM1</NAME>
  <TYPE>SDRAM</TYPE>
  <SIZE>128</SIZE>
  <START_ADDRESS>0</START_ADDRESS>
  <END_ADDRESS>0xFFFFFFFF</END_ADDRESS>
</STORAGE_COMPONENT>
```

The `BUS_COMPONENT` contains:

- `NAME`: used to identify the bus.
- `BANDWIDTH`: the size of one memory transfer in kbytes/sec (for example: 1024).
- `FUNCTIONAL_COMPONENTS`: functional components that can access this bus
  – NAME: the name of the functional component.
- `STORAGE_COMPONENTS`: storage components connected on this bus
  – `NAME`: the name of the storage component.
- `ADDR_BUS_WIDTH`: The size in bits of the address bus to the corresponding storage component. Used by the DWARV toolset for CCU interface generation, if the bus is not connected to a FPGA, the element is optional.
- `DATA_BUS_WIDTH`: The size in bits of the read and write data busses for the corresponding storage component. Used by the DWARV toolset for CCU interface generation. If the bus is not connected to an FPGA, the element is optional.
- `READ_CYCLES`: The number of cycles to fetch a word from the corresponding storage element. If the storage component runs at different frequency than the CCU, the number of access cycles has to be transferred as CCU cycles. For example, if the storage component is clocked at 200 MHz and requires 2 cycles to fetch a word and the CCU operates in 100 MHz, the cycles in the component description are reported as 1. If the storage component is clocked at 50 MHz, the CCU at 100 MHz, the fetch cycles are 2, then the component description contains 4 as read cycles. If the storage component has non-deterministic access time, the cycles element shall be set to “unavailable”.

Example:

```
<BUS_COMPONENT>
  <NAME>InternalFPGA</NAME>
  <TYPE>INTERNAL</TYPE>
  <BANDWIDTH>1024</BANDWIDTH>
  <ADDR_BUS_WIDTH>32</ADDR_BUS_WIDTH>
  <DATA_BUS_WITH>64</DATA_BUS_WITH>
  <READ_CYCLES>2</READ_CYCLES>
  <FUNCTIONAL_COMPONENTS>
    <NAME>FPGA</NAME>
  </FUNCTIONAL_COMPONENTS>
  <STORAGE_COMPONENTS>
    <NAME>MEM1</NAME>
```

```
  </STORAGE_COMPONENTS>
</BUS_COMPONENT>
```

The VBUS_COMPONENT element represents the direct interconnection between two functional elements

- NAME is the bus interconnect identifier. Example: VBUS1;
- FUNCTIONAL_COMPONENT_NAME (multiple) name of the functional component inside this bus;
- BITSPERUNIT the number of bits exchanged per unit;
- BANDWIDTH the number of units transferred per unit of time.

```
<ORGANIZATION>
  <HARDWARE>
    <VBUS_COMPONENT>
      <NAME>id</NAME>
      <FUNCTIONAL_COMPONENT_NAME>component 1</FUNCTIONAL_COMPONENT_NAME>
      <FUNCTIONAL_COMPONENT_NAME>component 2</FUNCTIONAL_COMPONENT_NAME>
      <BITSPERUNIT>21</BITSPERUNIT>
      <BANDWIDTH>102</BANDWIDTH>
    </VBUS_COMPONENT>
  <HARDWARE>
<ORGANIZATION>
```

The OPERATIONS XML Element

The OPERATIONS element is used for the description of the operations that are implemented on the hardware components. It contains a list of OPERATION elements which contain the associated functional components and describe the hardware features of each specific implementation.

```
<ORGANIZATION>
   <OPERATIONS>
      <OPERATION>
          ...
      <OPERATION>
      <OPERATION>
          ...
      <OPERATION>
   </OPERATIONS>
</ORGANIZATION>
```

The OPERATION element structure is:

- NAME, the name of the operation in the C file.
- multiple COMPONENT elements containing:
  - NAME, this has to be a name of an existing FUNCTIONAL_COMPONENT.
  - multiple IMPLEMENTATION elements containing
    - ID, an unique identifier (for all the XML) for each hardware implementation.

* SIZE, for the implementation size. For each type of component this will have a different meaning as follows: for the FPGA it will be the number of 100 slices, for the GPP and for the DSP it will be the code size in KB.
* START_INPUT_XR, START_OUTPUT_XR, for the first XRs with the input /output parameters.
* SET_ADDRESS, EXEC_ADDRESS, for the memory addresses of the microcode associated with SET/EXEC, can be omitted if not FPGA.
* SET_CYCLES, EXEC_CYCLES, for the number of component cycles associated with hardware configuration/execution phase, can be omitted if not FPGA.
* FREQUENCY—the frequency of the implementation on FPGA in MHz. If the functional component is not FPGA, this can be omitted.

Example:

```
<OPERATION>
  <NAME>SAD</NAME>
  <COMPONENT>
    <NAME>Arm</NAME>
    <IMPLEMENTATION>
      <ID>11</ID>
      <SIZE>100</SIZE>
      <START_INPUT_XR>3</START_INPUT_XR>
      <START_OUTPUT_XR>10</START_OUTPUT_XR>
      <SET_ADDRESS> 0X00000000 </SET_ADDRESS>
      <EXEC_ADDRESS> 0X00000000 </EXEC_ADDRESS>
      <SET_CYCLES> 100 </SET_CYCLES>
      <EXEC_CYCLES> 200 </EXEC_CYCLES>
    </IMPLEMENTATION>
  </COMPONENT>
</OPERATION>
```

In the presented Architecture Description File, there is a clear delimitation about the hardware/software features of the target architecture/application. The information contained in an OPERATION element (such as size, frequency) has to be provided by the automatic synthesis tools that generates a specific implementation for a specific operation. Finally, the information for the HARDWARE element (such as Memory sizes, GPP type) is general and should be introduced by the architecture designer.

The PROFILES XML Element

The PROFILES element stores the result of some tools in the hArtes toolchain.

HGPROF is a profiling program which collects and arranges statistics of a program. Basically, it captures performance information about specific elements of the program such as functions, lines, etc. Currently, HGPROF captures the following information which are stored in the PROFILES element:

- `NAME`: is the name of a function;
- `STIME`: is an average execution time (ms) of each function per call without subroutine calls;
- `CTIME`: is an average execution time (ms) of each function per call with subroutine calls;
- `NCALLS`: is the number of times a function is called;
- `LINE` element containing
  - `NUMBER` is the line number;
  - `NCALLS` is the number of time a line is executed.

Example:

```
<PROFILE>
    <HGPROF>
         <FUNCTION>
             <NAME> function1</NAME>
             <STIME>134</TIME>
             <CTIME>1340</TIME>
             <NCALLS>32</NCALLS>
             <LINE>
                  <NUMBER> 34</NUMBER>
                  <NCALLS> 350</NCALLS>
             </LINE>
         </FUNCTION>
    </HGPROF>}
</PROFILE>}
```

As HGPROF collect this information on the associated tag the same happened for QUIPU and QUAD.

2.3.2 Pragma Notation in hArtes

Pragmas are used to embed annotations directly into the source-code. The advantage of pragmas over XML is that the code itself carries the annotations; however it can also clutter the code and make it less easy to read. The biggest benefit of using #pragmas is that they can be used to annotate constructs such as assignment and loop statements, which, differently from functions, do not have obvious identifiers. Moreover, it is worth noting that the program structure has to be revisited when there are functions that interact outside the program itself. In fact, if the program contains IO directives or supervisor calls, it is problematic to extract parallelism from them since memory side-effects cannot be controlled or predicted. For this reason, functions are classified into two different types:

- data interfacing functions
- data processing functions

The former ones should be excluded from the parallelism extraction. In particular, when a data-interfacing function contains calls to functions communicating with the external world (e.g. IO or supervisor calls), it shall be marked with a specific directive to be excluded from the partitioning. In this way, a better performing code will be obtained since the parallelism extraction operates only on the data processing region, where no interfacing with external world occurs. For this reason, the source code has to be compliant with this structure to allow the parallelism extractor tool to concentrate its effort on meaningful parts of the application. Note that how functions exchange data is not a constraint, but the distinction between interfacing and processing functions is.

Few common format directives will be shared among the different pragma notations used in hArtes. The common directives will refer to the format of a single line pragma notation. No common rules will be defined to describe grammar expressions built with the composition of multiple pragma lines. The notation is case sensitive. Pragma notation lines will be specified with the following format:

```
pragma ::= #pragma <pragma_scope> [<pragma_directive>] [<clauses>]
new_line
```

White spaces can be used before and after the '#' character and white spaces shall be used to separate the *pragma_scope* and the *pragma_directive*. The *pragma_scope* is composed of a single word; multiple words, separated by white spaces, may compose a *pragma_directive*.

```
pragma_scope ::= <word>
pragma_directive ::= <word> [<word> [<word>...]]
```

The *pragma_scope* identifier specifies the different semantic domain of the particular *pragma_directive*. The specific hArtes pragma scopes are the following:

- **omp** for the OpenMP domain;
- **profile** for the profiling domain;
- **map** for the hardware assignment domain;
- **issue** for generic issues not related to the previous scopes, but mandatory for hArtes toolchain.

They are used to reduce the potential conflict with other non-hArtes pragma notations. The *pragma_directive* identifies the role of the pragma notation within the scope of the specific semantic domain. For the different directives, see the paragraph related to the specific scope.

`<clauses>` represents a set of clauses, where each clause is separated by white space:

```
clauses ::= <clause> [<clause> [<clause>]]
```

The clause may be used to define a parameter or a quantitative notation useful for a pragma_directive. Each clause is comprised by a name and a list of variables separated by commas:

```
clause ::= <name> [(<variable>[,<variable>...])]
```

The order in which the clauses appear is not significant. The *variable* identifier's, if any, shall be used to define only variable parameters.

OpenMP Domain

We adopt a subset of the OpenMP pragmas to describe parallelism (task partitioning and thread parallelism). OpenMP pragmas are denoted by the *pragma_scope* **omp**. Note that the OpenMP syntax is used to annotate how tasks are partitioned. Inter-thread synchronization is not used except for barriers, as the hArtes specifications explicitly ask for extraction of independent threads with a load-execute-commit behaviour.

Parallel Construct Program code supposed to run in parallel is introduced by the pragma:

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
{
      structured-block
}
```

This #pragma creates a team of threads. The thread that first encounters this pragma becomes the master of the new team. Note that all the spawned threads execute the code in the structured-block. Code inside square brackets [ ] denotes optional elements.

Even if the OpenMP specs provide support for different clauses, we only consider the *num_threads* and *default(shared)* clauses: the former to express the number of threads spawned for the parallel region, the latter to explicit the only supported variable management among the spawned threads. Other clauses are not supported. It is worth noting that if unsupported clauses are present in the original source code, the parallelism extraction analysis is aborted, since the original semantics could be changed. Note also that without the *num_threads* clause we cannot a priori determine the number of threads that will be created, since the behaviour is compiler dependent. Nested *#pragma omp parallel* constructs are supported: each thread that reaches this nested pragma becomes master of a new team of threads. Note that at the closing bracket } of the *#pragma omp parallel* there is an implicit barrier that joins all the spawned threads before returning control to the original master thread.

Worksharing Constructs OpenMP allows the use of work-sharing constructs inside a parallel region to distribute the execution of the code in the parallel region to the spawned threads. A work-sharing region must bind to an active parallel region: worksharing constructs are thus introduced only inside *#pragma omp parallel* constructs. OpenMP defines the following work-sharing constructs:

- loop (**for**) construct
- **sections** construct
- **single** construct

At the moment, we only support the **sections** worksharing construct. This is used to express the sections of code running in parallel. Each structured block is executed once by one of the threads in the team. The other construct are considered as invalid. The sections worksharing constructs is declared with the following pragma:

```
#pragma omp sections [clause[[,] clause] ...] new-line
{
    structured-block
}
```

Note that at the moment these clauses are ignored by the partitioning tool. Inside the brackets { } the different parallel blocks of code are declared with the pragma:

```
#pragma omp section
{
    structured-block
}
```

Notice the lack of the s at the end of the section clause.

This is how *#pragma omp sections* combines with *#pragma omp section*:

```
#pragma omp sections [clause[[,] clause] ...] new-line
{
   [structured-block]
   #pragma omp section new-line
   {
      [structured-block]
   }
   #pragma omp section new-line
   {
      [structured-block]
   }
   ...
}
```

Note that the first *#pragma omp section* can be omitted, since it is always assumed that the following structured code is executed by one thread only. The following *#pragma omp section* constructs must instead be present to signal that the structured blocks of code below will be executed by other threads. Note that nested parallel regions are always possible. So a nested *#pragma omp parallel* can be declared in the structured code belonging to a *#pragma omp section*.

Barrier Annotation Synchronization points (barriers) for parallel region and for sections are implicit at their end. This means that a barrier is present at the closing bracket } of each *#pragma omp parallel*, *#pragma omp sections* and *#pragma omp parallel sections*.

Anyway the explicit pragma is supported by our tool for additional information inside parallel regions and sections. The format is:

```
#pragma omp barrier new-line
```

Note that OpenMP allows the use of this pragma at least inside a first level parallel region. Of course it can be used inside *#pragma parallel sections* (which binds to a parallel region) to wait for the termination of certain structured blocks of code before starting the others. When this construct appears all the threads spawned to execute the binding parallel region must reach the barrier before proceeding. Note anyway that since implicit barriers are present at the end of parallel regions and sections, the code can be written without using this pragma, without losing expressive power. Since nested parallel regions are possible, the barrier construct binds to the innermost region.

Profiling Domain

The profiling domain is used to capture performance information about specific elements of the program (functions, loops, etc.). In general, the `#pragma` profiling directive must appear in the line before the construct it refers to. Note that white spaces (newlines, spaces and tabulation characters) may exist between the pragma and the corresponding C construct.

The notation used for this directive is: *#pragma profile* data where data contains the performance measures, defined by the following regular expression:

```
data := measure_name(mean,variance)
        [measure_name(mean,variance)]*
```

Profiling information is structured in clauses. These clauses contain statistical measures on the performance of the application, in the sense that both mean and standard deviation values should be given for each measure. This is due to the fact that application behaviour is, in general, dependent on the input data.

So far two measures are used:

- *num_calls*
- *time*

The former indicates the number of times the function is called while the latter expresses the running time of the program which was spent in the function in micro seconds.

The *#pragma profile* clause must appear immediately before the function it refers to; at most there can be blank spaces (newline, spaces or tabulation characters) among them. This clause can either be inserted before the function prototype or before the function body; in case both of the notations are present, only the one before the prototype will be considered Example:

```
#pragma profile num_calls(5,0.8)
void f1();
```

means that function f1 is called an average of 5 times and the standard deviation of the measurements is 0.8.

```
#pragma profile time(30,0.02)
void f1();
```

means that 30 micro seconds of the execution time of the program is spent is executing function f1; the standard deviation is 0.02

```
#pragma profile time(30,0.02) num_calls(5,0.8)
void f1();
```

is just the combination of the previous two notations; note that the order between time and num_calls is not important.

```
#pragma profile time(30,0.02) num_calls(5,0.8)
void f1()
{
.........
.........
}
```

is the same notation seen before, but this time used before the function body and not the function prototype. The *#pragma profile* clauses are not mandatory, but the information they carry may be useful to improve the partitioning process. The information depends of course on the platform used. The profile information is given for the GPP processor, which is the processor that contains the MASTER element in the architecture description file.

Mapping Domain

The mapping domain is used to instruct the backend compilers how tasks are mapped to different processing elements. We use two pragma directives:

- **generation pragmas**, used to mark that, an implementation needs to be built for one particular functional component (like FPGA or DSP)
- **execution pragmas**, used to indicate which functional component or which specific implementation a particular function is mapped for being executed.

The generation pragma is placed immediately before the definition of the function to be compiled by one of the hardware compilers. The syntax is:

```
#pragma  generate_hw  impl_id
```

The <*impl_id*> corresponds to the implementation identifier in the XML file.

The hardware compilers (DWARV, the Diopsis compiler) will read the XML and determine if they need to generate bitstream/code for that function.

The execution pragma is placed immediately before the definition of the function to be executed or offloaded to the hardware component and the corresponding *pragma_directive* is **call_hw**. The syntax is:

```
#pragma call_hw  <component_name> [<impl_id>]
```

The *<component_name>* is the unique identifier of the hardware component where the function will be executed. The *<impl_id>* is the identifier of the chosen implementation associated with the function on the specified component. Additional information about the implementation is in the XML architecture file. If the *<impl_id>* is not specified, it means that no implementations are still available and the mapping gives information only about the target processing element and not the implementation (e.g., specific information about area/performance is not available).

Examples:

```
#pragma call_hw ARM
void proc1 (int* input, int* output1, int* output2)
{
   ...
}

#pragma call_hw FPGA 2
void proc2(int* input, int* output1, int* output2)
{
   ...
}

#pragma call_hw ARM 0
void proc3 (input1, &output1, &output);
```

The function *proc1* will be executed by the ARM component, but no information about a specific implementation is available. Note that, since this pragma has been put before the declaration, it means that all the instances of the *proc1* function will be mapped in this way. The function *proc2*, instead, will be executed on the component named FPGA, with the implementation identified by the id 2. In this case, the synthesis toolchain has also information about the corresponding implementation that has to be generated. In the other hand, if you desire to specify a different mapping for each call of the function, you have to insert the pragma immediately before the related function call. In this example, only the specified *proc3* function call will be implemented on ARM with the implementation with id 0 and no information is given about the other calls of the same function in the program.

Issue Domain

These clauses contain general information or issues for the application. An issue could be the desire for the programmer to exclude a routine from the partitioning process. The notation used for this directive is: *#pragma issue* and a list of directives to be applied to the function that follows:

```
#pragma issue [directive[[,] directive] ...] new-line
```

So far only the *blackbox* directive has been considered. This directive forces the related routine to be excluded from the partitioning process. It can be used by the

programmer to indicate that the function will not been partitioned. This directive is often applied to functions containing I/O since it is problematic to extract parallelism when they are involved. For this reason the IO regions (data input and data output) should be marked with the *#pragma issue blackbox* directive. In this way, these two regions will be excluded from parallelism extraction and a better performing code will be obtained.

The *#pragma issue* clause must appear immediately before the function it refers to; at most there can be blank spaces (newline, spaces or tabulation characters) among them. This clause can either be inserted before the function prototype or before the function body; in case both of the notations are present, only the one before the prototype will be considered.

Example:

```
#pragma issue blackbox
void proc4 (int* input, int* output1, int* output2)
{
   ....
}
```

In this case, the routine *proc4* will be considered as a black-box and no partitioning is tried to be extracted from it.

2.4 Algorithm Exploration

The Algorithm Exploration Toolbox (AET) has the role of enabling the high level description of the application, which shall be translated in C code. The main aim of the AET is to provide tools with two basic functionalities:

- Assist the designers in tuning and possibly improving the input algorithm at the highest level of abstraction in order to easily obtain feedback concerning the numerical and other high-level algorithmic properties.
- Translate the input algorithms described in different formats and languages into a single internal description common for the tools to be employed further on. In the hArtes tool-chain, this internal description is done in C language.

The Algorithm Exploration Toolbox, in particular, deals with high-level algorithms design tools. Its main goal is to output a C description of the input algorithm complemented by optional specification directives reflecting the algorithmic properties obtained from the tools.

This ToolBox translates the multiple front-end algorithmic entries considered into a single unified C code representation.

In these terms the most simple AET Tool may be just a traditional text editor, for C programming, possibly with the typical features such as syntax highlighting, automatic indenting, quick navigation features, etc. The user which is writing C code to be processed by the hArtes tool-chain, differently from other C programmers, has the opportunity to enrich the code with additional information that may be useful

in the activity of task partitioning, code profiling and code mapping. Nevertheless, the syntax that has been defined in the hArtes project, shall be based on C pragma notation, and then compatible with the use of a typical C text editor.

C code programming, anyway, is often not the most efficient way of designing the application algorithms. The hArtes AET toolbox offers the opportunity of describing the application also using both graphical entry and computational-oriented languages.

The hArtes Consortium decided, since the beginning, to adopt NU-Tech as Graphical Algorithm Exploration (GAE) solution and Scilab as computation-oriented language. Later on this chapter the specific issues concerning the integration of these tools in the hArtes ToolChain will be analyzed.

2.4.1 Scilab

Scilab is a scientific software package for numerical computations providing a powerful open computing environment for engineering and scientific applications.

Scilab is an open source software. It is currently used in educational and industrial environments around the world.

Scilab includes hundreds of mathematical functions with the possibility to add interactively programs from various languages (C, C++, Fortran...). It has sophisticated data structures (including lists, polynomials, rational functions, linear systems...), an interpreter and a high level programming language.

Scilab supports the following features:

- 2-D and 3-D graphics, animation
- Linear algebra, sparse matrices
- Polynomials and rational functions
- Interpolation, approximation
- Simulation: ODE solver and DAE solver
- Xcos: a hybrid dynamic systems modeler and simulator
- Classic and robust control, LMI optimization
- Differentiable and non-differentiable optimization
- Signal processing
- Metanet: graphs and networks
- Parallel Scilab
- Statistics
- Interface with Computer Algebra: Maple package for Scilab code generation
- Interface with Fortran, Tcl/Tk, C, C++, Java, LabVIEW

Overview

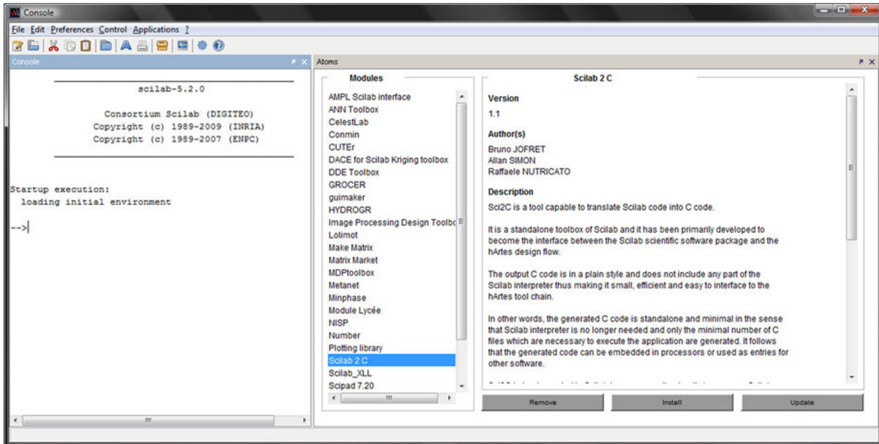
Scilab2C is a tool capable to translate Scilab code into C code.

It is a standalone toolbox of Scilab and it has been primarily developed to become the interface between the Scilab scientific software package and the hArtes design flow.

The output C code is in a plain style and does not include any part of the Scilab interpreter thus making it small, efficient and easy to interface to the hArtes tool chain.

In other words, the generated C code is standalone and minimal in the sense that Scilab interpreter is no longer needed and only the minimal number of C files that are necessary to execute the application are generated. It follows that the generated code can be embedded in processors or used as entries for other software.

Scilab2C is a Scilab toolbox available through Atoms (AuTomatic mOdules Management for Scilab) and can be directly installed/used in the Scilab Development Environment.



From Scilab Script to C Code

The aim of the tool is to give an easy path, from Scilab code down to C code, to a user who may not have any skills on low level language programming but who wants to have an accelerated execution of its high level representation of the algorithm.

Scilab takes advantage from:

- an easy high level programming and testing environment.
- C code for quicker execution and possibly other optimizations (hardware acceleration, parallelization, . . .).

Scilab Scripting Language Scilab provides a powerful language to exploit those capabilities:

- High level, non-typed and non-declarative programming language: the user does not need to take care about memory allocation, variable type declaration and other programming habits C programmers are used to. Moreover Scilab is a non-declarative language which means the user is allowed to use variables without declaring (nor typing) them before.
- Lazy syntax and control structure instructions: with respect to other programming languages Scilab has a different way to write control structures (if/then, for, func-

tion) that will give the user some freedom writing their scripts/function avoiding the syntax strictness of C code. Nevertheless Scilab language is as rich as other programming language regarding control capabilities: if, for, while, select, try, ...

- Matrix oriented language with an extended choice of standard mathematics computations: Scilab provides and extended set of standard mathematics capabilities, allowing the user to apply mathematical algorithms and will be able to easily manipulate the obtained results.

To sum it up the user can write any algorithm, found in a publication or any paper, using Scilab scripting language, and the result will "look like" the original except that it can physically run on a PC and then be tested, improved, etc. (see figure below).

$$b(n, p, j) = \binom{n}{j} p^j q^{n-j}$$

$$\binom{n}{j} = \frac{(n)_j}{j!}$$

$$\begin{aligned}(n)_j &= n \cdot (n-1) \cdots (n-j+1) \\ &= \frac{n \cdot (n-1) \cdots 1}{(n-j) \cdot (n-j-1) \cdots 1} \\ &= \frac{n!}{(n-j)!j!}\end{aligned}$$

prototype.sce - Scilab text editor

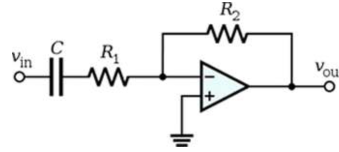
File Edit Search View Document Execute ?

```
function r = b(n, p, j)
  r = combinaison(n,j) * p^j * q^(n-j)
endfunction

function c = combinations(n, j)
  c = n_j(n,j) / factorial(j)
endfunction

function r = n_j(n, j)
  r = factorial(n) / (factorial(n-j) * factorial(j))
endfunction
```

Example: High-pass filter

Let us consider a simple example like this high-pass filter, the following equations can be extracted:

$$\begin{cases} V_{out}(t) = I(t) \cdot R \\ Q_c(t) = C \cdot (V_{in}(t) - V_{out}(t)) \\ I(t) = \frac{dQ_c}{dt} \end{cases}$$



This equation can be discretized. For simplicity, assume that samples of the input and output are taken at evenly-spaced points in time separated by Δ_t time.

$$V_{out} = RC\left(\frac{V_{in}(t) - V_{in}(t - \Delta_t)}{\Delta_t} - \frac{V_{out}(t) - V_{out}(t - \Delta_t)}{\Delta_t}\right)$$

$$\begin{cases} V_{out}(i) = \alpha V_{out}(i-1) + \alpha\,(V_{in}(i) - V_{in}(i-1)) \\ \alpha = \frac{RC}{RC+\Delta_t} \end{cases}$$

We can now easily implement this high-pass filter using Scilab:

high_pass.sce (D:\high_pass.sce) - Scilab text editor

File Edit Search View Document Execute ?

high_pass.sce

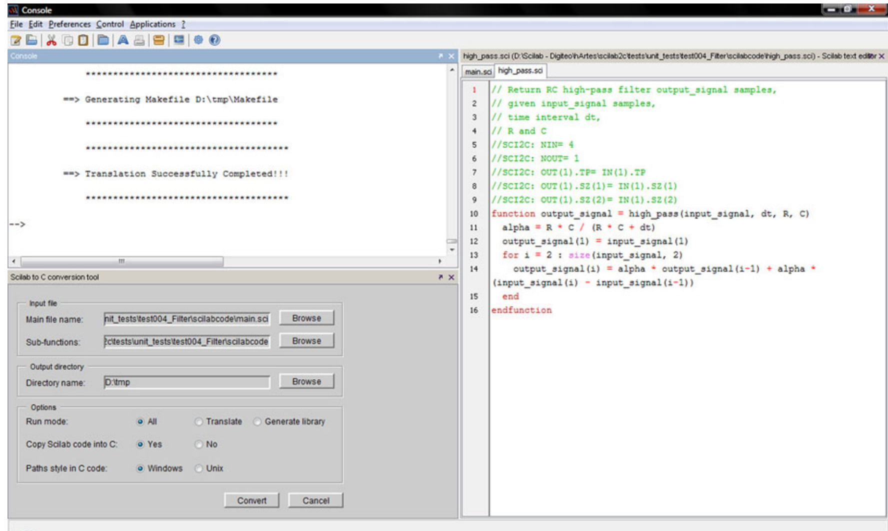
```
// Return RC high-pass filter output_signal samples,
// given input_signal samples,
// time interval dt,
// R and C
function output_signal = high_pass(input_signal, dt, R, C)
  alpha = R * C / (R * C + dt)
  output_signal(1) = input_signal(1)
  for i = 2 : size(input_signal, "*")
    output_signal(i) = alpha * output_signal(i-1) + alpha * (input_signal(i) - input_signal(i-1))
  end
endfunction
```

C Code Generator The output C code produced by the tool is written in plain C style, so it can be easily manipulated by optimizing tools available in the hArtes toolchain.

The generated code is standalone and minimal in the sense that Scilab interpreter is no longer needed and only the minimal number of C files, which are necessary to execute the application, are generated and linked. The generated code can then be embedded in processors or used as entries for other software.

As indicated in the following figure, from Scilab the user only has to:

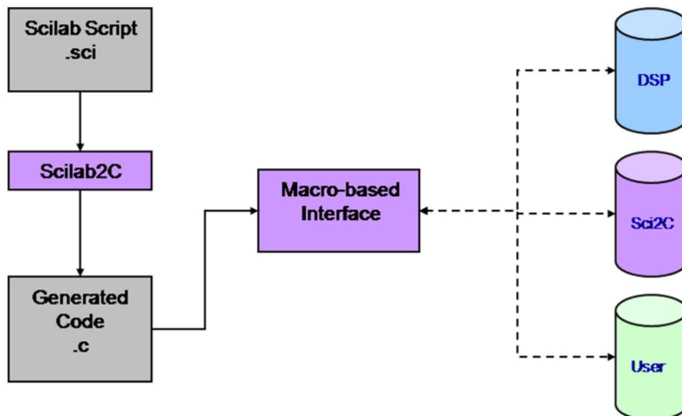
1. annotate the Scilab functions in order to specify the size and type of the output arguments;
2. (optional) specify the precision to be used for the data;
3. launch a GUI to generate the corresponding C-Code.



Using Generated C Code The generated code follows some clear naming rules that will allow the user to link this code with the given library or with a dedicated one (DSP, GPU, . . .).

Each time Scilab2C finds a function call to translate, it automatically generates an explicit function name containing information about input and output types and dimensions.

According to the naming rules, the user will be able (see figure below) to link its generated code with the "implementation" he considers as the best:



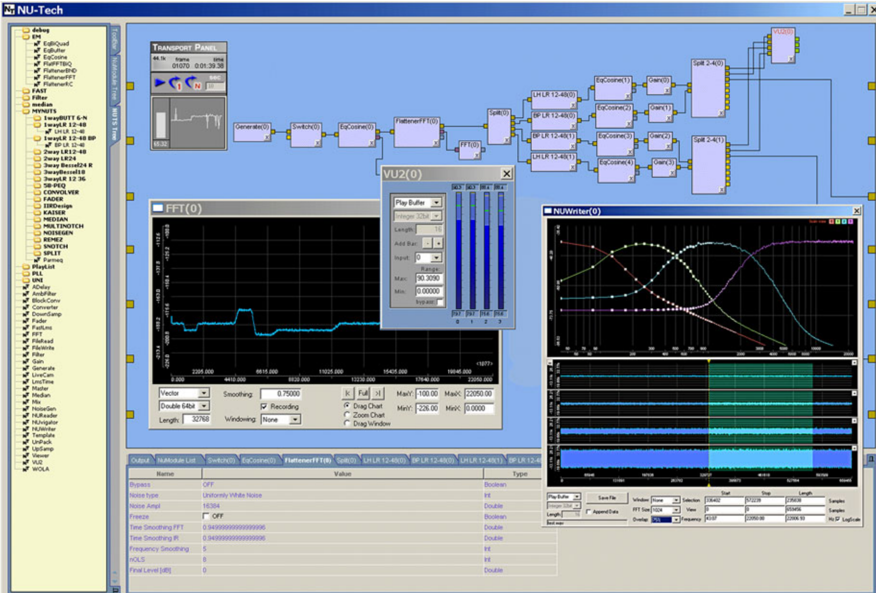


Fig. 2.2 NU-Tech graphical user interface

2.4.2 GAETool

NU-Tech Overview

Starting from the Leaff NU-Tech core the GAE Tool has been designed to play a central role as a starting point in the hArtes tool chain.

NU-Tech is a platform aimed to real-time scenarios, where a strict control over time and latencies is paramount. It represents a powerful DSP platform to validate and real-time debug complex algorithms, simply relying on a common PC. If the goal is implementing a new algorithm, no matter what kind of hardware the final target will be, NU-Tech offers all the right benefits.

During the early stage of the development, it is possible to evaluate the feasibility of a project saving time and money.

The user interface is simple and intuitive, based on a plug-in architecture: a work area hosts the NUTSs (NU-Tech Satellites) that can be interconnected to create very complex networks (limited only by the power of the computer). The graphical representation of a NUTS is a white rectangle with a variable number of pins, and it is the elementary unit of a network. From a developer point of view it is nothing but a plug-in, a piece of code compiled as a DLL. The overview of NU-Tech graphical user interface is show in Fig. 2.2.

A basic set of logical and DSP NUTS is provided and an SDK shows how to develop user's own Satellites in few steps in C/C++ and immediately plug them into the graphical interface design environment. That means the user can strictly focus on his piece of code because all the rest is managed by the environment.

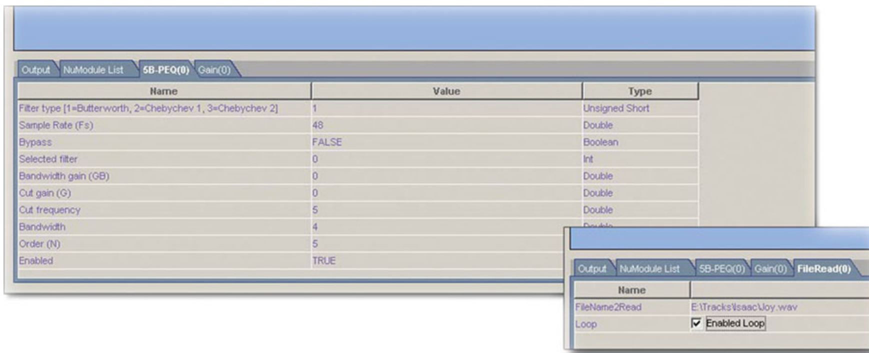


Fig. 2.3 NU-Tech RealTime Watch

The core of NU-Tech has been thought in a modular way so to *connect* to the external world by means of interchangeable drivers. For audio real-time applications ASIO 2.1 has been adopted, providing compatibility with all soundcards supporting the protocol.

A switch to a different driver turns NU-Tech into another tool but NUTSs are still there and the user can always take advantage of their functionalities. New drivers will be continuously added in the future giving the platform even more flexibility.

NUTSs are not compelled to provide a settings window in order to change algorithm parameters and for hArtes needs they should not. To ease the developer in quickly creating new NUTSs without having to deal with GUI programming, NU-Tech provides a window called “RealTime Watch” to be associated to each NUTS. In brief, the developer can choose, by code, to *expose* some NUTSs’ internal variables on this window, and effectively control his plug-in. The window is nothing but a tab on the bottom Multitab pane that automatically pops up when at least one parameter is exposed by the developer. When the user double-clicks on a NUTS, the associated tab (if any) immediately jumps in foreground.

A *Profiling Window* (see Fig. 2.3) has been designed in NU-Tech indicating:

- Time Consumption of all the NUTSs Process Functions
- Percentage of time slot (ex. FrameSize/Fs) used for Processing

NU-Tech profiling Window is shown in Fig. 2.4. NU-Tech distinguish between audio and video processing. This could be useful in a scenario where audio and video processing are performed by two different dedicated hardware.

GAETool Overview

The Graphical Algorithm Exploration Tool is a new feature of the NU-Tech Framework. Its main purpose is to produce a C description of an application designed in NU-Tech as a network of functional blocks (NUTSs) interconnected with each other and interacting with the PC inputs/outputs. The general overview of GAETool is shown in Fig. 2.5.

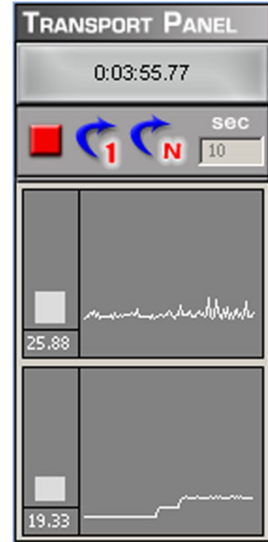


Fig. 2.4 NU-Tech Profiling information

Developing an application with NU-Tech plus GAETool feature is the most straightforward way to take advantage of the hArtes toolchain benefits. The user can use the library of CNUTS to build the core of his application, use the guidelines to code his own algorithm and benefit by the graphical NUTS to get visual feedbacks, plot diagrams and, most of all, real-time tune his parameters in order to obtain satisfactory results and start the porting to the hArtes hardware.

In order to produce a C description of a network of NUTS the source code of each block should be available to the tool. Moreover some specific guidelines should be followed when coding a NUTS in order to let the tool work properly.

CNUTS blocks have been introduced for this purpose. A CNUTS is nothing but a NUTS coded following the hArtes CNUTS specifications. A CNUTS is a .dll that must export some functions in order to be considered by NU-Tech a valid CNUTS. Very simple and easy to follow guidelines have been given to those willing to code a CNUTS. Each CNUTS is characterized by three main functions:

- Init(. . .): executed during CNUTS initialization phase.
- Process(. . .): the main processing, starting from input data and internal parameters produces output values.
- Delete(. . .): frees CNUTS allocated resources.

Any algorithm should consist of these three functions. NUTSs and CNUTSs programming therefore relies on the following structure:

- LEPlugin_Init: Called by the host when streaming starts. Initialization code should be here included.
- LEPlugin_Process: Called by host during streaming to process input data and pass them to the output. Processing code should be placed here.

Fig. 2.5 GAETool

Parameters:

– Input[H]: Pointer to an array of pointers each identifying a PinType structure. Each structure contains information about the type of data of the related input.
– Output[P]: Pointer to an array of pointers each identifying a PinType structure. Each structure contains information about the type of data of the related output.

- LEPlugin_Delete: called by the host when streaming is stopped. It contains deinitialization code.

GAETool is based on:

- Eclipse Ganymede version
- GNU Toolchain (MinGW 5.1.4 + GDB 6.8)
- GCC 3.4.5 compiler to compile CNUTS

GAETool is mainly composed by three different features:

- an Eclipse Wizard: to help the user to create a CNUTS from scratch;
- an Eclipse Feature: to export coded CNUTS to NU-Tech environment;
- a NU-Tech feature: to generate C code of CNUTS applications.

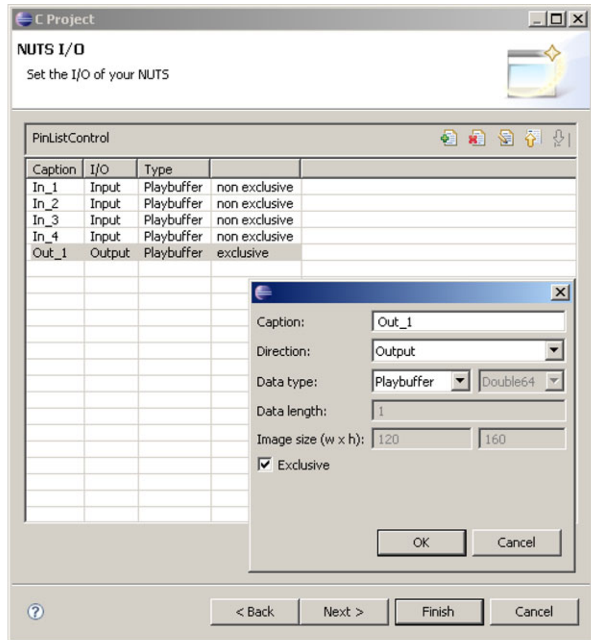


Fig. 2.6 GAETool Eclipse Wizard

GAETool Eclipse Wizard

A snapshot of GAETool Eclipse wizard is shown in Fig. 2.6. When creating a new C Project CNUTS template can be used to start a new CNUTS from scratch. The wizard follows the user during the creation phase and helps him defining:

- basic Settings (name, author, etc.);
- CNUTS mode: NUTS can work in ASIO, DirectSound, Trigger and offline mode;
- processing settings: which kind of processing flow to use (Audio, Video or MIDI)
- NUTS I/O: defining each single input/output of the CNUTS, its name, type and properties
- RTWatch settings: defining which variables should be monitored in realtime

As a result the wizard generates a complete CNUTS projects with all source files needed to correctly compile a CNUTS. All needed functions to handle all NUTS capabilities are included. The project is ready to host the programmer's code and properly compile using MinGW toolchain.

GAETool Eclipse Feature

Once the programmer compiled his project with no errors he can then export it to NU-Tech environment through the GAETool Eclipse feature. A "Export to GAE-Tool" button is included in Eclips IDE. Pressing the button resulting in a transfer to NU-Tech appropriate folders of CNUTS source code and compiled DLL. The user can now run NU-Tech being able to use his new CNUTS.

GAETool NU-Tech Feature

Once the new CNUTS has been ported to NU-Tech it can be used in a configuration together with other NUTS and CNUTS. At this stage of development one can debug and tune his CNUTS in order to get desired behaviour. Graphical NUTS can be also used to get visual feedback, for example: plotting a waveform or a FFT can really help understanding if an audio algorithm is correctly working. Once the user is satisfied with results he can drop all non-CNUTS and prepare his code to be passed to the next stage: the hArtes toolchain. All he has to do is to click on the GAETool icon in NU-Tech toolbar in order to generate the code of the whole configuration. The code is now ready to be used by the hArtes toolchain.

2.5 Application Analysis

This phase aims at providing information on the current status of the application in order to increase the effectiveness of the design space exploration, both for the partitioning and the mapping. In fact, it can be executed not only at the beginning of the exploration, to provide initial information, but also after the partitioning, to support the decision mapping. Moreover, when the toolchain is iteratively executed, the resulting application from each iteration can be further analyzed to provide an additional support to lead the exploration towards an efficient solution. In particular, two different analyses can be performed to support the toolchain:

1. profiling information can improve the partitioning and reduce the computation time focusing the analysis and the transformations only on the most executed paths;
2. cost estimation for the hardware implementations has to be provided to support the decision mapping in deciding which tasks can fit into the reconfigurable device.

2.5.1 HGPROF: Profiling Applications

In the context of hArtes, Hgprof is used as a preliminary profiling tool. Hgprof uses GNU profiling tools such as gprof [15] and gcov[17] tools. Gprof and gcov are an example of a software profiler that provides functional level profiling. Gprof analyzes the program at functional level and provides various information on the functions. Gcov is a coverage tool and analyzes a program to provide information such as how often each line of code executes. Towards this end, the profiling information from gprof and gcov has been processed to achieve various functional characteristics. Hgprof takes applications (C sources) as an input and it generates profiling information as XML output. Figure 2.7 shows the Hgprof tool flow.

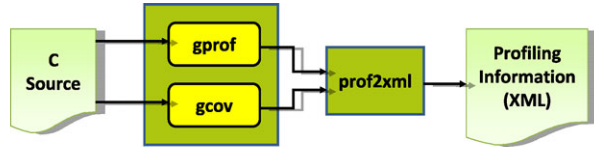


Fig. 2.7 Hgprof Tool Flow

The tool provides the following profiling information:

- Number of times a function is called,
- Average execution time (ms) of each function per call without subroutine calls,
- Average execution time (ms) of each function per call with subroutine calls,
- Number of times a line is executed per function call.

An example of the profile information provided by the tool is shown below. This information is provided in the .xml format under <HGPROF> tag under <PROFILE>.

```
<PROFILE>
    ....
    <HGPROF>
         <MODULE  Filepath = "C:/application" >
               <FUNCTION>
                    <NAME> function1 </NAME>
                    <STIME>134</TIME>
                    <CTIME>1340</TIME>
                    <NCALLS>32</NCALLS>
                    <LINE>
                      <NUMBER> 34 </NUMBER>
                      <NCALLS> 350 </NCALLS>
                    </LINE>
               </FUNCTION>
         </MODULE>
    </HGPROF>
     ....
</PROFILE>
```

where,

<MODULE>: is the header of the module section;
<NAME>: is the name of a function;
<STIME>: is an average execution time (ms) of each function per call without subroutine calls;
<CTIME>: is an average execution time (ms) of each function per call with subroutine calls;
<NCALLS>: is a number of times a function is called;
<NUMBER>: is a line number;
<NCALLS>: is a number of time a line is executed.

2.5.2 Cost Estimation for Design Space Exploration: QUIPU

During the implementation on reconfigurable heterogeneous platforms, developers need to evaluate many different alternatives of transforming, partitioning, and mapping the application to multiple possible architectures. In order to assist developers in this process the hArtes project uses the Design Exploration (DSE) ToolBox to analyze the application at hand and subsequently transform and map the application. The profiling and analysis tools in this toolbox identify compute-intensive kernels, estimate resource consumption of tasks, quantify bandwidth requirements, etc.

The Quipu modeling approach is part of the DSE ToolBox and generates quantitative prediction models that provide early estimates of hardware resource consumption with a focus on the Virtex FPGAs that are used within the hArtes project. Usually estimation of hardware resources is performed during high-level synthesis, but in contrast Quipu targets the very early stages of design where only C-level descriptions are available. By quantifying typical software characteristics like size, nesting, memory usage, etc. using so-called Software Complexity Metrics, Quipu is able to capture the relation between hardware resource consumption and C-code. These software complexity metrics are measures like the number of variables, the number of loops, the cyclomatic complexity of the CFG, and so on. These metrics can be determined in a short time, which makes fast estimates possible. Especially in the early stages of design where many iterations follow each other in a short amount of time, this is an advantage.

The estimates of the quantitative models that Quipu generates help drive system-level simulation, when reconfigurable architectures are targeted. Such modeling and simulation frameworks require estimates for FPGA resource consumption like area and power. Furthermore, Quipu models can provide valuable information for task transformations, e.g. large tasks can be split, while small tasks can be aggregated, etc. Also, task partitioning on reconfigurable heterogeneous platforms is possible when early estimates are available. And of course, manual design decisions benefit from resource estimates.

Quipu Modeling Approach

The Quipu modeling approach uses software complexity metrics (SCM) to represent code characteristics relevant to hardware estimation. Based on a library of kernels, it then extracts a set of SCMs for each every kernel, as well as a calibration set of corresponding actual resource consumption measurements. The dependence of the two datasets is then quantified using statistical regression techniques. The outcome of this process is a (linear) model that can be used to predict the resource consumption of kernels in applications that are being targeted to a heterogeneous platform. There are two tools and a set of scripts that implement the modeling approach as depicted in Fig. 2.8:

- *Metrication Tool* This tool parses the source code and calculates the software complexity metrics needed for prediction. The tool is off-line, although Quipu might also incorporate metrics from run-time in the future.

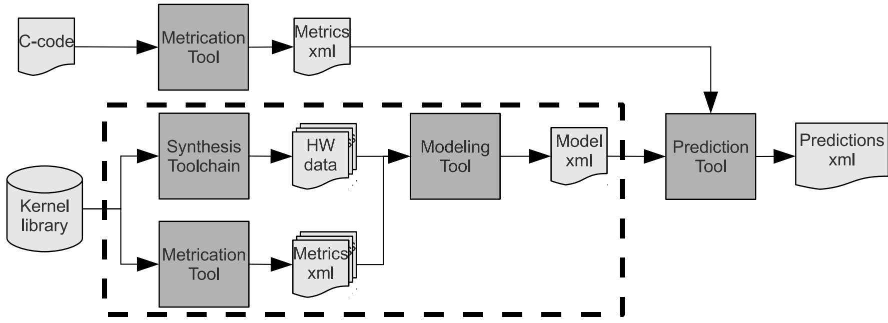


Fig. 2.8 Interaction of tools, scripts, and intermediate results in the Quipu modeling approach

- *Prediction Tool* This tool reads in an earlier generated model file and a set of metric values in order to make predictions for the modeled hardware characteristics.
- *Modeling scripts* These scripts generate calibration sets using a kernel library and help choose and tune the correct regression techniques. However, the process itself remains largely manual as building a good statistical model requires creativity and insight in the problem at hand.

As Quipu is a modeling approach instead of only a model, it is able to generate models for different combinations of architectures, tools, parameters, and hardware characteristics. As an example, Quipu could generate an area estimation model for Xilinx Virtex-5 XC5VLX330 FPGAs using the DWARV C2VHDL compiler and Xilinx ISE 11.1. However, other models are possible, i.e. estimating interconnect or power, targeting Altera Stratix or Actel IGLOO FPGAs, considering optimizing area or speed-up, assuming Impulse-C or Sparc, etc. As long as the predicted measure is to some extent linearly dependent on software characteristics, Quipu can generate specific models. Within the hArtes framework, Quipu has generated models for the Virtex2pro and Virtex4 FPGAs using the DWARV C2VHDL compiler and Xilinx ISE 10.1. We have shown that in those instances Quipu can produce area and interconnect models that produce estimates within acceptable error bounds. Indeed, Quipu trades in some precision for speed and applicability at a high level, but these properties are very important in the highly iterative and changing context of the early design stages. In the following we will discuss Quipu in more detail.

Statistics

In order to model the relation between software and hardware, Quipu utilizes statistical regression techniques. Regression captures the relation between two data sets. Observations of the dependent variables are assumed to be explained by observations of the independent variables within certain error bounds. Consider the following equation:

$$y_i = F(x_{i1}, \ldots, x_{in}) = \hat{F}(x_{i1}, \ldots, x_{in}) + \epsilon_i \tag{2.1}$$

where y_i is the i'th dependent variable, x_{ij} are the independent variables, $F()$ is the relation between the variables, $\hat{F}()$ is the estimated relation, i.e. the regression model, and ϵ_i is the error involved in using the regression model. The linear regression techniques employed by Quipu generate regression models that are of the form:

$$\hat{y}_i = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n + \epsilon_i \tag{2.2}$$

where β_i are the regression coefficients that were fitted to the model. It is clear from this equation that predictions using such models requires the measurement of the independent variables and a fixed amount of multiplications and additions. This is one of the main reasons for the speed of Quipu models.

In order to perform linear regression, a calibration set of measurements of the dependent and independent variables is needed translating to the following equation:

$$\bar{y}_i = \mathbf{X}_i \bar{\beta}_i + \bar{\epsilon}_i \tag{2.3}$$

Using linear regression on this equation the vector of linear coefficients ($\bar{\beta}_i$) can be estimated using the design matrix ($\mathbf{X}_i$) with all SCM observations and the vector of dependent variable observations ($\bar{y}_i$) i.e. the hardware measurements. There are several different regression techniques to choose from. Up to now, we have used Principal Component Regression, Generalized Linear Regression, Partial Least Squares Regression, LEAPS Regression Subset Selection, etc. in the context of Quipu.

A clear advantage of linear regression modeling is that the final prediction consists of a few additions and multiplications in addition to measuring the required SCMs for the kernel. On the other hand linear models are not capable of capturing the non-linear aspects of the transformation of C to hardware, nor can it predict run-time behavior without factoring in other variables that relate to e.g. input data. These issues translate into a relatively large error for QUIPU models. However, at the very early stages of design even a rough indication of resource consumption can be invaluable.

Kernel Library

In order to perform regression we need to build a calibration set of measurements for the dependent and independent variables. For this purpose we have built a library of software kernels that represents a broad range of applications and functionalities. We have shown that a model calibrated using this kernel library can be used for area prediction with acceptable error bounds. Currently, this library consists of over a hundred kernels. Table 2.1 shows the details on the composition of this library. There are no floating point kernels in the library at the time of writing, because when the library was constructed the C2VHDL tools at hand were restricted to integer arithmetic.

Table 2.1 Number of functions in each domain with the main algorithmic characteristics present in each application domain

Domain	Kernels	Bit-based	Streaming	Account-keeping[1]	Control-intensive
Compression	2	x		x	x
Cryptography	56	x	x		x[2]
DSP	5	x	x		x^b
ECC	6	x	x		x
Mathematics	19				
Multimedia	32	x^b	x		x
General	15			x^b	x
Total	135				

[1]Non-constant space complexity

[2]Only some instances in that domain express this characteristic

Software Complexity

In order to quantify the software characteristics of a certain kernel, our approach utilizes software complexity metrics (SCM). The SCMs applied in Quipu capture important features of the kernel source code that relate to the hardware characteristics that it wants to predict. Some examples of SCMs are: Halstead's measures (# of operators, operands), McCabe's cyclomatic number, counts of programming constructs (# of loops, branches, etc.), nesting level, Oviedo's Def-Use pairs, and so on. At the moment, Quipu employs more than 50 SCMs in several categories: code size, data intensity, control intensity, nesting, code volume, etc. The more complex SCMs come mostly from the field of Software Measurement and were originally intended for planning source code testing and project cost estimation. However, we also have introduced some new metrics that specifically try to quantify features related to hardware implementation.

Results

The Quipu modeling approach has been demonstrated to generate usable models for FPGA area and interconnect resources on Xilinx Virtex FPGAs. In Fig. 2.9 we see the predicted versus the measured number of flip-flops. The predictions were made by a Quipu model generated for the Virtex-4 LX200 [59] combined with the DWARV C2VHDL compiler and Xilinx ISE 10.1 Synthesis toolchain optimizing for speed-up. We Observe that the predictions are fairly accurate, however non-linear effects seem to affect smaller kernels more severely. The overall expected error of this model is 27%.

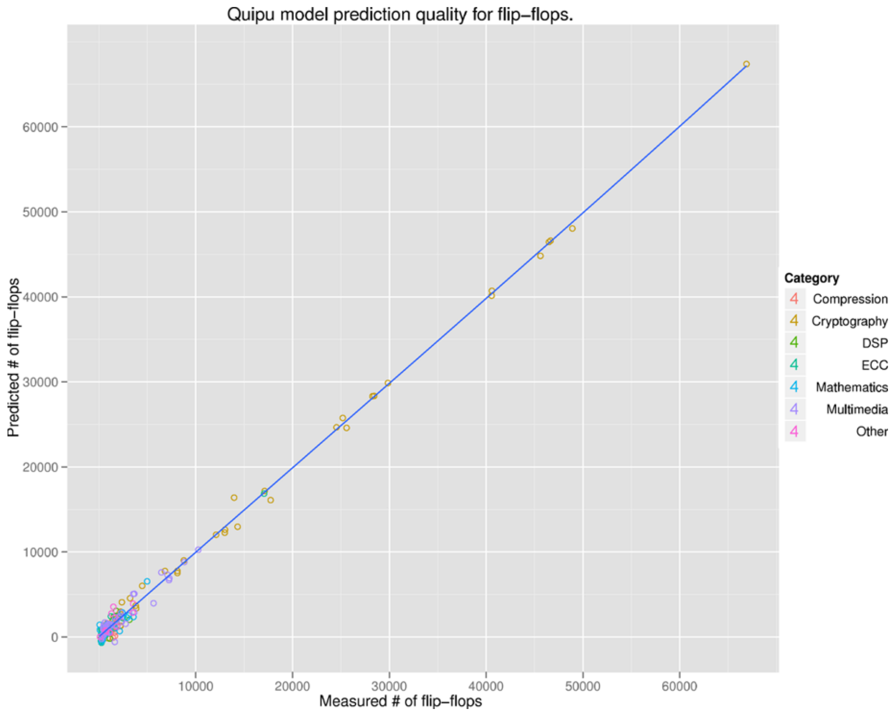


Fig. 2.9 Predicted versus measured number of flip-flops on a Virtex-4 LX200 using the Quipu prediction model. These are validated predictions using leave-one-out validation

2.6 Application Partitioning

MultiProcessor systems are becoming common, not only in the high performance segment, but also in the consumer and embedded markets. Developing programs for these architectures is not easy: the developer needs to correctly decompose the application in order to enhance its performance and to exploit the available multiple processing elements.

There already exists a set of approaches which aim at developing hardware-software co-design tools that also partially address some of the issues relevant in this research.

Some prominent examples from academia are the COSMOS tool [22] from TIMA laboratory, SpecC [62] from the UC Irvine, or the Ptolemy environment [7] and the Polis/Metropolis [2] framework from the UC Berkeley. There are also some commercial products, such as CoWare's ConvergenSC, N2C [6], and the meanwhile discontinued VCC [46] environment from Cadence. These approaches reveal significant drawbacks since they mainly restrict the design exploration to predefined library-based components and focus on simulation and manual refinement.

Many of the proposed methodologies for parallelism extraction have been implemented by reusing an existing compiler framework and not by creating a new one,

even if this fact could limit the potential of the proposed approach. For example Jin et al. extend CAPTools [21] to automatically generate OpenMP directives with one [23] or two [24] levels of parallelism; the implementation of their methodology can only be applied to Fortran 77 because this is the only code language that CAPTools can take as input.

Banerjee et al. [3] present a valid overview of the different techniques adopted for parallelization both at the instruction and coarse grained levels.

Most research works, dealing with partitioning of the initial specification, adopt specific intermediate representations. Task graphs are used as intermediate representation by various methodologies that aim at extracting parallelism and consider target architectures different from Shared Memory Multiprocessors. For example Vallerio and Jha [50] propose a methodology for building Task Graphs starting from C source code. These task graphs are then used as input for HW/SW co-synthesis tools that do not necessarily address a fork-join programming model, so they have fewer restrictions in the construction of the Task Graphs.

Girkar et al. [14] propose an intermediate representation, called Hierarchical Task Graph (HTG), which encapsulates minimal data and control dependences and which can be used for extraction of task level parallelism.

Similarly to [14] our intermediate representation is based on loop hierarchy even if we use a more recent loop identification algorithm [43]. Moreover, as in [14] and in [11] we use data and control dependences (commonly named Program Dependency Graph—PDG) as defined in [11] but we target the explicit fork/join concurrency model. In fact, since they do not use this model to control the execution of tasks their work mainly focuses on the simplification of the condition for execution of task nodes (they define a sort of automatic scheduling mechanism).

Luis et al. [33] extend this work by using a Petri net model to represent parallel code, and they apply optimization techniques to minimize the overhead due to explicit synchronization.

Franke et al. [12] try to solve the problems posed by pointer arithmetic and by the complex memory model on auto-parallelizing embedded applications for multiple digital signal processors (DSP). They combine a pointer conversion technique with a new modulo elimination transformation and they integrate a data transformation technique that exposes to the processors the location of partitioned data. Then, thanks to a new address resolution mechanism, they can generate programs that run on multiple address spaces without using message passing mechanisms.

Newburn and Shen [35] present a complete flow for automatic parallelization through the PEDIGREE compiler; this tool is targeted to Symmetric Multi-Processor Systems. They work on assembly code thus their tool can exploit standard compiler optimizations. They are also independent of the high level programming language used to specify the application. The core of PEDIGREE works on a dependence graph, very similar to the CDG graph used by our middle end; this graph encodes control dependences among the instructions. Parallelism is extracted among the instructions in control-equivalent regions, i.e. regions predicated by the same control condition. Applying PEDIGREE to the SDIO benchmark suite, the authors show an average speed-up of 1.56 on two processors.

The work proposed in [39] focuses on extracting parallelism inside loops: each extracted thread acts as a pipeline-stage performing part of the computation of the original loop; the authors fail to specify how synchronization among the threads is implemented. By considering only loops inside benchmarks, a speed-up of 25% to 48% is obtained.

Much work on thread decomposition has been done also for partitioning programs targeted to speculative multiprocessor systems. Speculative multiprocessors allow the execution of tasks without absolute guarantees of respecting data and control dependences. The compiler is responsible for the decomposition of the sequential program in speculative tasks. Johnson et al. [25] propose a min-cut-based approach for this operation, starting from a Control Flow Graph (CFG) where each node is a basic block and each edge represents a control dependence. In their implementation the weights of every edge are changed as the algorithm progresses in order to account for the overhead related to the size of the threads being created.

The clustering and merging phases have been widely researched. Usually, these two phases are addressed separately. Well known deterministic clustering algorithms are Dominant Sequence Clustering (DSC) by Yang and Gerasoulis [60], linear clustering by Kim and Browne [27] and Sarkar's Internalization Algorithm (SIA) [44]. On the other hand, many researches explore the cluster-scheduling problem with evolutionary algorithms [16, 53]. A unified view is given by Kianzad and Bhattacharyya [26], who modify some of the deterministic clustering approaches by introducing probability in the choice of elements for the clusters; they also propose an alternative single step evolutionary approach for both the clustering and cluster scheduling aspects.

The approach proposed in hArtes is very similar to the one presented by Newburn and Shen: GNU/GCC is used to generate the input to the partitioning tool thus there is no need to re-implement standard compiler optimizations. In addition to concentrating on extracting parallelism inside control-equivalent regions, several Task Graph transformations have been implemented to improve efficiency and to adapt it to OpenMP needs. In fact, as previously mentioned, we consider the explicit fork/join model of concurrency efficiently implemented by the OpenMP standard; this concurrency model does not require to explicitly define the conditions for which a task can execute.

2.6.1 Partitioning Toolchain Description

The Task Partitioning tool is named **Zebu** and it is developed inside PandA, the HW/SW co-design framework shown in Fig. 2.10 and currently under development at Politecnico di Milano. It aims at identifying the tasks in which the application can be decomposed to improve its performance. It also proposes an initial mapping solution to be refined by the mapping tool.

In particular, the requirement analysis shows that the problem can be clearly separated into two distinct sub-problems:

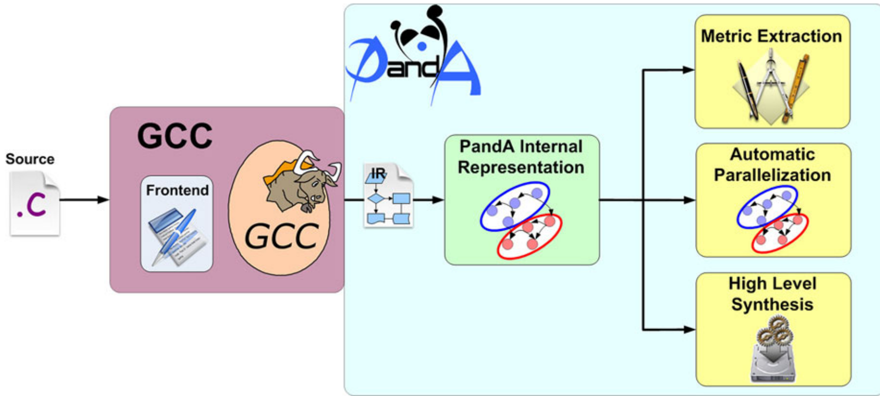


Fig. 2.10 PandA framework

1. in the first phase the tool identifies the parallelism in the application, i.e. it annotates, using task partitioning pragma notations, how it can be decomposed in separate threads, presenting the parallel tasks within the annotated code;
2. in the second phase the tool indicates, using specific mapping pragma annotations, an initial guess concerning which processing element (GPP, DSP or programmable logic) should execute the identified tasks, i.e. where each task should be mapped in the system. The pandA automatic parallelization overview is shown in Fig. 2.11.

The input of the tool is the C source code of the application and the XML file containing information about the target architecture and available data of the performance of the different parts of the application on each processing element. It also contains information about library functions and the header files where they are defined. It can also represent a feedback from the toolchain, that can improve the accuracy of the performance estimations and therefore the quality of the solution identified by the Zebu tool.

The output of Zebu is a C source code annotated with pragmas representing the task partitioning and the mapping suggestions. Note that the tasks identified by the tool will be represented as new functions. Moreover, it also reproduces the XML file, augmented with these new functions. It is worth noting that the internal performance estimations are not reported into the output XML since they are almost related to the internal representation of the tool and, thus, they could not exploited by the rest of the toolchain.

This tool behaves as a compiler in that it is composed of

- the **frontend**, which creates the intermediate representation of the input code,
- the **middle-end**, an internal part which manipulates the intermediate representation, creates an efficient partitioning also exploiting internal performance estimation techniques and suggesting a initial guess of mapping, and
- the **backend**, which prints the executable C code annotated with OpenMP [45] and mapping directives.

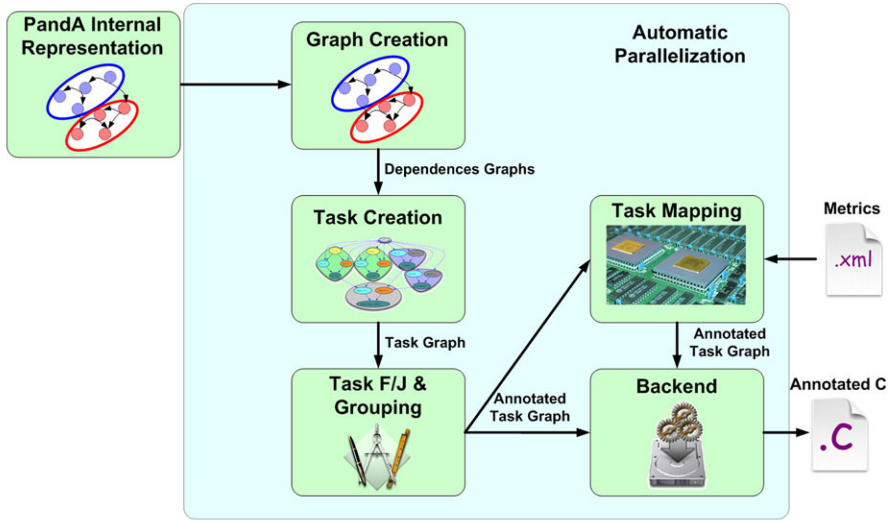


Fig. 2.11 PandA automatic parallelization

Frontend

The frontend does not directly read C source code, but it invokes a slightly modified version of the GNU/GCC compiler and parses the resulting GIMPLE compliant tree-representation. In this way it is possible to control the code optimizations performed by this compiler, avoiding to re-implement them into Zebu. Note that, the analysis performed by Zebu can only take into account target-independent optimizations, such as constant folding, constant propagation or dead code elimination, that have a direct impact on the GIMPLE representation. In fact, since the different parts of the application could be executed by any of the processing elements, based on mapping decisions, target-dependent optimizations cannot be considered at this stage. These optimizations can be enabled or disabled directly passing to Zebu the same optimization flags that would be given to the GNU/GCC compiler. Besides the code optimizations, each original C instruction is converted by GNU/GCC in one or more basic operations and the loop control statements `while`, `do while` and `for` are replaced with `if` and `goto`, and have to be reconstructed in the following.

Parsing: since the Zebu frontend does not directly parse the source code, this part is mainly composed of a wrapper to an internally modified version of the GNU/GCC 4.3 compiler, that exploits the Single Static Assignment (SSA) form as representation. Currently, only few patches have been applied to the original GCC version and most of them concern how the tree, which holds the GCC internal representation, is written to file: what has been done is to make GCC dump, in its debug files, as much useful information as possible. This was necessary since part of its intermediate representation (virtual operands [36]—used for tracking dependencies among aggregate variables and for serialization purposes—, the structure of

```
b[1] = 0
a = b[1] + c;
b[1] = a - 1;
c = d + a;
if (c > 0)
{
   d = b[1] * c;
}
else
{
   d = b[1] - c;
}
b[1] = 2;
printf("%d", d);
```

(a)

```
A:  b[1] = 0;
B:  a_1 = b[1] + c_1;
C:  b[1] = a_1 - 1;
D:  c_2 = d_1 + a_1;
E:  if (c_2 > 0)
    {
F:     d_2 = b[1] * c_2;
    }
    else
    {
G:     d_3 = b[1] - c_2;
    }
H:  d_4 = phi(d_2, d_3);
I:  b[1] = 2;
L:  printf(%d, d_4);
```

(b)

Fig. 2.12 Example of program in original format (**a**) and in the corresponding SSA-form (**b**)

the basic block control flow graph, operands of some special operators, etc.) wasn't originally printed out.

GCC is thus invoked to produce the ASCII file containing the intermediate representation. We chose to dump the intermediate tree representation after as many optimizations as possible have been performed on it by GCC. Ideally the last created tree, after all GIMPLE optimizations are applied to it, should be used. However the tree that satisfies our needs is the tree dumped after the last phase of the phi-optimization since it is still in SSA-form [37]. The SSA form is based on the assumption that program variables are assigned in exactly one location in the program. Multiple assignments to the same variable create new versions of that variable (see Fig. 2.12 for an example). Naturally, actual programs are seldom in SSA form initially, thus the compiler modifies the program representation so that every time a variable is assigned in the code, a new version of the variable is created. Different versions of the same variable are distinguished by subscripting the variable name with its version number. Variables used in the right-hand side of expressions are renamed so that their version number matches that of the most recent assignment. SSA-form is used in the GCC GIMPLE tree both for scalar and aggregated variables, but not in the same way: scalar SSA variables are used directly as operands of GIMPLE operations replacing original operands, while aggregated SSA variables are only used in annotation of GIMPLE statements to preserve their correct semantic and they will be referred as *virtual operands*.

The optimizations that GCC performs on the input code are controlled by our tool for mainly three reasons:

1. To evaluate the improvement of every single optimization on the parallelization flow;
2. Some optimizations could reduce the available parallelism; an example is the common subexpression elimination. In fact, suppose there are two code fragments which do not have any inter-dependences but have a common subexpression; at the end of the partitioning flow it is possible that these two segments

are assigned to two different parallel tasks. This is true only if common subexpression elimination is not applied. Otherwise, the two segments would share the same subexpression: in this way an inter-dependence has been created and this prevents the two segments from being assigned to parallel tasks.

3. Some optimizations and the precision of some analyses are parameterized. For example it is possible to change the size threshold which controls if a loop should be unrolled or not or it is possible to change the maximum number of virtual operands used during alias analysis. Even if these parameters could impact significantly on GCC compilation time, it should be considered that this time is usually only a little fraction of the total time spent in performing the parallelization, so direct impact of these changes on execution time could be ignored. Nevertheless, the choice of these parameters impacts indirectly on the total execution time of the tool since it could change very much the size of the data on which the tool works; these effects on timing must be taken into consideration when deciding which optimizations have to be enabled. For example, loop unrolling increases the number of statements of the specification. In this way there is an obvious trade-off between total execution time and analysis precision; this means a trade-off between the transformation time and quality of the results (in terms of amount of parallelism which could be extracted and therefore in terms of speed-up of the produced code).

After parsing the GIMPLE-compliant tree-representation, two different collections of graphs are built for each function of the original specification. In the first one, a vertex corresponds to a basic block, as defined by GCC itself. In particular, according to this definition, a function call does not start a new basic block. In the second one, instead, a Control Flow Graph (CFG) is created for each C function. Every node of these graphs corresponds to one simple GIMPLE operation and can represent zero, one or more C basic operations. Phi-nodes of real variables or of virtual operands are treated as if they were statements, since, in this way, the sequential behavior can be easily maintained. In the first type of graphs an edge represents dependences or precedences between two basic blocks, while in the second type precedences or dependences between two operations. The type of dependency or of precedence depends upon which particular graph inside the collection is considered. During this preliminary phase, the operations inside each basic block are also annotated with some of the information present in the GCC tree such as the real and virtual operands read and written by each operation (as shown in Fig. 2.13).

A first analysis of the specification is also performed and its results are annotated onto the graphs. This analysis basically retrieves information about the type of each operation and which variables are read/written by each of them. Moreover at the end of this phase, Zebu analyzes the produced CFGs to identify those edges that close a cycle in a path starting from the entry node (entry node is a symbolic node which represents the beginning of the computation flow into each function). The results of these analyses allow Zebu to correctly identify and rebuild cycle constructs in the following phases.

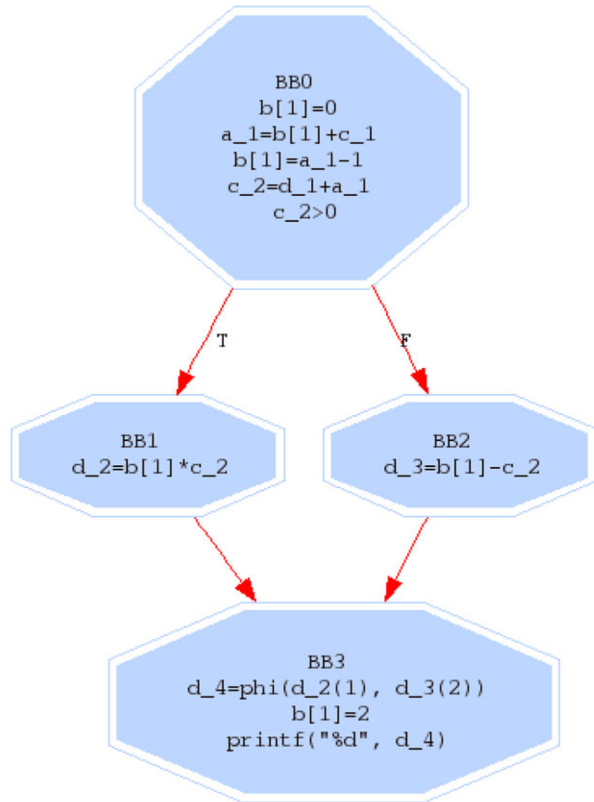


Fig. 2.13 Example of Control Flow Graph of basic blocks

Middle-End

As in traditional compilers, the middle-end can be considered the main part of the compilation flow implemented in Zebu. This phase is mainly composed of three steps:

- the dependence analysis and the manipulation of the intermediate representation;
- the performance estimation and partitioning;
- the mapping onto the target platform.

Dependence Analysis and Manipulation of the Intermediate Representation The dependence analysis step consists of the analysis of the Control Flow Graphs and of the tree to compute all the dependences between each pair of operations (nodes). Dependences between an operation A and an operation B can be basically of one or more of the following types:

- **Control Dependence**: execution of operation B depends on the result of operation A or operation B has to be executed after operation A;
- **Data Dependence**: operation B uses a variable which is defined by operation A;

- **Anti-Dependence**: operation B writes a variable, which is previously read by operation A. Even if we exploit the Static Single Assignment (SSA) form, these dependences can still occur between pointer accesses and they are necessary to maintain the correct semantic of the application;
- **Feedback Dependence**: operation B depends on the execution of A in the previous iteration of the loop which both the operations belong to.

Additional graphs are thus built to represent these dependences. All these graphs are produced into two different versions, i.e., with or without feedback dependences.

These graphs are produced for each C function of the original specification. To extract as much parallelism as possible, this dependence analysis must be very precise. A false dependence added between two operations indicates that they cannot be concurrently executed, so it eliminates the possibility of extracting parallelism among those instructions. On the other hand, all the true dependences have to be discovered otherwise the concurrent code produced by Zebu could have a different functionality from the original one.

Before computing data dependences and anti-dependences, alias analysis has to be performed: this is necessary to correctly deal with pointers. An inter-procedural alias analysis model is used. In this way less conservative results than those produced by intra-procedural methods are obtained, despite an overhead in computation time.

In addition to the analyses of the original specification, Zebu also partially manipulates the intermediate representation by applying different transformations. For example it applies further dead code elimination made possible by alias analysis and by loop transformations, such as loop interchange.

Dependence extraction: loop detection is the first analysis performed. The Control Flow Graph of basic blocks is analyzed to detect loops which are classified into reducible or irreducible ones. A loop forest is also built using a modified version of the Sreedhar-Gao-Lee algorithm [43]. Then the Control Flow Graph of GIMPLE operations is built from the Control Flow Graph of Basic Blocks simply by creating sequential connections among the operations inside each basic block and among the operations ending and starting two connected basic blocks. The next step is the creation of the dominator and post-dominator tree of the Control Flow Graph of the Basic Blocks. In particular the second of these trees is used to compute the **Control Dependence Graph** (CDG) of Basic Blocks from which the Control Dependence Graph of operations is easily built. Edges in this class of graphs could be of two types: normal edges or feedback edges. The latter indicates which operations control the execution of the iterations of a loop execution. Moreover, each edge in the CDG is annotated with the condition on which a conditional construct operation controls other operations. Values of these labels can be "true" or "false" for edges going out conditional constructs of type if, or a set of integers for edges going out from switch constructs (each integer represents a case label). In this way all information about control dependences between each pair of operations is extracted and annotated; this information will be used by the backend to correctly write back the intermediate representation to C code.

After the control dependences analysis, the tool starts the computation of data dependences. This is based on the information of variables in SSA-form read and written by the different GIMPLE statements, annotated in the corresponding nodes of the Control Flow Graph. For each pair of definition and use of a variable in SSA-form (either real or virtual), an edge in the **Data Dependence Graph** is added starting from the node of operation which defines the SSA variable and going to the target node where the variable is used. Since also virtual operands are considered, in some cases, additional data flow edges are inserted even if there are no real data dependences. This happens, for example, when a fixed ordering between pair of operations is required to preserve the semantic of the original C source code. For example, the order of the `printf` calls between all functions of a program has to been maintained to force the output of our produced code to be the same of the original one. In this case the serialization of these function calls is forced by adding reading and writing of special fake variables in all corresponding nodes with the same purpose of GCC with *virtual operands*.

Thanks to the fact that scalar SSA-form variables are directly embedded into the GIMPLE code, if different SSA versions of the same scalar variable are dumped back in the produced source code, they are actually different variables thus the anti-dependences (dependences of type write after read and write after write) can be ignored during reordering of the operations: this type of dependences never occurs in pure SSA code. On the other hand, dumping back aggregated variables in SSA-form is more complex; the reasons are the impossibility of writing back directly the phi-nodes of array of variables and the difficulty in managing situations where pointers could point to different variables (so operations could represent define of multiple virtual operands). We decided to maintain the original formulation of GIMPLE operations in all cases: scalar variables will be printed out considering their SSA version (each SSA form of a variable is treated as a different variable), while aggregated variables are printed back using base form and using the SSA-form of virtual operands only to track the dependences. As a consequence, simple data dependences of type read after write are not enough to describe all possible correct orderings between the operations. Consider for example the fragment of code shown in Fig. 2.12(b) and suppose that no optimizations have been performed by GCC. In this case, according to the previous definition, the assignments F and G use the virtual definition of the element of the array defined in the assignment C; then the assignment I redefines that definition.

Hence, according to simple data dependences, it could be possible to execute the assignment I before the F and G ones, that results in a wrong run-time behaviour. In fact, it could happen that, in the following phases of the partitioning flow, these two assignments are clustered into different and parallel threads and so they could be executed in whatever order. This simple example proves that data flow analysis based only on read after write dependences is not sufficient to catch all the actual dependences between operations. For this reason we build the **anti-dependence graph** to take correctly into account all the dependences. As in the previous graph computations, this type of dependences can be derived from the GCC GIMPLE tree. In fact, GIMPLE annotates a virtual definition of a variable with an additional

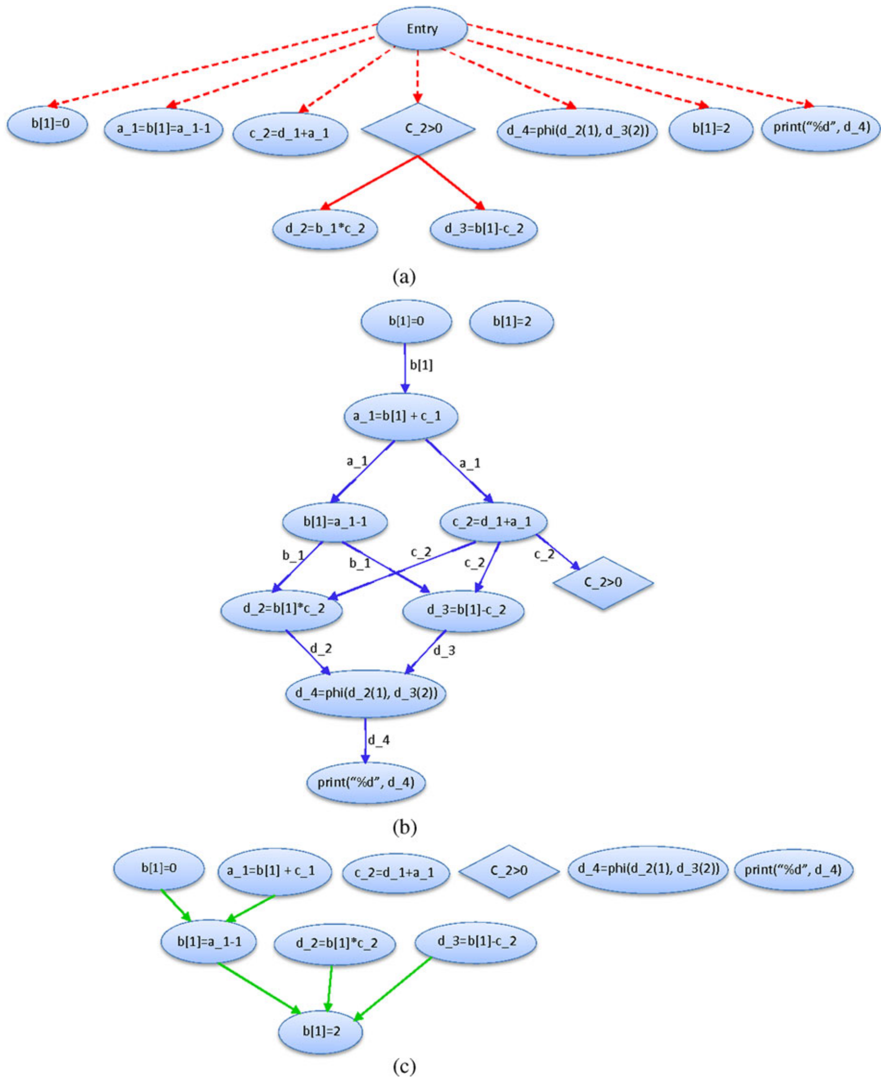


Fig. 2.14 Control dependence graph (**a**), data dependence graph (**b**), anti-dependence graph (**c**)

operand that is the related *killed* definition. In this way, in the previous example, the assignment I is annotated with the information on the killed definition, that is the one defined by the assignment C and used by the F and G ones. Therefore anti-dependence edges can be added between the C and the I assignments (write after write dependence) and between F (and G) and the I (write after read dependence) ones. In this way, the correct behavior can be preserved (see Fig. 2.14).

The next phase of our flow tries to exploit the scalar replacement of aggregates techniques presented in [5] by applying it to the data dependences analysis just performed. The GCC *SRA* optimization is not applied directly because it causes an

increase of the size of the produced source code (indeed it is applicable only to arrays with very few elements) and it reduces its readability. Moreover, in this way it is also possible to select where this replacement is performed: for example this technique could be applied only in critical points where the elimination of some false dependences (added by the compiler in a conservative approach) could help to extract more parallelism.

Combining the Data Dependence, the Anti-Dependence and the Control Dependence Graph, we build the Program Dependence Graph which will be used in the following phases of the task partitioning process (see Fig. 2.15).

Parallelism Extraction This phase aims at the division of the created graphs into subsets, trying to minimize the dependences among them; hence this phase is often identified as the partitioning phase.

The first step consists in the analysis of the feedback edges in order to identify the loops and separate each of them from the other nodes of the graph; from now on, the partitioning steps will separately work on each of the identified subgraphs: parallelism can be extracted either inside a loop, between loops or by considering the nodes not part of any loop (see Fig. 2.16).

After computing the loop subgraphs, the core partitioning algorithm is executed (refer to Fig. 2.17).

In a similar way to what performed in [35] the algorithm starts by examining the control edges to identify the control-equivalent regions: each of them groups together the nodes descending from the same branch condition (*true* or *false*) of predicated nodes; nodes representing switch statements are treated in a similar way. Data dependences and anti-dependences among the nodes inside each control-equivalent region are now analyzed to discover intra-dependent subgraphs: all the elements inside such subgraphs must be serially executed with respect to each other.

The analysis starts from a generic node in a control-equivalent region with a depth-first exploration. A node is added to the cluster being formed if it is dependent from one and only one node or if it is dependent from more than one node, but all its predecessors have already been added to the current cluster. Otherwise, the cluster is closed and the generation of a new set starts. These operations are iterated until all the nodes in the control-equivalent partition are added to a set.

Each obtained "partition" (subgraph) represents a single block of instructions, with none, or minimal interdependence. Partitions that do not depend on each other contain blocks of code that can potentially execute in parallel. Edges among partitions express data dependences among blocks of code, thus the data represented by in-edges of a partition must be ready before the code in that partition can start. Note that the identified partitions are just a first approximation of the tasks into which the input program is being divided.

The result of this partitioning is thus represented by Hierarchical Task Graphs (HTGs). HTGs introduce the concept of hierarchy, and provide an easy representation of cyclic task graphs and function calls.

Fork/Join Task Creation and Task Grouping Since OpenMP [45] is used to annotate the parallelism in the produced C specification, transformations to the

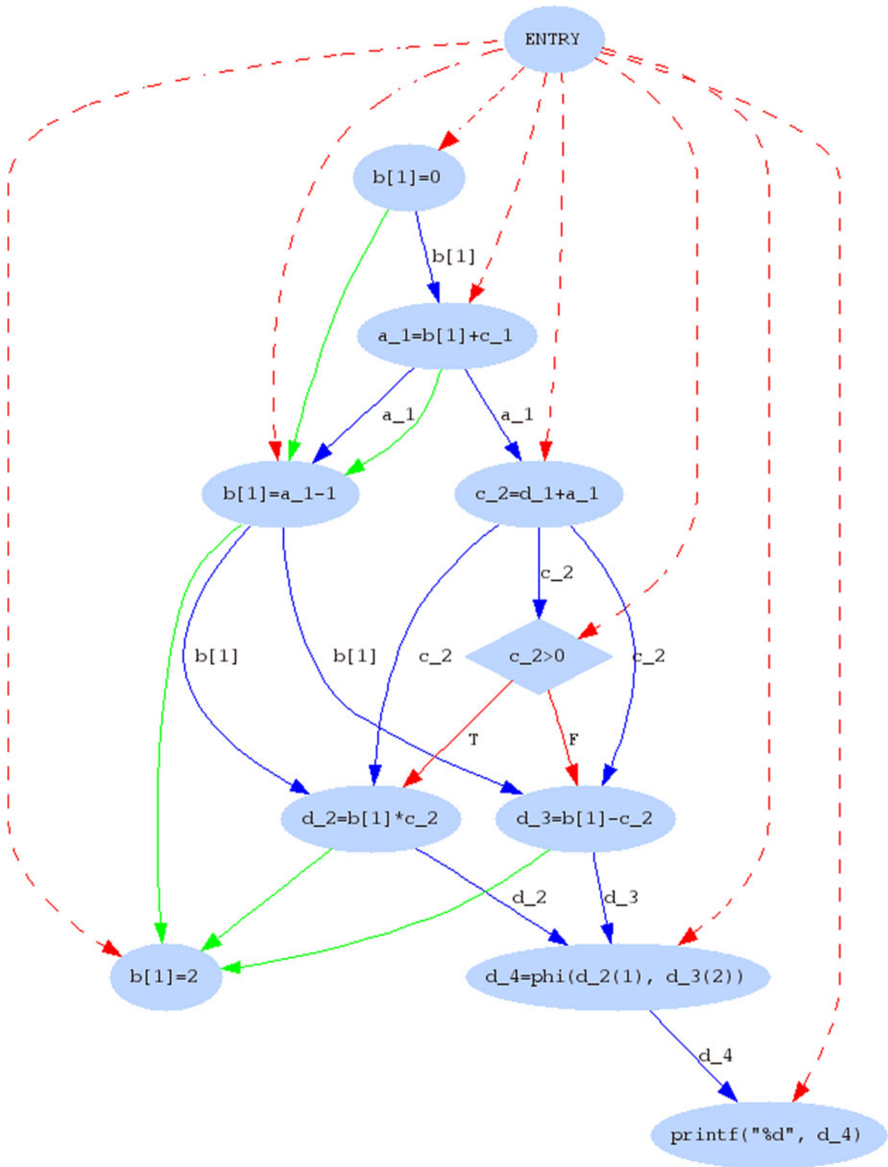


Fig. 2.15 Program dependence graph

task graph are necessary in order to make it suitable for the OpenMP programming model, that is the Fork/Join programming model. This programming model requires that each task spawning threads (called *fork* task) has a corresponding *join task*, which can be executed only after all the created threads have completed their execution. Figure 2.18 shows the leftmost task graph compliant with the fork/join programming model and the one on the right not compliant.

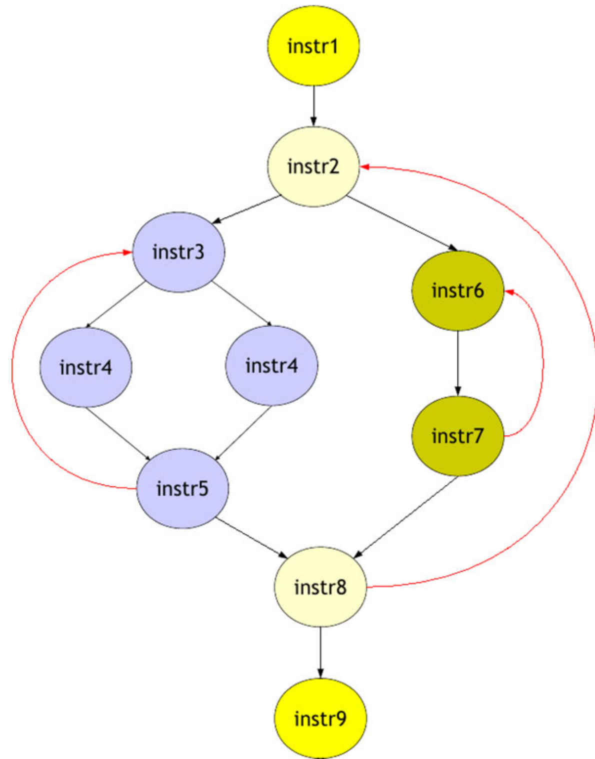


Fig. 2.16 Nodes belonging to different loops are clustered in different subgraphs

The algorithm which transforms a generic task graph into one compliant with the fork/join programming model is composed of two main phases, which are iterated until the whole graph is compliant:

1. identification of the non fork/join compliances; they are always discovered in correspondence of join tasks (node 5 in the example);
2. reconstruction of the corrected task graph, that satisfies the tasks' dependences but with a compliant structure.

Experiments show that the tasks, created with the presented algorithm, are usually composed of a limited number of instructions: on most systems (even the ones containing a lightweight operating systems) the overhead due to the management (creation, destruction, synchronization) of these small tasks could be higher than the advantages obtained by their concurrent execution.

The **optimization** phase (task grouping) tries to solve this problem by grouping tasks together. Two different techniques are used: optimizations based on control dependences and optimizations based on data dependences.

Control statements, such as `if` clauses, must be executed before the code situated beneath them, otherwise we would have speculation: it seems, then, reasonable to group together the small tasks containing the instructions which depend on the same control statement. Another optimization consists of grouping the `then` and

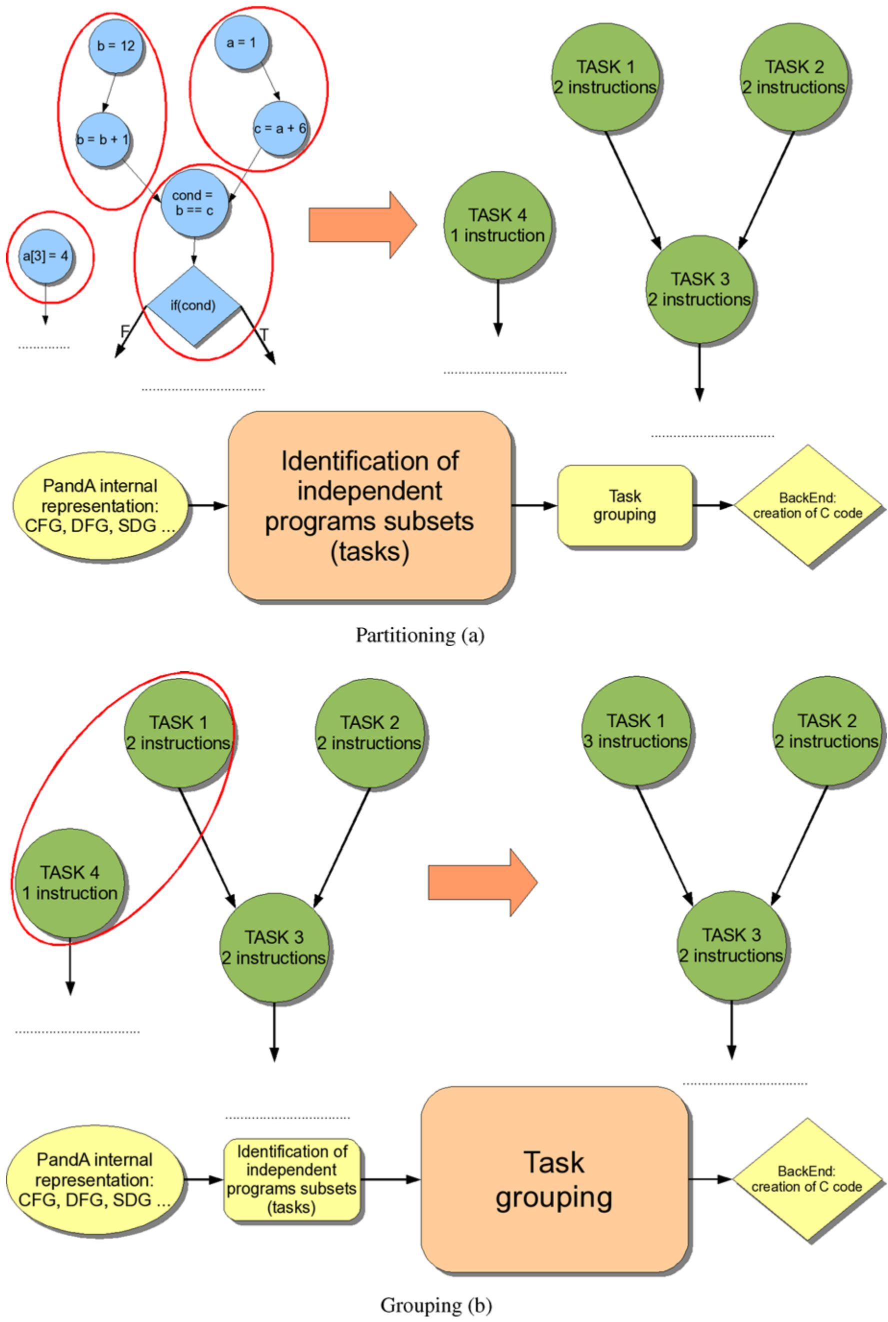
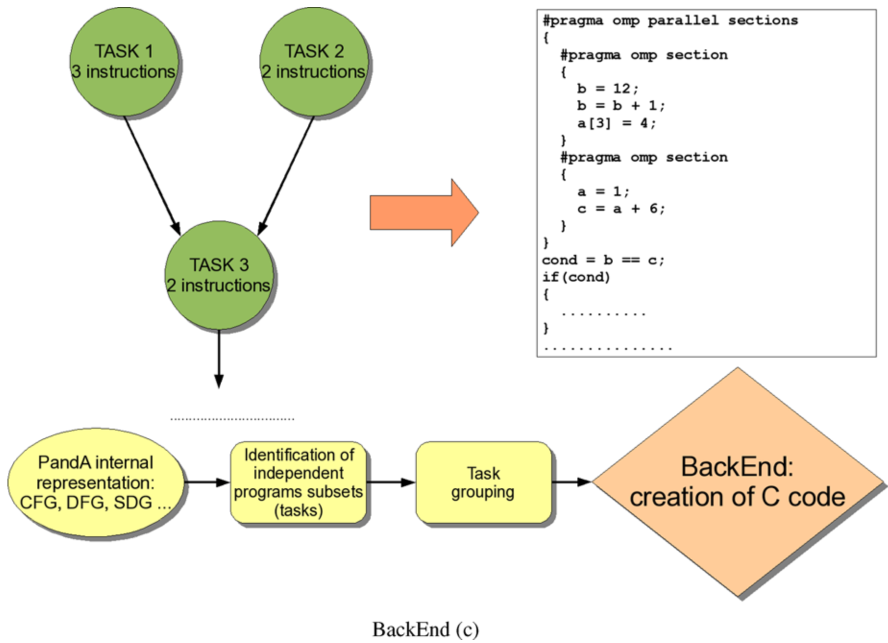


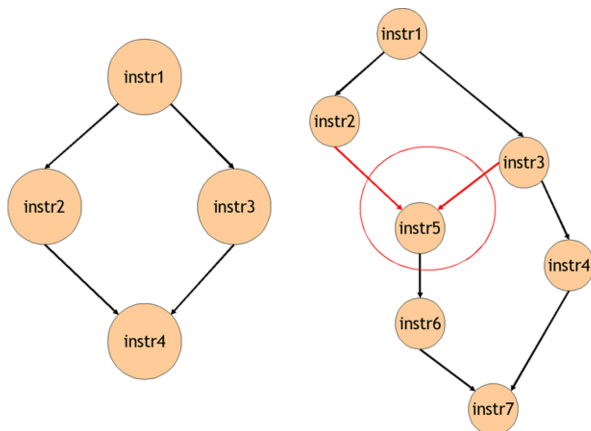
Fig. 2.17 Phases of the parallelism extraction algorithm



BackEnd (c)

Fig. 2.17 (Continued)

Fig. 2.18 A task graph compliant with the fork/join programming model (on the *left*) and one not compliant



`else` clauses in the same cluster: they are mutually exclusive, the parallelism is not increased if they are in separate clusters.

Data dependent tasks can be joined together when their weight is smaller than a predetermined number n. Those tasks, which are also neighbors of the same fork node, are joined together; if all the tasks of the fork group are to be joined together, then the whole fork group disappears and all its nodes (including the fork and join ones) are collapsed in the same task.

Besides the optimization with respect to the fork-join compliance also the estimated target platform overhead is considered. In addition to parallel tasks, sequential ones are also created in order to exploit the heterogeneity of the architecture and obtain an overall speed-up of the application.

Initial Mapping of the Application onto the Target Platform Different approaches have been proposed for mapping the application onto the target platform. In particular, a heuristic optimization methodology that has recently gained interest in dealing with such class of problems for its performance and the quality of the results is the Ant Colony Optimization (ACO) approach.

Starting from the HTG intermediate representation, we apply an Ant Colony Optimization (ACO) approach that explores different mapping combinations. Differently from previous approaches, the proposed solution iteratively builds different combinations of mapping, evaluating the global effects and identifying the decisions that improve the overall performance of the application. Furthermore, it allows the share of the information at the different levels of the hierarchy and it is able to support a large set of constraints that can be imposed by the target architecture, such as the available area for the hardware devices.

We compared [10] this solution with traditional techniques for mapping parallel applications on heterogeneous platforms and the results are shown in Table 2.2. In details, we compared the execution time needed to generate the solution (Cpu column) for the different approaches, namely the proposed ACO, the Simulated Annealing (SA) and the Tabu Search (TS). The quality of the results, in terms of execution time of the emulation platform, has also been reported for the proposed approach, along with the percentage difference for the other solutions with respect to this one. A solution obtained with a dynamic scheduling and allocation has also been reported to demonstrate the need for such methodology on heterogeneous platforms.

Note that the proposed approach systematically outperforms existing methodologies by at least 10% in average and it is also much faster than the other approaches to converge to a *stable* solution, as represented by the differences in terms of computation time.

Backend

In the last phase, **Zebu** needs to produce a **parallel executable C** program containing the results of the partitioning and of the initial mapping. In particular, the code is annotated with hArtes pragma notations that express the partitioning in a set of tasks and, for each task, the target processing element suggested for the execution. In the same way, the other annotations, contained in the original specification, are reproduced without modifications in the produced code.

As described in the previous paragraph, in order to be able to use this library and address the MOLEN paradigm of execution, the task graph must strictly adhere to the fork/join programming model and the OpenMP formalism. For this reason, during the production of the concurrent C code, the backend has to perform the following tasks:

Table 2.2 Comparison of results of Ant Colony Optimization (ACO), Simulated Annealing (SA) and the Tabu Search (TS), along with a dynamic policy

Benchmarks	ACO				SA		TS		Dynamic scheduling
	Proposed approach	Cpu (s)	Only mapping	Only scheduling	Mapping scheduling	Cpu (s)	Mapping scheduling	Cpu (s)	
sha	1.72 msec	4.20	2.28	12.14%	8.23%	5.18	6.71%	7.11	29.44%
FFT	13.41 sec	8.12	103.57	108.38%	31.11%	11.89	27.84%	17.20	257.21%
JPEG	0.46 sec	10.67	0.12	5.15%	1.13%	14.63	4.57%	13.07	27.64%
susan	9.31 sec	6.08	0.15	21.96%	4.41%	7.16	7.58%	9.18	21.30%
adpcm coder	1.42 msec	0.20	0.15	7.08%	9.10%	0.22	4.33%	0.25	7.08%
adpcm decoder	1.76 msec	0.19	0.05	4.65%	9.24%	0.23	8.96%	0.21	5.56%
bitcount	0.15 sec	0.10	1.12	1978.77%	11.02%	0.10	14.12%	0.11	2024.77%
	1.14 sec	0.34	0.07	178.07%	35.30%	0.62	29.89%	0.58	178.77%
rijndael	0.81 sec	2.58	2.12	6.30%	3.12%	3.36	1.01%	4.32	3.40%
	8.36 sec	2.72	2.02	6.20%	0.09%	2.91	0.74%	3.10	3.39%
Avg. difference			+11.17%	+232.87%	+11.28%	+27.53%	+10.58%	+45.21%	+255.86%

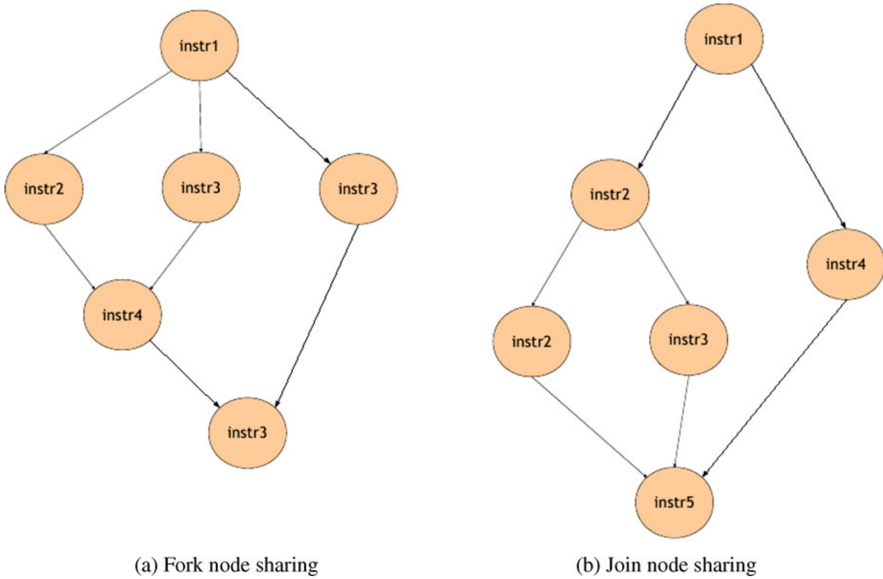


Fig. 2.19 Fork (**a**) and Join (**b**) node sharing

1. identification of all the fork/join groups; note that different groups may share the fork node (Fig. 2.19.a) or the join node (Fig. 2.19.b);
2. identification of the task functions to be created and the related parameters that have to be given to them;
3. printing of the C code that respects the original semantic of the application.

Finally, the resulting C code is provided to the rest of the toolchain for the decision mapping phase.

Note that not only phi-nodes of virtual operands, but also phi-nodes which define scalar SSA-form variables, are not printed back. We could print back them as assignments with "conditional expression", but in this way a significant overhead to the application execution occurs; moreover, this technique is difficult to be used as it is for phi-nodes with more than two *uses*. For these reasons a phi-node of a scalar variable is replaced by as many assignments as the *variable uses* present in that node, placed immediately after the related variable definition. For example, suppose you have a phi-node `a_1 = PHI <a_4(3), a_5(4)>`, where `a` is a variable and the suffix `_<index>` identifies its SSA version; note that the number inside the parentheses represents the basic block where the definition occurs. According to SSA-definition, variable `a_4` should be defined only in one statement of the function and according to phi-node definition, this statement should be mutually exclusive with the others in the same phi-node. Therefore, the assignment `a_1 = a_4` can be added after the statement that defines `a_4`, that will be in the basic block numbered as 3. Then the same procedure is repeated for definition of variable `a_5`, that will be in the basic block numbered as 4. In this way phi-nodes are not necessary anymore and the consistency of the sequential model is maintained.

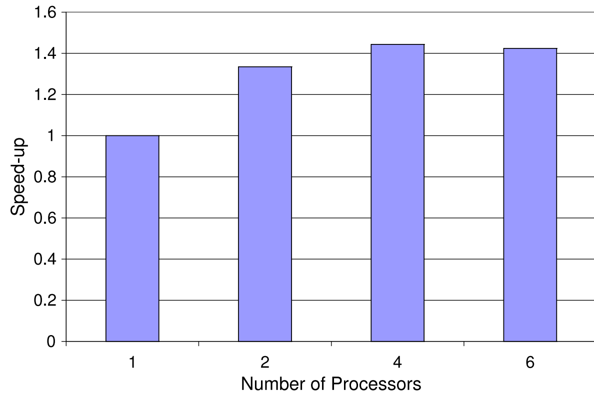


Fig. 2.20 Parallelization of JPEG encoder

Case Studies: The JPEG Encoder and the DRM Decoder

In this section, we analyze two different case studies for our partitioning methodology: the JPEG encoder and the core of the Digital Radio Mondiale (DRM) decoding process, that was one of the benchmark applications proposed in the hArtes project.

The kernel of the JPEG encoder contains a sequence of data transformations applied to the raw image, as shown on the left side of Fig. 2.20: Color Space Transformation, Downsampling, Block Splitting, Discrete Cosine Transformation, Quantization and Entropy Encoding. Among these phases, Color Space Transformation (RGBtoYUV), Discrete Cosine Transformation (DCT) and Quantization are the most computationally intensive. The analysis performed by Zebu is able to identify some Single Instruction Multiple Data parallelism in the first two functions. Then, after applying the different transformations to reduce the overhead, Zebu produces a hierarchical task graph representing which parts can be executed in parallel, as shown on the right side of Fig. 2.20. The task graph of the RGBtoYUV function contains a single parallel section with four parallel tasks. The task graph of the DCT function has a similar structure, but with only three tasks. This structure is then reproduced in the output C code through the corresponding OpenMP pragmas compliant with the hArtes guidelines.

The DRM refers to a set of digital audio broadcasting techniques designed to work over shortwave. In particular, different MPEG-4 codecs are exploited for the transmission and the core of this decoding process is mainly composed of the Viterbi decoder. Exploiting GNU/GCC optimizations, Zebu is able to identify and extract some data parallelism from the proposed implementation of the algorithm, as shown in Fig. 2.21. A task graph with four parallel tasks describing the introduced parallelization is then reproduced annotating the output C code through OpenMP as well.

Note that, in both the cases, the parallelism identified among loop iterations is expressed through `omp parallel sections` instead of `omp parallel for` pragmas. In fact, in the hArtes toolchain, the different tasks have to be represented as functions to support the mapping that is performed at compile time and not during the execution of the code. For this reason, we explicitly represent the partitioning of the loop as `omp sections`.

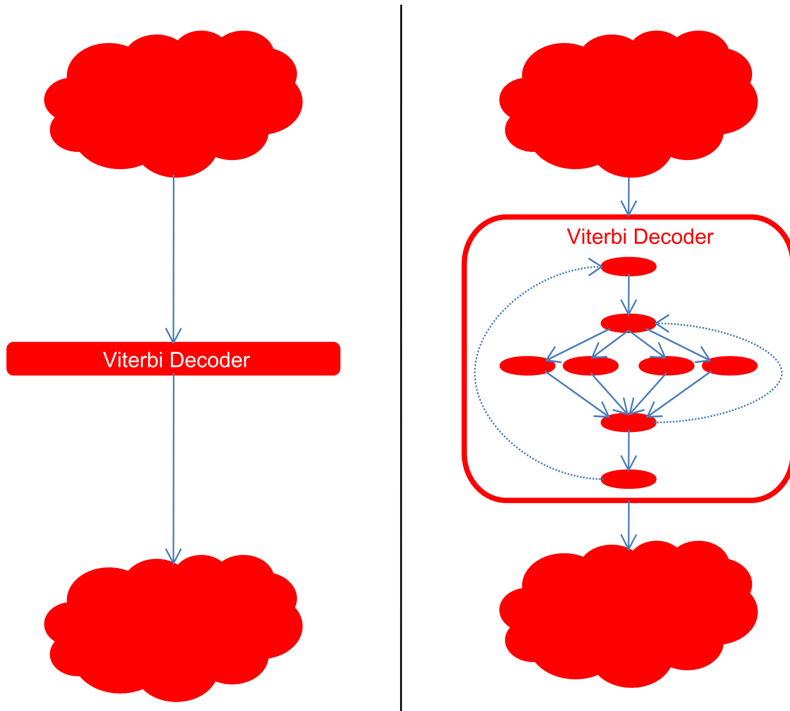


Fig. 2.21 Parallelization of DRM decoder

To evaluate the obtained partitioning, we executed the resulting applications on two different simulators: a modified version of the SimIt-ARM cycle accurate simulator [41] and ReSP [4], a MPSoC simulation platform developed at Politecnico di Milano. Both the simulators have been configured to simulate a homogeneous multiprocessor architecture composed of four ARM processors and a shared memory, connected through a common bus. We decided to target a homogeneous architecture to verify the potential benefits introduced by the partitioning, without being affected by the mapping decisions.

In details, we modified the original version of SimIt-ARM in order to support a multi-core simulation, with private caches and shared memory. The OpenMP pragmas are substituted by code instrumentations that communicate with a co-processor that manages the counting of the simulation time. In this way, our version of SimIt-ARM is able to take into account the concurrent execution of the different tasks. On the contrary, ReSP is able to model a complete architecture where the OpenMP pragmas are translated into proper `pthread` functions supported by a very light operating system. As a result, ReSP allows an accurate simulation of the complete behaviour of a multiprocessor system also considering the overhead due to bus contention and system calls.

Both the sequential and the partitioned versions of the two case studies have been executed on these simulators and the resulting speed-ups are reported in Table 2.3.

Table 2.3 Speed-ups for the parallel versions of JPEG and DRM applications as estimated by Zebu and measured on SimIt-ARM and ReSP

Application	Speed-up		
	Zebu	SimIt-ARM	ReSP
JPEG encoder	2.17	2.56	2.36
DRM decoder	1.72	1.87	1.40

Since SimIt-ARM does not consider the delays due to the contention on accessing the resources (e.g., limit number of processors or concurrent accesses to the bus), it systematically produces an overestimation of the speed-up with respect to the one obtained with ReSP that simulates a more complete model of the target architecture. For this reason, it can represent the maximum speed-up that can be obtained with the current parallelization.

It is worth noting that Zebu completely parallelizes the whole Viterbi decoder that takes most of the execution time of the DRM application. However, the overheads due to thread management and synchronization costs reduce most of the potential benefit in terms of parallelization.

Finally, we can conclude that Zebu is able to statically estimate the speed-up with a good accuracy with respect to methods that require a dynamic simulation of the entire application and the architectures. The corresponding methodology will be detailed in the following section.

2.6.2 Cost Estimation for Partitioning

In all the phases of Zebu, information about performance of the different parts of the application on the different processing elements of the architecture is crucial. Indeed, Zebu needs this type of information to correctly create tasks of proper granularity, to manipulate the task graphs removing inefficient parallelism (i.e., a parallelism which slows down the processing because tasks creation/synchronization/destruction overhead nullifies the gain) and to compute the initial mapping of the application onto the target platform.

Such information could be provided by the tools that precede Zebu in the hArtes toolchain, but, most of the time this information is incomplete and it is limited to the execution time of existing functions, such as library ones. In particular, there is no detailed information about the execution time of an arbitrary piece of code (i.e., a candidate task) clustered by Zebu. For this reason, additional performance estimation techniques have been necessarily included into the tool.

Two types of performance estimation techniques have been implemented in Zebu:

- techniques for estimating the performance of a single task on each processing element of the architecture;
- a technique for accurately estimating the performance of the whole task graph, given a partitioning solution.

The first ones are based on the building of a linear performance model of the different processing elements (GPP and DSP) exploiting a regression technique both for software and hardware solutions.

Linear Regression Analysis

The approach for internal performance estimation is based on both standard statistical and more advanced data mining techniques aimed at identifying some correlations between some characteristics of the piece of code to be analyzed and the actual execution time on the given processing element. As usually done in Data Mining applications, the model is viewed as a simple black box with an output C representing the set of cost functions of the input I representing a certain task and its mapping. Starting from the set of pairs $\langle Ij, Cj \rangle$, statistical and data mining techniques can extract interesting relationships among inputs, i.e. among elements of partitioning configurations, trying also to extrapolate the system model.

Note that, the number of pairs $\langle Ij, Cj \rangle$ usually contains only a small subset of all the possible input-output configurations. The type of relations extracted depends on the technique used. For instance, some statistical techniques (e.g. linear regression) extract relations represented as linear combinations of the input values; whereas data mining techniques (e.g. regression trees, neural networks, or learning classifier systems) are able to extract highly non-linear relations among input-output configurations. These models are then used to evaluate the performance of arbitrary pieces of code during the early phases of the partitioning. To improve the accuracy of the estimation, the performance models are combined with profiling information (e.g., branch probability, loop iteration numbers) retrieved by executing the application on the host machine with a significant dataset for the input application.

The linearity of the models is known to introduce an approximation in the performance estimation, but it simplifies the transformations performed on the task graphs, speeding up the design space exploration. For example, in this way the performance of a new task resulting from the merging of other two existing ones is computed by summing their performance. Finally, the available performance information provided via XML is used during this phase to evaluate the performance of library functions and of not-partitioned functions.

FPGA Area Estimation Model

When the target architecture contains also hardware components, we need an estimation not only for the execution time, but also for the area occupation, to deal with the architectural constraints (e.g., the limited area into the device) in the mapping phase. For this reason, given a candidate task, we first analyze if it can be synthesized by the available synthesis tool (e.g., DWARF in the hArtes toolchain). In particular, if it contains constructs that cannot be synthesized, the estimation is

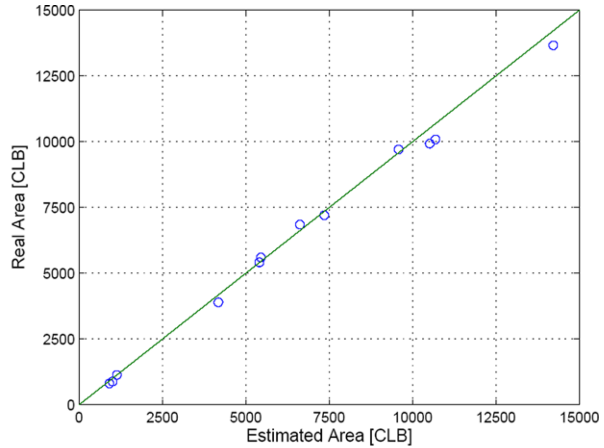


Fig. 2.22 Validation of the FPGA area model

not performed and a mapping to this component will be forbidden for this task. On the other hand, if the implementation is possible, the area model we use for fast estimation is obtained starting from the results of a *fast* high-level synthesis (HLS) of the task, while the performance is related to the worst case execution time of the controller synthesized by the HLS. The total area estimated by the model is composed of two distinct parts: the sequential part (FF) and the combinatorial part (LUT). While the FF part is easy to be estimated, since it is composed of the data registers used in the data-path and of the flip flops used for the state encoding of the controller finite state machine, the LUT part is a little more complex to estimate. Four contributions define the LUT estimation value: FU, FSM, MUX and Glue. The FU part corresponds to the contribution of the functional units and so its value is the sum of the area occupied by each functional unit. The other three parts (FSM, MUX, Glue) are obtained by using a linear regression-based approach and they are defined as follows:

1. the FSM contribution is due to the combinatorial logic used to compute the output and next state;
2. the MUX contribution is due to the number and size of multiplexers used in the data-path;
3. the Glue contribution is due to the logic to enable writing in the flip flops and to the logic used for the interaction between the controller and the data-path.

The coefficients of the model have been obtained starting from a set of standard high-level synthesis benchmarks and the comparing the values with the ones obtained with actual synthesis tools (i.e. Xilinx ISE). The resulting FPGA area model shows an average and a maximum error of 3.7% and 14% respectively and Fig. 2.22 shows the such a comparison where the bisector represents the ideal situation (i.e., the estimated value is equal to the actual one). Further details can be found in [40].

Software Cost Model

The software cost estimation methodology is mainly based on linear regression and it consists of two different steps: construction of a cost estimation model for each processing element and estimation of the performance of the application under analysis using the built cost models. Given a processing element, the model is produced starting from a set of embedded system benchmarks significant for the class of applications which should be estimated. These benchmarks are then analyzed to extract a set of features from each of them that are used as input training set for the building of the estimation model.

There are several different types of features which could be extracted by the analysis of the application without executing it directly onto the target platform. However, only a part of these features is suitable to be used as input for building the cost model because some of them have no correlation with the total performance cost and they can be ignored. Moreover, the larger is the number of input features selected for building the model, the larger should be the size of the training set. To choose among them, some characteristics of the estimation problem have to be analyzed, such as the aspects of the architecture that are usually difficult to be caught by a model. Considering the class of processors and of the target applications of this methodology, the more relevant aspects are the presence of the cache, the characteristics of the pipeline and the type of possible optimizations introduced by the compiler.

The building of the model does not require information about the overall target architecture but only information about the execution cost of a set of significant benchmarks on the target processing element. This information could be retrieved by executing the instrumented code directly onto the element or by using a cycle-accurate simulator of it. This is necessary only during the training to produce the cost model, that will be then used for estimation. In particular, to estimate the performance of an application onto a processing element, only its source code and the model related to the component are used. The source code is compiled and translated into the intermediate representations, that are then analyzed and profiled to extract the selected features for each function. These features are then given as input to the model obtained by the previous phase which quickly produces the performance estimation for each function of the application. By combining the information about cost and the number of executions of each function the total performance cost of the application can be easily estimated.

Besides the features described above, the following metrics have been also considered:

- number of loop iterations;
- static metrics for tasks communication evaluation to be used during the mapping to select an efficient assignment of tasks onto different processing elements.

The number of loop iterations can be easily extracted from the source code for countable loop. For uncountable loops, an estimation of the average number is obtained by an internal dynamic profiling of a proper annotated source code. Our metrics on communication are based on an abstract model that formalizes most of the

Table 2.4 Performance estimation error on single processing element

Processing element	Opt. level	Mean error	Standard deviation of error
ARM	O0	16.9%	15.6%
	O1	14.6%	14.2%
	O2	16.7%	15.8 %
DSP		18.0%	15.9%

information of the communication aspects. Upon this model, we defined a set of metrics that provide information useful for architectural mapping and hardware-software co-design. Further details can be found in [1].

Concerning the software cost model, we built models for the ARM processor and the DSP MAGIC on a suitable set of benchmarks (see Fig. 2.23 for a representation of the dataset). Since these performance estimations are obtained by applying regression techniques and in order to evaluate the effective accuracy of the resulting models, we exploit the cross-validation technique of them. Cross-validation is a technique for assessing the performance of a static analysis on an independent data set and can be performed with different methods. We used the *K-fold cross-validation* where the initial data set is randomly divided into K subsets (in our case, $K = 10$). The model building process is repeated K times and, at each iteration i, a performance model is built by analysing all the subsets but the i-th, which is used to test the model. At the end, the average error across all the K trials is computed and a unique model is built by combining the K models. The obtained results are reported in Table 2.4. Note that different optimization levels have been also considered in building performance models of the ARM processor.

Task Graph Estimation Model

Even if the created tasks are of proper granularity, the parallelism introduced with the initial task graphs cannot produce an actual benefit because of the overhead introduced by the threads creation and synchronization. For this reason, we introduce a methodology to estimate the performance of the whole task graphs highlighting the inefficient parallelism which has to be removed. Such methodology has been introduced since computing the overall performance by simply combining the performance of the single tasks can produce wrong estimations, in particular when the execution time of a task heavily depends on control constructs, on the particular execution path that results to be activated and on its frequency.

Consider for instance the example shown in Fig. 2.24.

Annotating the tasks with their average performance with respect to the branch probability and computing the task graph performance by considering the worst-case execution time on them would lead to a wrong estimation ($1000+500+1000 = 2500$ cycles) with respect to the actual execution time (3000 cycles in all the cases).

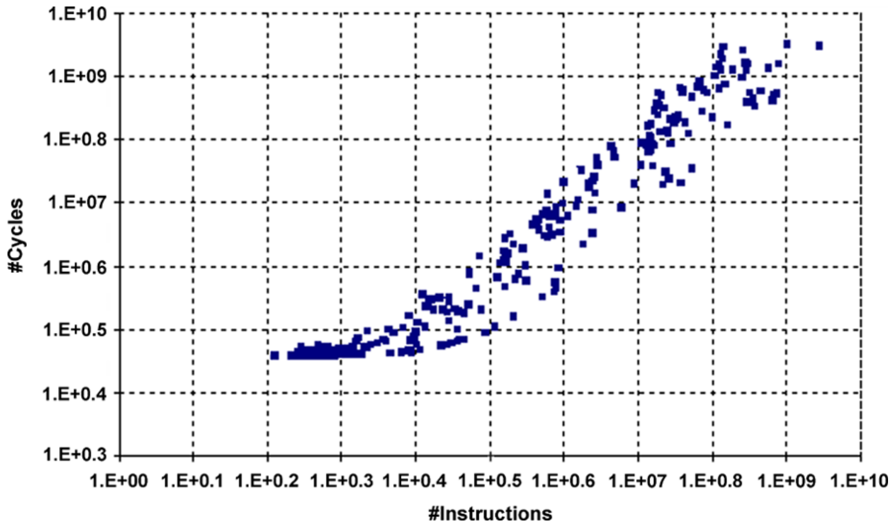


Fig. 2.23 Distributions of the benchmarks dataset with respect to the number of instructions and the number of cycles

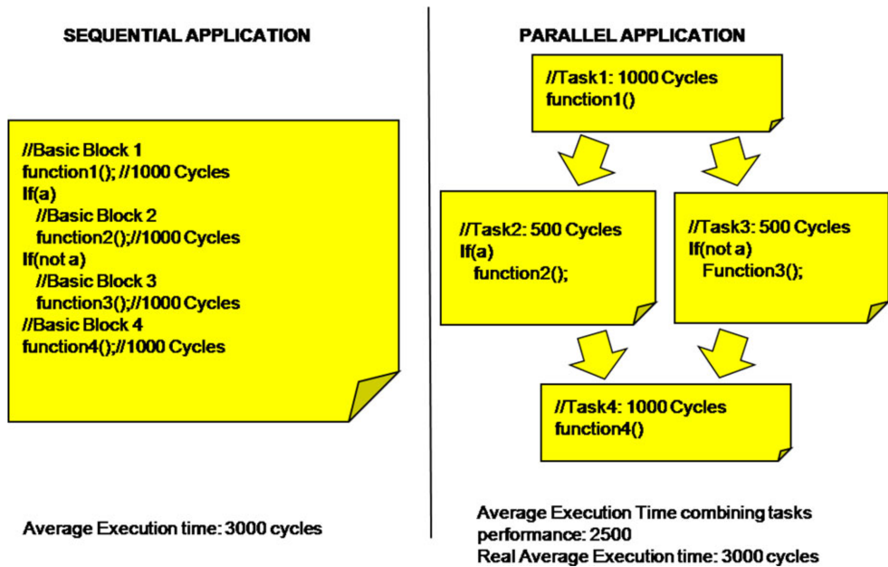


Fig. 2.24 Example of unprofitable parallelism

For this reason, we have proposed [9] a task graph performance estimation technique which combines the performance models of the target processing element, the execution paths, the dynamic profiling information and the structure of the task graph to produce a more accurate estimation of the performance of a partitioned

Table 2.5 Average and Maximum Speed-up Estimation Error on whole task graph compared with techniques based on Worst Case and Average Case

Optimization level	Techniques					
	Worst Case		Average Case		Zebu	
	Avg.	Max.	Avg.	Max.	Avg.	Max.
O0	25.3%	85.4%	24.3%	85.4%	3.9%	12.6%
O2	20.9%	85.7%	21.6%	85.7%	4.3%	11.5%

application. In particular we exploit the processing element performance models described above to estimate the performance of the single pieces of code (e.g.: `function1()`). For each application execution path (e.g., Basic Block 1–Basic Block 2–Basic Block 4) identified by the profiling of the code, we evaluate its contribution to the overall execution time of the task graph (e.g.: 3000 Cycles for the considered path). This combination takes also into account the cost for creating and synchronizing the parallel tasks. These costs have been retrieved by averaging a set of measures obtained running parallelized and automatically instrumented code on the target platform [8]. Finally we obtain the overall performance of the task graph combining the contribution of each execution path, that is averaged by its relative frequency measured executing the application on the host machine on a set of significant input datasets. These estimations are then used to evaluate the effects of the different parallelizations and accordingly transform the task graph to improve the performance. In particular, all the unprofitable parallel sections are removed by applying different transformations, such as merging the corresponding parallel tasks. Moreover, since OpenMP is used to annotate the parallelism in the produced C specification, transformations of the task graph are necessary to make it suitable for the OpenMP paradigm. The task graph is thus arranged to accomplish this model and balance the number of parallel tasks in each section.

Note that when different transformations are available, we choose the one that introduces the greatest benefit (or the lowest degradation) in terms of performance, according to the estimation techniques described above.

The methodology has been validated on a set of applications parallelized either by hand or by Zebu. In particular, the parallel code has been executed on the modified version of SimIt-ARM [41] described in the previous section. Then, this cycle-accurate simulation has been compared with the estimation obtained with the proposed methodology and the results that can be achieved by traditional techniques based on the computation of Worst Case and Average Case. The results about the error on speed-up estimation are reported in Table 2.5.

These results show that the proposed methodology is able to obtain an estimation of the speed-up much closer to the actual simulation with SimIt. It has been thus integrated into Zebu since it better drives the exploration process about the benefits that can be obtained with different parallelizations, as described above.

2.7 Task Mapping

This section describes the task mapping process of the hArtes toolchain [34]. The goal of task mapping is to select parts of the C application to execute on specialized processors (DSP or FPGA), in order to minimize the overall execution time of the application. The task mapping process is designed with three novel features. The first is that near optimal solutions can be generated by a heuristic search algorithm to find the optimal solution. The second is that developers can guide the decision process by providing directives to constrain the mapping solution. Finally, a transformation engine has been devised to perform source-level transformations that can improve the execution of each individual task on the selected processing element, and as a consequence helps the mapping process generate a better solution. In this section we cover the following topics:

1. **Task Mapping Approach (Sect. 2.7.1)**. We provide an overview of the task mapping process including the basic design flow and its main components.
2. **Task Filtering (Sect. 2.7.2)**. The task filtering process is responsible for constraining the mapping solution and ensuring that it complies to the hArtes platform.
3. **Task Transformation (Sect. 2.7.3)**. The task transformation is part of the task mapping approach which enables developers to select, describe and apply transformations to individual tasks, taking into account application-specific and platform-specific requirements.
4. **Mapping Selection (Sect. 2.7.4)**. The mapping selection is responsible for assigning a processing element to each task in the program in order to exploit as much as possible the full capabilities of the heterogeneous system, while providing a feasible solution for the hArtes platform.
5. **Experimental Features (Sect. 2.7.5)**. We report three research experiments that have been performed in conjunction with the task mapping approach, and show that they improve the quality of the solution.
6. **hArmonic Tool (Sect. 2.7.6)**. We illustrate some of the main features of the hArmonic tool, which implements the task mapping process in the hArtes toolchain. In addition, we summarize the key results.

2.7.1 Mapping Approach

The task mapping design flow and its main components are shown in Fig. 2.25. There are two main inputs: (1) the source code, which supports an arbitrary number of C source files, and (2) the platform specification. The platform specification describes the main components of the heterogeneous system, including the processing elements, the interconnect, storage components, and system library functions.

The first stage of task mapping is the automatic C guidelines verification, which determines whether the source code complies with the restrictions of the mapping process. Some of these limitations correspond to language elements and coding

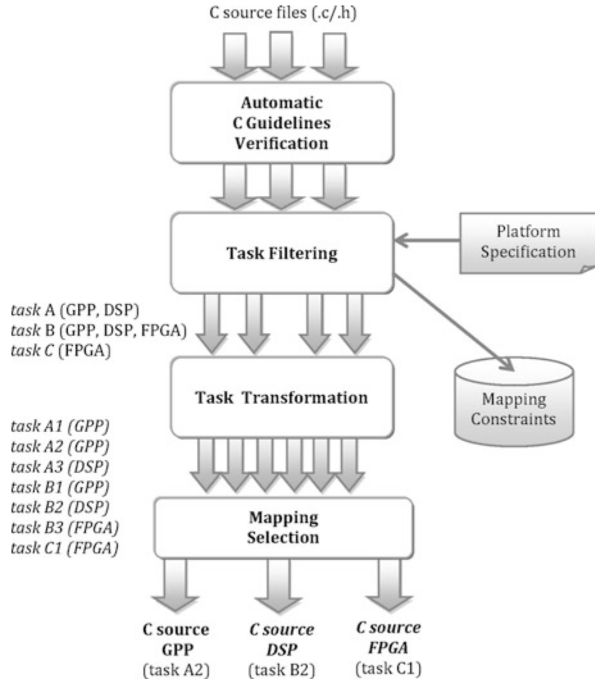


Fig. 2.25 The task mapping approach is comprised by four main modules: (1) an automatic C guidelines verification process that automatically determines if the code conforms to the restrictions of the task mapping process, (2) a filtering engine that determines which processing elements can support each task, (3) a task transformation engine which produces one or more implementations of a given task that can potentially exploit the capabilities of the processing elements in the system, and (4) the mapping selection process which assigns a processing element implementation to each task of the application

styles that can hinder the efficiency of the task mapping process, such as the use of function pointers. Other limitations include forbidding language elements not supported by our approach, for instance static function definitions. Hence, the C guidelines verification can provide hints to improve the task mapping process, enabling developers to revise their code in specific parts of the program. Furthermore, any violation of the C guidelines found by the automatic verification process will terminate the task mapping process.

The second stage of task mapping is task filtering. The objective of task filtering is to determine which processing elements can support each individual task in the application, and under what conditions. The filtering engine has two main outputs. The first is a list of supported processing elements for each task in the application. For instance, in Fig. 2.25, the filtering engine determines that task A can only be supported by the GPP and DSP processors, whereas task C can only be synthesized to the FPGA. The second output of the filtering engine is a set of mapping constraints which ensure that the mapping result would comply with the hArtes platform requirements and limitations. For instance, one mapping constraint could state that task B and task C must be mapped to the same processing element. Another constraint could state that task C can only be mapped to an FPGA. When combining these two constraints, the mapping solution should map task B to an FPGA to satisfy all constraints, otherwise the mapping solution is infeasible. Section 2.7.2 provides more details about the task filtering process.

The third stage of task mapping is task transformation [49]. Once each task has been assigned to a list of feasible processors by the previous stage, the transforma-

tion engine generates one ore more implementations of each task according to an established portfolio of transformations for each processing element. The transformation engine has two main benefits. First, it can potentially uncover optimizations that exploit the properties of each individual processing element in the system, and can consequently improve the performance of the resulting mapping solution. Second, it enables the mapping process to target a wide range of implementations for each task, making the selection process less biased towards a particular processing element. Section 2.7.3 presents the task transformation engine in more detail.

The fourth and final stage of task mapping is mapping selection [31]. At this point, we have a set of tasks and associated implementations generated by the task transformation engine, a set of mapping constraints produced by the filter engine to comply with the hArtes platform as well as constraints provided by the developer, and a cost estimator which computes the cost of each task. The search process is geared towards finding a solution that minimizes as much as possible the cost of the application and satisfies the mapping constraints. Once a solution is found, and tasks have been assigned to their corresponding processing elements, we generate the code for each processing element. The generated code is compiled separately by the corresponding backend compilers, and linked afterwards to produce a single binary. Each individual compilation unit corresponds to a subset of the application designed to realize the overall mapping solution.

2.7.2 Task Filtering

The task filtering process relies on individual filters to determine what processing elements can support each application task in order to find a feasible mapping solution. A total of 15 different filters are employed, a sample of which are shown in Table 2.6.

Each individual filter is responsible for computing the list of processing elements to which each task can be mapped. Since the result of one filter can affect the result of other filters, they are run sequentially in a loop until there are no changes in the result. In addition to assigning a list of processing elements to each task, each filter can derive a set of mapping constraints which can, for instance, specify that a task mapping is valid only if other task mappings are part of the solution (e.g. mapping task A to ARM $\Rightarrow$ mapping task B to ARM). By constraining the solution, the task filtering process can considerably reduce the mapping selection search time as well as ensuring that the resulting mapping solution can be realized on the hArtes platform.

2.7.3 Task Transformation Engine

The task transformation engine (Fig. 2.26) applies pattern-based transformations, which involve recognizing and transforming the syntax or dataflow patterns of design descriptions, to source code at task level. We offer two ways of building task

Table 2.6 List of filters employed to find mappings

Filter	Description
FRMain	Ensures that the main function is always mapped to the master processor element (GPP)
FRCall	Ensures that a function that calls another is either mapped to the master processing element, or both functions are mapped to the same processing element
FRDwarvFPGA	Ensures that a function is not mapped to FPGA if it does not comply with the restrictions of the DWARV compiler, such as not supporting `while` loops
FRTargetDSP	Ensures that a function is not mapped to DSP if it does not comply with the restrictions of the DSP compiler, such as the use of array declarations with nonconstant sizes
FRGlobal	Ensures that functions that share one or more global variables are mapped to the same processing element
FRLibrary	Ensures that a function cannot be mapped to a processing element (other than the master processing element) if it invokes a library function not specified in the platform specification file
FRRemoteFn	Ensures that if one function calls another, they are either mapped to the same processing element or to different processing elements if the parameter types can be used in a remote call
FRTypes	Ensures that a function can only be mapped to a processing element if it supports the basic data types specified in the platform specification file

transformations: using the underlying compiler framework, ROSE [42], to capture transformations in C++; this is complex but offers the full power of the ROSE infrastructure. Alternatively, our domain-specific language CML simplifies description of transformations, abstracting away housekeeping details such as keeping track of the progress of pattern matching, and storing labeled subexpressions.

CML is compiled into a C++ description; the resulting program then performs a source-to-source transformation. For design exploration, we also support interpreting CML descriptions, allowing transformations to be added without recompiling and linking. Task transformations could be written once by domain specialists or hardware experts, then used many times by non-experts. We identify several kinds of transformations: input transformations, which transform a design into a form suitable for model-based transformation; tool-specific and hardware-specific transformations, which optimize for particular synthesis tools or hardware platforms.

Each CML transformation (Fig. 2.26) consists of three sections: (1) pattern, (2) conditions, and (3) result. The pattern section specifies what syntax pattern to match and labels its parts for reference. The conditions section typically contains a list of Boolean expressions, all of which must be true for the transformation to apply. Conditions can check: (a) validity, when the transformation is legal; (b) applicability: users can provide additional conditions to restrict application. Finally, the result section contains a pattern that replaces the pattern specified in the pattern section, when conditions apply.

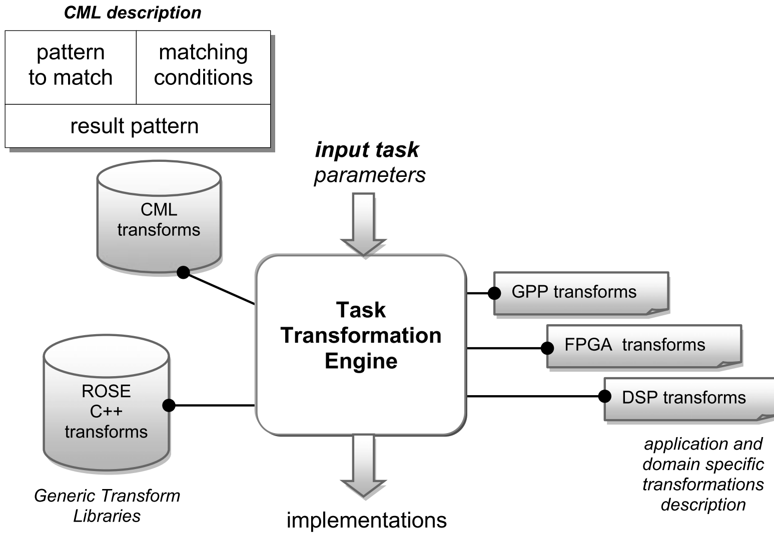


Fig. 2.26 An overview of the task transformation engine. The task transformation engine receives as input a task and additional parameters such as the processing element that we wish to target, and generates a set of implementations. The set of transformations to be applied to each processing element is provided by the user. The implementations of transformations are stored as shared libraries for ROSE transformations, and as text files for CML-based transformations. A CML description consists of three sections: the pattern to match, the matching conditions, and the resulting pattern (Listing 2.1)

```
1       pattern {
2          for (var(a)=0;var(a)<expr(e1);var(a)++){
3             for (var(b)=0;var(b)<expr(e2);var(b)++)
4                stmt(s);
5          }
6       }
7       conditions {

9       }
10      result {
11         for (newvar(nv)=0;
12            newvar(nv)<expr(e1)*expr(e2);
13            newvar(nv)++)
14         {
15            var(a) = newvar(nv) / expr(1);
16            var(b) = newvar(nv) % expr(1);
17            stmt(s);
18         }
19      }
20   }
```

Listing 2.1 CML description of the loop coalescing transformation

A simple example of a CML transformation is loop coalescing (Listing 2.1), which contracts a nest of two loops into a single loop. Loop coalescing is useful

in software to avoid loop overhead of the inner loop, and in hardware to reduce combinatorial depth. The transformation works as follows:

- Line 1: LST 2.1 starts a CML description and names the transformation
- Lines 2–7: LST 2.1 gives the pattern section, matching a loop nest. CML patterns can be ordinary C code, or labelled patterns. Here `var(a)` matches any value and labels it "`a`". From now on, each time `var(a)` appears in the CML transform, the engine tries to match the labelled code with the source code.
- There is no conditions section (lines 8–10: LST 2.1), as coalescing is always valid and useful.
- Lines 11–20: LST 2.1 gives the result section. The CML pattern `newvar(nv)` creates a new variable which is guaranteed unique in the current scope. The resulting loop behaves the same as the original loop nest. The values of the iteration variables, `var(a)` and `var(b)`, are calculated from the variable `newvar(nv)` in the transformed code. This allows the original loop statement, `stmt(s)`, to be copied unchanged.

When the transformation engine is invoked, it triggers a set of transformations that are specific to each processing element, which results in a number of implementations associated with different tasks and processor elements. The implementation description of ROSE transformations are stored as shared libraries, and the CML definitions as text files. Because a CML description is interpreted rather than compiled, users can customise the transformation by using a text editor, and quickly evaluate the effects of the transformation without requiring an additional compilation stage.

To show the effect of our transformation engine, we apply a set of transformations to an application that models a vibrating guitar string (provided by UNIVPM). These transformations have been described in both CML and ROSE, and allow the user to explore the available design space, optimizing for speed and memory usage. We modify the application for a 200 second simulated time to show the difference between the various sets of transformations. The set of transformations includes:

- **S**: simplify (inline functions, make iteration variables integer, recover expressions from three-address code)
- **I**: make iteration bounds integer
- **N**: normalise loop bounds (make loop run from 0 to N-2 instead of 1 to N-1)
- **M**: merge two of the loops
- **C**: cache one array element in a temporary variable to save it being reread
- **H**: hoist a constant assignment outside the loop
- **R**: remove an array, reducing 33% of memory usage (using two arrays instead of three)

Figure 2.27 shows how the design space can be explored by composing these transformations. Transformation S provides an almost three-fold improvement, mostly by allowing the compiler to schedule the resulting code. Transformation I gives nearly another two-fold improvement, by removing floating-point operations from the inner loop. Transformation N gives a small improvement after transformation I. Transformation M slows the code down, because the merged loop uses

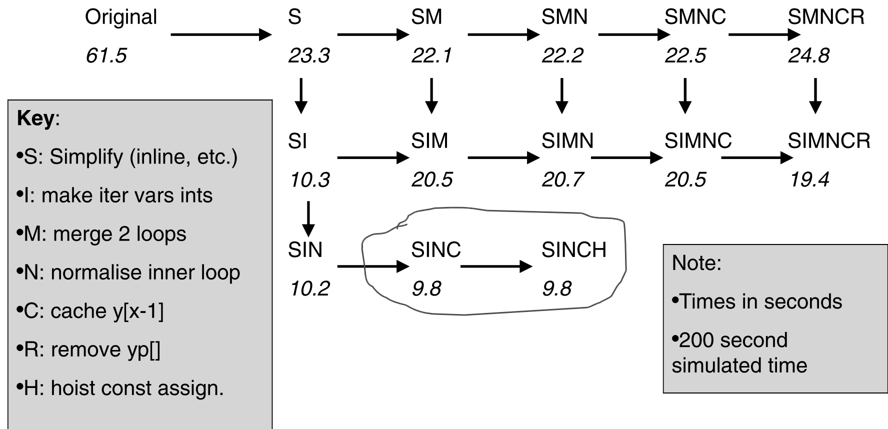


Fig. 2.27 Starting with the original code for the application that models the vibration of a guitar string, we explore ways of using seven different transformations to attempt to improve the run time and memory usage. Much of the speedup comes from simplifying the code and making iteration variables integer, while the remainder comes from caching to prevent repeat memory access and removing a constant assignment from the loop body. The caching also enables one array to be eliminated (about 33% reduction in memory usage), possibly at the expense of performance

the GPP cache badly. Transformation C improves the integer code (I) but leaves the floating point version unimproved. Finally, transformation R gives a small improvement to the integer version, but actually slows down the floating-point version. Overall, we have explored the design space of transformations to improve execution time from 61.5 seconds to 9.8 seconds, resulting in 6.3 times speedup.

2.7.4 Mapping Selection

Our mapping selection approach is unique in that we integrate mapping, clustering and scheduling in a single step using tabu search with multiple neighborhood functions to improve the quality of the solution, as well as the speed to attain the solution [30]. In other approaches, this problem is often solved separately: a set of tasks are first mapped to each processing element, and a list scheduling technique then determines the execution order of tasks [54], which can lead to suboptimal solutions.

Figure 2.28 shows an overview of the mapping selection approach. Given a set of tasks and the description of the target hardware platform, the mapping selection process uses tabu search to generate different solutions iteratively. For each solution, a score is calculated and used as the quality measure to guide the search. The goal is to find a configuration with the lowest score, representing the execution time.

Figure 2.29 illustrates the search process. At each point, the search process tries multiple directions (solid arrows) using different neighborhood functions in each move, which can increase the diversification to find better solutions. In the proposed

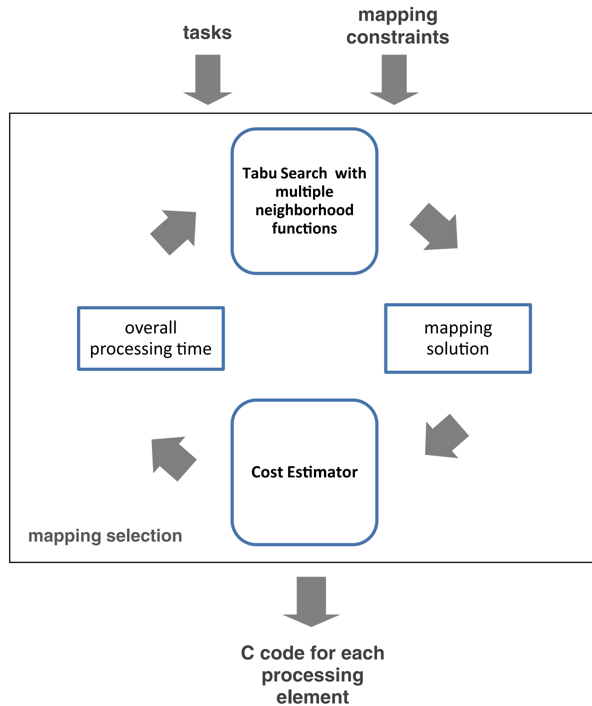


Fig. 2.28 An overview of the mapping selection process

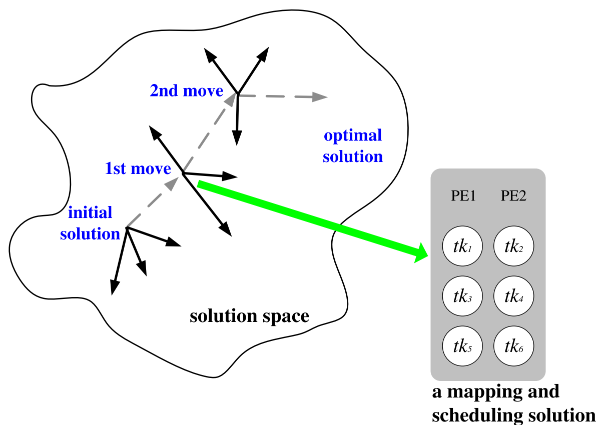


Fig. 2.29 Searching for the best mapping and scheduling solution using multiple neighborhood functions. The *solid arrows* show the moves generated by different neighborhood functions. The *dotted arrows* denote the best move in each iteration of the search. PE: processing element, tk: task

technique, after an initial solution is generated, two neighborhood functions are used to generate neighbors simultaneously. If there exists a neighbor of lower cost than the best solution so far, and it is not in the tabu list, this neighbor is recorded. Otherwise a neighbor that cannot be found in the tabu list is recorded. If all the above conditions cannot be fulfilled, a solution in the tabu list with the least degree, i.e. a solution resident in the tabu list for the longest time, is recorded. If the recorded solution has a smaller cost than the best solution so far, it is recorded as the best

solution. The neighbors found are added to the tabu list, and solutions with the least degree are removed. This process is repeated until the search cannot find a better configuration for a given number of iterations. An advantage of using multiple neighborhood functions is that the algorithm can be parallelized, and therefore the time to find a solution can be greatly reduced.

Another important component of the mapping selector is the cost estimator (Fig. 2.28). The cost estimator computes the overall processing time, which is the time for processing all the tasks using the target computing system including data transfer time between processing elements. The processing time of a task on a processing element is calculated as the execution time of this task on the processing element plus the time to retrieve results from all of its predecessors. The data transfer time between a task and its predecessor is assumed to be zero if they are assigned to the same processing element.

Our approach for estimating the cost of a task running on a particular processing element currently exploits rule-based techniques. Our rule-based estimator makes use of linear regression to estimate the cost based on a set of metrics:

$$EstTime = \sum_{i=1}^{N} T_{P_i}$$

where N is the number of instructions, P_i is the type of instruction i, T_{P_i} is the execution time of instruction P_i. Each processing element contains one set of T_{P_i} for each type of instruction. Instructions include conditionals and loops, as well as function calls.

2.7.5 Experimental Features

In this section we provide a brief overview of three experiments that have been developed independently from the task mapping approach. They have been proved to enhance the task mapping process.

- **Automatic Verification**. A verification framework has been developed in conjunction with the task transformation engine [48]. This framework can automatically verify the correctness of the transformed code with respect to the original source, and currently works for a subset of ANSI C. The proposed approach preserves the correct functional behavior of the application using equivalence checking methods in conjunction with symbolic simulation techniques. The design verification step ensures that the optimization process does not change the functional behavior of the original design.
- **Model-Based Transformations**. The task transformation engine (Sect. 2.7.3) supports pattern-based transformations, based on recognizing and transforming simple syntax or dataflow patterns. We experiment with combining such pattern-based transformations with model-based transformations, which map the source

code into an underlying mathematical model and solution method such as geometric programming. We show how the two approaches can benefit each other, with the pattern-based approach allowing the model-based approach to be both simplified and more widely applied [32]. Using a model-based approach for data reuse and loop-level parallelization, the combined approach improves system performance up to 57 times.

- **Data Representation Optimization**. Another approach used in conjunction with the task transformation engine is data representation optimization for hardware tasks. The goal of data representation optimization is to allow developers to trade accuracy of computation with performance metrics such as execution time, resource usage and power consumption [38]. In the context of reconfigurable hardware, such as FPGAs, this means exploiting the ability to adjust the size of each data unit on a bit-by-bit basis, as opposed to instruction processors where data must be adjusted to be compatible with register and memory sizes (such as 32 bits or 64 bits).

  The data representation optimization approach has two key features. First, it can be used to generate resource-efficient designs by providing the ranges of input variables and the desired output precision. In this case, the optimization process analyzes the code statically and derives the minimum word-lengths of variables and expressions that satisfy user requirements. In addition, a dynamic analysis can be employed to automatically determine the input ranges and output requirements for a particular set of test data.

  Second, this approach can be used to generate power-efficient designs using an accuracy-guaranteed word-length optimization. In addition, this approach takes into account library functions where implementation details are not provided. Results show power savings of 32% can be achieved by reducing the accuracy from 32 bits to 20 bits.

2.7.6 The hArmonic Tool

This section describes some of the features of the hArmonic tool which implements the functionality described in Fig. 2.25. The hArmonic tool receives as input an arbitrary number of C sources, and generates subsets of the application for each individual processing element according to the derived mapping solution. The list of features include:

- **System and Processor Driver Architecture**. hArmonic uses a driver-based architecture. Each driver captures the specific elements of the platform (system driver) and the processor elements (processor driver) and interfaces with the mapping process which is generic. For instance, the system driver is responsible for estimating the cost of the whole application for the corresponding platform. The processor driver, on the other hand, shows whether a particular task can be supported by associated processing elements. This way, hArmonic can be customized to support different hardware platforms and processing elements by adding new drivers, without the need to change the interfaces or the core mapping engine.

Guidelines	Description
Avoid function pointers	Detects the use of function pointers
Avoid union types	Detects the use of union types
No static functions	Detects the use of static functions
No comma operators	Detects the use of complex expressions using comma operators
No implicit function calls	Detects the use of function calls which were not previously declared

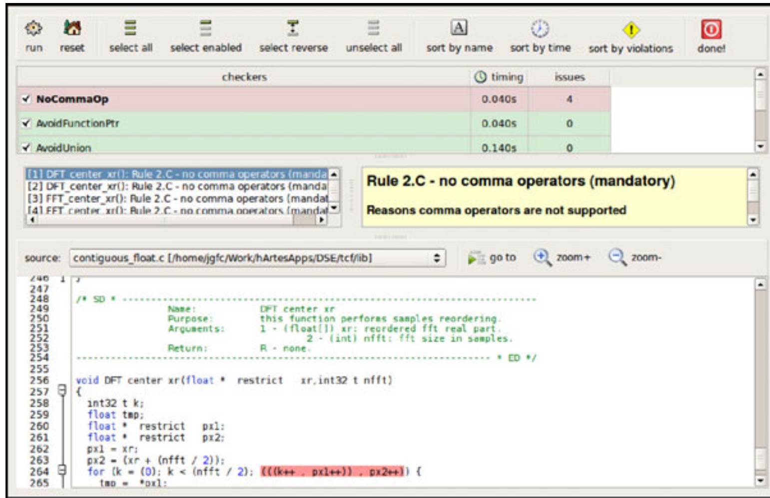


Fig. 2.30 A description of C guidelines automatically detected by hArmonic, and a screenshot of its graphical user-interface

- **Automatic C Guidelines Verifier**. The automatic C guidelines verifier analyzes the input C source and highlights any guideline violation and makes recommendations about the code, to ensure conformance and integration between the tools. For instance, it automatically detected instances of static functions in code generated by SCILAB-to-C tool (AET), and the use of function pointers in the x264 source-code. The former is a violation because static functions are constrained to the source-file where they are located, whereas hArmonic must move functions to different source-files according to the mapping solution. Avoiding the use of function pointers, on the other hand, is a recommendation, as hArmonic does not support dynamic mapping and therefore the quality of the solution can be diminished. Developers are therefore invited to change their sources or tools in a way that can maximize the efficiency of the toolchain and the mapping solutions.

  The C guidelines supported by hArmonic are shown in Fig. 2.30. This figure also presents a screenshot of the corresponding graphical user interface, which illustrates how hArmonic automatically pinpoints potential problems in the source-code that prevent either good mapping or a feasible solution.

- **Multiple Source Files Support**. To support full C projects where applications can span multiple source files and contain preprocessing macros and directives, hArmonic requires a complex analysis of the code to reconcile multiple symbols in each compilation unit and to obtain a complete picture of the application.
- **Mapping Constraint System**. To facilitate design exploration and to allow tools and developers to influence the task mapping process, hArmonic supports a mapping constraint system that can be used to guide the mapping process. For instance, a tool can automatically generate C code specifically for DSP and instruct hArmonic to map those functions to DSP. There are three ways in which developers can define mapping constraints:
  – **Source Annotation**. In this case, a #pragma annotation can be optionally placed next to a function declaration or a function call to set the final mapping for that task. In the example below, any task relating to function f ( ) will be mapped to MAGIC (DSP) or to VIRTEX4 (FPGA).

```
#pragma map call_hw MAGIC VIRTEX4
void f() {...}
```

  – **Constraint File**. Rather than annotating the source which can span through many files and require parsing the code, the mapping constraints can be placed in a separate text file making it easier for tools to interact with hArmonic. In the example below, all tasks related to function f( ) are mapped to either ARM or MAGIC, task g( ) with id 12 must be mapped to VIRTEX4, and all tasks defined in source-file dsp_fn.c are to be mapped to MAGIC.

```
f():= ARM,MAGIC
g()/12:= VIRTEX4
dsp_fn.c:=MAGIC
```

  – **Graphical User Interface**. Additionally, developers can use hArmonic's graphical user-interface to set the constraints directly by selecting one or more tasks and associating them to the desired processing element as shown below:

reset | select all | select reverse | unselect all | clear constraint | map ARM | map MAGIC | map VIRTEX4 | run mapper

tasks	ARM	MAGIC	VIRTEX4	Constraint
FFT_apply_window()/416	NO	NO	YES	VIRTEX4
FFT_apply_window()/468	NO	NO	YES	VIRTEX4
FFT_dfto_rdx2_cf()/426	YES	YES	NO	
FFT_dfto_rdx2_cf()/448	YES	YES	NO	
FFT_generate_twiddles()/263	YES	NO	NO	
FFT_generate_twiddles()/325	YES	NO	NO	
FFT_idfto_rdx2_cf()/431	YES	YES	NO	
FFT_idfto_rdx2_cf()/453	YES	YES	NO	

- **OpenMP Support**. OpenMP directives can be introduced to indicate that two or more tasks can be parallelized. When the OpenMP mode is enabled, hArmonic is able to provide a mapping solution that exploits parallelization. In the example below, depending on the cost estimation of f( ) and g( ), these functions may be mapped to different processing elements (such as DSP and FPGA) to reduce the execution time of the application.

```
#pragma omp parallel sections
...
{
 #pragma omp section
 f();
}
#pragma omp section
{
 g();
}
```

- **Source Splitting**. In the initial versions of hArmonic, the mapping process would generate a single C source file with source annotations for each function indicating to which processing element they were assigned, and therefore which compiler to use. However, this approach turns out to be infeasible because (a) compilers would have to be made more complex so that they could compile selectively, (b) there are subtle and not so subtle differences between the C languages supported by the compilers, which mean that it would be difficult to generate one single source that all parsers could support, (c) there are specific headers required for each processing element which can be incompatible when used together in a single source. To avoid these problems, later versions of hArmonic generate one source file for each processing element that is a part of the mapping solution. Each compilation unit is a subset of the application and includes all the necessary function definitions that are mapped to a particular processing element, as well as every type and symbol definitions required for a correct compilation and execution. Furthermore, hArmonic supports C customizations in the source for each processing element to conform to the hArtes platform, backend compilers and system headers.

Table 2.7 summarizes the results of the complete automatic mapping performed by the final version of the hArmonic tool on all hArtes applications. For the largest application (Audio Enhancement) with more than 1000 tasks, it takes a single minute to complete the whole task mapping process, resulting in a 30 times speed-up. The speed-up corresponds to the ratio between the application running solely on the GPP (no mapping) and execution of the application where selected tasks have been automatically assigned to the available accelerators on the hArtes platform. The hArtes constraints column corresponds to the number of mapping restrictions generated by the mapping filter rules (see Sect. 2.7.4) in order to comply with the hArtes platform and ensure that the mapping solution is feasible.

Table 2.7 Evaluation of the main hArtes applications

hArtes application	Total number of tasks [mapped to DSP]	hArtes constraints	Time to compute solution (sec)	Speed up
From UNIVPM				
Xfir	313 [250]	1180	8	9.8
PEQ	87 [39]	261	3	9.1
FracShift	79 [13]	251	24	40.7
Octave	201 [137]	692	6	93.3
Audio enhancement	1021 [854]	4366	59	31.6
From Thales				
Stationary noise filter	537 [100]	3101	17	5.2
From FHG-IGD				
Beamformer	168 [38]	677	6	7.9
Wave-field synthesis	145 [24]	455	5	9.2

In addition, the hArmonic tool automatically achieves speed-ups that ranges from 5.7 times to more than 90 times by mapping selected tasks to available processing elements in the hArtes platform. Several factors contribute to the quality of the solution: (a) the hArtes applications have been developed with the recommended C guidelines and best practices to exploit the toolchain, (b) the performance and efficiency of the hArtes toolchain synthesis tools (GPP, DSP and FPGA compilers), (c) the use of specialized libraries such as DSPLib, and finally (d) the architecture and efficiency of the hArtes hardware platform. Further descriptions of the hArtes applications and the automatic mapping approach can be found in Chaps. 4 and 5.

2.8 Compiling for the hArtes Platform

In order to execute the produced application on the hArtes target platform, the following steps have to be performed:

- compiling the code for the GPP processing element and generating the additional code to address the MOLEN paradigm;
- compiling the code for the DSP processor;
- generating the hardware components;
- linking and creating the effective executable.

2.8.1 HGCC Compiler and MOLEN Programming Paradigm

The MOLEN programming paradigm is a paradigm [52] that offers a standard model of interaction between the processing elements of a heterogeneous systems.

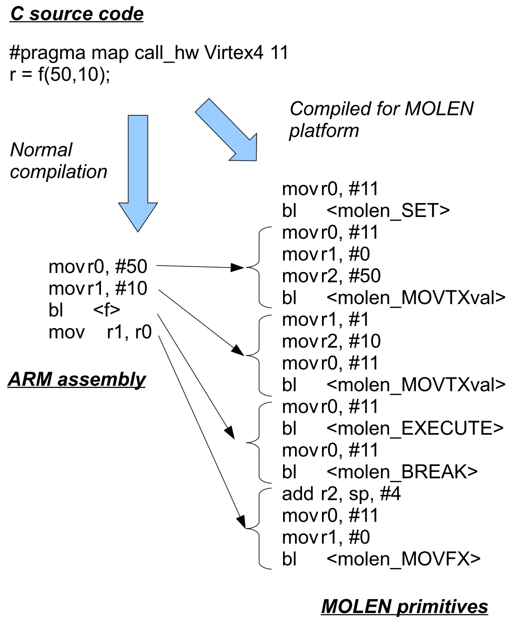


Fig. 2.31 Example of MOLEN

By extending the run-time system and the compiler with 6 primitives a theoretically unlimited number of different processing elements can be managed transparently. New types of processing elements can be added without complex modifications to the existing infrastructure—the only thing needed being the specific implementation of the 6 primitives for the new element.

The paradigm is based on the processor/coprocessor model and the basic computation unit is a function. We consider that the application is running on the General Purpose Processor (**GPP**) and the computationally intensive functions (named from this point on **kernels**) are executed by different accelerators or processing elements (**PE**). The memory is considered to be shared between the GPP and all the PE-s. The parameters have to be placed in special transfer registers that have to be accessible by both the GPP and the PE.

The modified GPP compiler, namely **HGCC**, will replace a normal call to the function with the 6 primitives as shown in Fig. 2.31.

Each of the primitive has to be implemented for each processing element. The list of the primitives for one processing element is the following:

- SET(int *id*)—performs any initialization needed by the processing element. In the context of hArtes it means: reconfiguration for FPGA, load of binary executable file for Diopsis DSP.
- MOVTXval(int *id*, int *reg*, int *val*)—moves the value *val* to the specific transfer registers *reg* that will be used by the kernel identified by *id*
- EXECUTE(*id*)—starts the kernel identified by *id* on the processing element processing element. The GPP can continue execution as this call as asynchronous.

- BREAK(*id*)—used as a synchronization primitive, will not return until the PE execution the kernel identified by *id*, has not finished execution.
- MOVFXval(int *id*, int *reg*, int &*val*)—does the reverse of MOVTXval. Will transfer the value from the register *reg*, used by kernel identified by *id* and will store it at *val*.

These functions are provided in a library, and are linked with the binary generated by the GPP compiler.

To mark a function is a kernel that has to be accelerated a pragma is used. The syntax of the pragma is the following:

```
#pragma map call_hw Virtex4 1
```

The third element in the pragma is the processing element, while the fourth element is the unique identifier of the kernel. This pragma can be placed at the following points in the program:

- function declaration—which means all the calls to that function will be replaced with MOLEN primitives.
- function call—which means that just that call will be called accelerated.

Parallel Execution and MOLEN

Even if it is not explicitly specified MOLEN is a asynchronous paradigm, i.e. the PE can execute in parallel with the GPP. To describe this in the application we use the OpenMP parallel pragma, because it is a well established syntax. We do not use any other feature of the OpenMP specification so far.

Let's assume we have two independent kernels: *kernelA* and *kernelB*. Without taking into account the parallelism the compiler will generate the code and schedule in Fig. 2.32.

Assuming *kernelA* and *kernelB* are put in an OpenMP parallel section the compiler will generate the optimized code and schedule as in Fig. 2.33.

An important aspect is that even though it uses OpenMP syntax, the parallelism is obtained without any threading on the general purpose processor, which can save a lot of execution time. To obtain such an effect the compiler does the following steps:

- it replaces the calls with the MOLEN primitives;
- when it generates the assembly code, the compiler starts processing the parallel regions one by one. For each of them, it copies to the sequential output all the code before the last BREAK primitive of the section;
- copies the rest of each section.

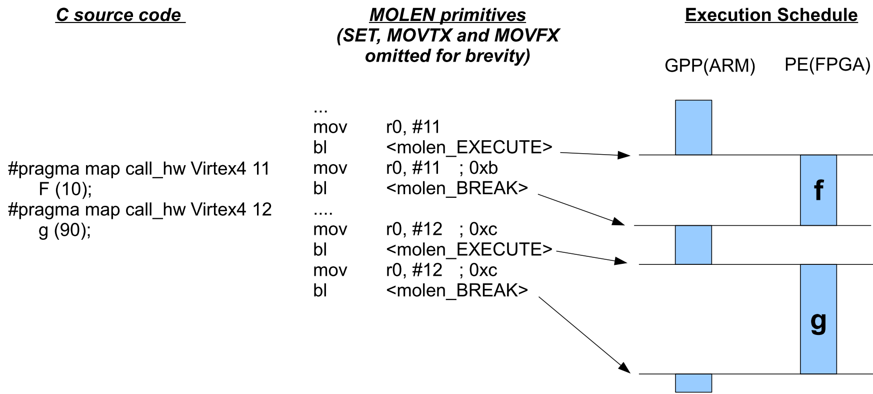


Fig. 2.32 Molen without parallelism

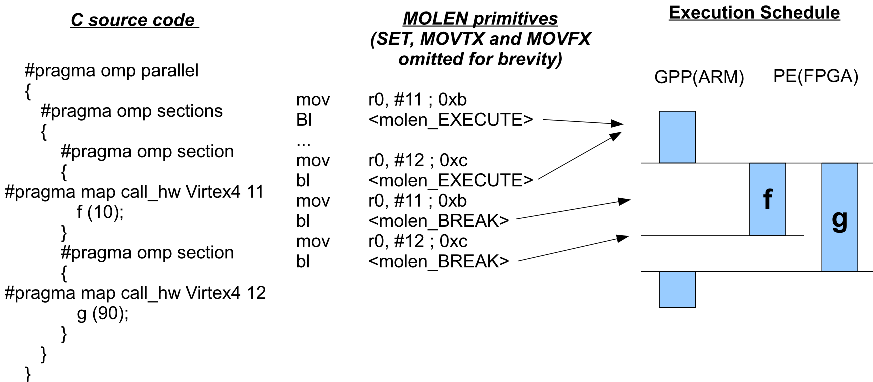


Fig. 2.33 Molen with parallelism expressed with OpenMP syntax

2.8.2 DSP Compiler

For the DSP Processing Element, the hArtes toolchain uses a commercial toolchain from Target™. This toolchain has its own binary format, the C-compiler is not fully C99 compliant and has particular syntaxes to optimize C code. One of the goal of the hArtes toolchain is to behave exactly as the compilation chain of a GPP where one single executable is created containing all symbol and debug information. So the main integration task has been to convert this proprietary image into something that could be linked together with the GPP compiler.

The integration of this tool inside the hArtes toolchain has been made possible through the *mex2elf* tool that translates Target's DSP proprietary format into an ARM ELF compliant format. A more complex task has been to re-create debugging information following DWARF2 specifications, from logs and disassembler outputs of the Target compiler. There is some debug information still missing such as: frame

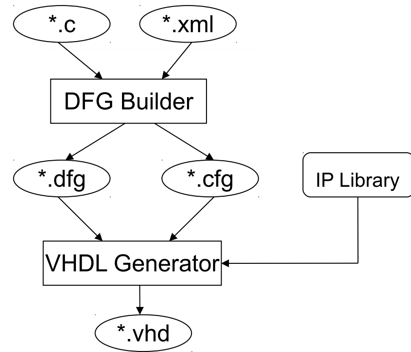


Fig. 2.34 The DWARV toolset

and variable info. However debugging location information has been produced, so it's possible put line/file or function breakpoint.

2.8.3 *Hardware Generation*

In this section, we present the Delft Workbench Automated Reconfigurable VHDL Generator (DWARV) C-to-VHDL generation toolset. The purpose of the compiler is to exploit available parallelism of algorithms and generate designs suitable for hardware/software co-execution. This chapter is structures as follows: First, an overview of the toolset and description of the modules it is composed of is given (Sect. 2). An explanation of the current C subset it accepts and a brief overview of possible transformation needed to make c-functions DWARV-compatible is given in Sect. 3. Finally, in Sect. 4, we will present results obtained when using the toolset with the Molen computing organisation.

DWARV Compiler

This subsection is focused on the design and implementation of the DWARV toolset for automatic generation of VHDL designs from C code considering execution on a hardware prototyping platform. We present the design flow of the DWARV toolset and explain the main components in details.

The main objective of the DWARV toolset is generation of VHDL designs from C code considering execution on a real hardware prototyping platform and without limitations on the application domain. The toolset consists of two modules, namely Data-Flow Graph (DFG) Builder and VHDL Generator, depicted in Fig. 2.34.

The input of the toolset is a pragma annotated C code. The pragma annotation identifies the C code segments to be translated into VHDL code. Currently, the translation is performed at function-level. Hence, the pragma annotation is placed just prior the function signature.

The input C code is processed by the DFG Builder. This module is implemented as a pass within the SUIF2 compiler framework [47]. Its current purpose is to transform the C code into a data-flow representation suitable for hardware generation. The SUIF2 front-end is used to generate SUIF intermediate representation (IR) of the input file. A series of code transformations are applied by the DFG Builder:

1. The code is transformed into static single assignment (SSA) form;
2. Scalar replacement is performed. During, the scalar replacement pass, the memory access expressions are analyzed and if necessary and possibly, un-aliased;
3. A second SSA pass goes over the newly generated scalar variables.

After the code is normalized, the actual data-flow graph construction is performed. For that purpose, the SUIF IR is traversed and for each operation, a corresponding DFG node is created. The edges, corresponding to the data-dependencies between the operations are also constructed. During this traversal, if-conversion is performed. In addition, the precedence order between the memory accesses as well as between the operations within the loop bodies is analyzed and the corresponding edges are inserted. As a final optimization, common sub-expression and dead code elimination are performed. The output of the DFG Builder module is a Hierarchical Data-Flow Graph (HDFG), serialized into a binary file (*.dfg).

The HDFG, used as intermediate representation, is a directed cyclic graph $G\langle V, E\rangle$. The vertices of the graph are divided into two major types: simple and compound. The simple nodes correspond to the arithmetic and logic operations, the memory transfers, the input/output parameters transfers, the constants, and the registers. Simple nodes corresponding to registers in the graph represent only the function parameters and the local variables used to transfer values produced in one loop iteration and consumed in the next iteration. The compound nodes correspond to the loops in the function body. These nodes contain the sub-HDFG of the loop body.

The edges of the graph are also two types: data-dependency edges and precedence edges. A data-dependency edge $e(v \to u), v \in V, u \in V$, indicates that the value produced by node v is consumed by node u. A precedence edge $e(v \to u), v \in V, u \in V$, indicates that the operation(s) in node v has to complete before the operation(s) in node u is initiated. The precedence edges prevent a value, prepared for the next loop iteration to be consumed in the same loop iteration. These edges are also used to order the (possibly) dependent memory accesses.

The HDFG is the input of the second module in the toolset called VHDL Generator. This module is implemented as a stand-alone Linux executable. Its current purpose is to schedule the input graph and to generate the final VHDL code. The performed scheduling is As-Soon-As-Possible. During the scheduling, each executable node, except the memory and parameters transfer and loop nodes, is assumed to take one cycle. An operation is scheduled at a given level (cycle) if all node inputs are already available (scheduled at preceding cycles) and there are no precedence constraints. The scheduling of the loop nodes is merged with the scheduling of the rest of the graph. These nodes are considered as normal nodes, until they are ready to be scheduled. When a loop node is ready to be scheduled, first all other ready nodes

are scheduled. Then the scheduling of the upper-level graph nodes is suspended and the scheduling of the compound node sub-graph is started. When the loop body is completely scheduled, the scheduling of the upper-level nodes is resumed.

The number of cycles, necessary for the memory and parameters transfers is provided as an additional input to the DFG Builder. These data are specified in the configuration file (*.xml) shown in Fig. 2.34. This file also containing the memory and the register banks bandwidth and the address size of the corresponding busses. As additional configuration parameters, the endianess of the system and the sizes of the standard data types are also specified in this file. The xml file is transformed in a text file (*.cfg) that is more suitable for processing by the VHDL Generator.

The last block in Fig. 2.34 constitutes the IP Library. The purpose of this is to provide VHDL components (primitives and cores) that are functionally equivalent to C code fragments which can not be translated automatically to a very efficient VHDL code. As an example of what is described in this library, consider the floating point division of two variables in c code. This operation would be translated to VHDL by instantiating the **fp_sp_div** core that is described in the IP Library, e.g. number of cycles the operation takes, port names of the inputs and output, size of the operands.

The generated output VHDL code represents a FSM-based design. Only one FSM is instantiated for the entire design. The transition between the states is consecutive, unless a loop-back transition is performed. The generated VHDL code is RTL as specified in IEEE-Std 1076.6-2004 [18] and uses the numeric_std synthesis packages [19]. In addition, the RTL designs are generated with the MOLEN CCU interface [28, 29, 51], which allows actual execution on a real hardware prototype platform.

C Language and Restrictions

An objective of the toolset described above is to provide support for almost all standard C-constructs that can be used in the input C code. Nevertheless, in the current version of the toolset several syntax limitations are imposed. These limitations are listed below. A note should be made that the listed restrictions apply only to the functions translated into VHDL code:

- Data types: the supported data types are integer (up to 32-bit) and one-dimensional arrays and pointers. There is not support for 64-bit data types, multi-dimensional arrays and pointers, structures, and unions;
- Storage types: only auto local storage type is supported with limitation for arrays initialization;
- Expressions: there is full support for the arithmetic and logic operations. One-dimensional array subscripting and indirection operation as memory access are also supported. There is no support for function calls, field selection, and address-of operation;
- Statements: The expression statement is limited to the supported expressions, the selection statements are limited to if-selection, and the iteration statements are

Table 2.8 C language support—data types

Data types	Current support	Future support	Future work
Integer types	long long not supported	Full	Use other compiler framework
Real FP types	Supported	Full	N/A
Complex types	Not supported	Not supported	N/A
Pointer types	1D, memory location	ND, local un-aliasing	Pointer analysis
Aggregate types	1d arrays	Full	Data manipulation extension
Unions	Not supported	Full	Data manipulation extension

Table 2.9 C language support—storage

Storage	Current support	Future support	Future work
Auto/local	No array initialization	Full for supported data types	VHDL model extension
Auto/global	Not supported	Constant	VHDL model extension
Static	Not supported	Not supported	N/A
Extern	Not supported	Not supported	N/A
Register	Ignored	Ignored	N/A
Typedef	Full	Full	N/A

Table 2.10 C language support—expressions

Expressions	Current support	Future support	Future work
Arithmetic and logic	FOR & IF	Full	Expression restore analysis extension
Function calls	Not supported	Partial	Function analysis and VHDL extension
Array subscripting	1D	Full	Data manipulation extension
Field selection	Not supported	Full	Data manipulation extension
Address-of	Not supported	Full	Data manipulation extension
Indirection	1D mem	ND mem	Pointer analysis

limited to for-statements. There is no support for jump and labeled statements. The compound statements are fully supported.

Although currently limitations on the input C code are imposed and no sophisticated optimizations are implemented, the toolset is able to translate kernels from different application domains that exhibit different characteristics. In addition, performance improvement over pure software execution is also observed [61]. For a detailed overview of the current C language support, the reader is referred to Tables 2.8, 2.9, 2.10, 2.11.

Although the DWARV toolset does not restrict the application domain, based on the presented C Language limitations above (see Table 2.8), some transformations still have to be performed on the original c-code to make it compliant with the cur-

Table 2.11 C language support—statements

Statements	Current support	Future support	Future work
Expression	Limited to supported expr	Limited to supported expr	N/A
Labeled	Not supported	Switch-case	Data-flow analysis extension
Jump	Not supported	(Switch-) break & continue	Data-flow analysis extension
Selection	If-statement	Full	Data-flow analysis extension
Iteration	For-loop	Full	Data-flow analysis extension
Compound	Full	Full	N/A

rent version of the DWARV compiler. In the future, these restrictions will be relaxed. The transformations needed to make a function DWARV-compliant are summarized below:

- The code to be transformed to VHDL has to be extracted as annotated C function.
- All external data have to be passed as function parameters.
- Multi-dimensional array accesses have to be transformed to 1D memory accesses.
- Pointers are interpreted as memory accesses (hence, no pointers to local data).
- Switch statements have to be re-written as series of if-statements.
- While and do-while loops have to be re-written as for-loops.
- Function calls have to be inlined.
- Structures or unions have to be serialized as function parameters.

Results

For the evaluation of the DWARV compiler, kernels from various application domains identified as candidates for acceleration were used. These generated designs were implemented in Xilinx's VirtexII Pro XC2VP30 FPGA and Virtex-4 XC4VFX100 and the carried experiments on the MOLEN polymorphic processor prototype [28, 29] suggest overall application speedups between 1.4x and 6.8x, corresponding to 13% to 94% of the theoretically achievable maximums, constituted by Amdahl's law [61]. For a more detailed explanation of the carried experiments and results obtained, the reader is referred to the applications chapter.

Hardware Cores for Arithmetic Operations

Many complex arithmetic operations and functions can be more efficiently implemented in hardware than software, when utilizing optimized IP cores designed from FPGA vendors. For example, Xilinx provides a complete IP core suite [58] for all widely used mathematical operations, ranging from simple integer multipliers and dividers, to customizable double precision division and square root. These IP cores are finely-tuned for speed or area optimization, thus providing robust solutions that can be used as sub-modules for larger designs. Based on this fact, we developed

a hardware components library that consists of all widely-used mathematical functions. The latter are built using various Xilinx IP cores. In the next section, we describe which functions are supported by pre-designed hardware modules and elaborate on their specifications.

Operations Supported by the Library Components Table 2.12 shows all the operations that are supported by the library components. The latter can be divided into three main categories, based on the operands used:

- IEEE Floating-Point Arithmetic Single Precision Format (32 bits);
- IEEE Floating-Point Arithmetic Double Precision Format (64 bits);
- Integer (32 bits).

As it is shown in Table 2.12, all mathematical operations are supported for both floating point formats. The reason we decided to support these operations with hardware modules, is that they require less cycles when executed with custom hardware modules, than executed on an embedded PowerPC processor in emulation mode. In addition, we developed hardware modules that perform various types of comparisons between two floating point numbers, like $>$ or $>=$. All these hardware accelerators can be designed to be also pipelined, thus reducing even more the application total execution time [55].

Except from all mathematical operations, we developed custom accelerators for functions that are commonly used in software applications written in C. These functions are also shown in Table 2.12, while Table 2.13 describes each one of them.

Implementation of the Hardware Modules In order to implement all library elements we use the Xilinx ISE 9.2i CAD tools. Xilinx provides dedicated IP cores [56] that utilize the IEEE 754 Standard [20] for all basic mathematical operations, like addition, subtraction, multiplication, division and square root. We used these IP cores as the fundamental building block for all single and double precision hardware accelerators. During the development of each library element, we tried to keep a trade-off between its latency and maximum operating frequency.

The hArtes hardware platform accommodates a Virtex4 FX100 FPGA with an external clock of 125 MHz, thus all hardware accelerators should be able to operate at least to that frequency when mapped onto the FPGA. On the other hand, increasing an element maximum operating frequency much more than 125 MHz, would introduce additional latency cycles. In this case, the element performance would be degraded, since it would never have to operate at a frequency more than 125 MHz. Based on these facts, all hardware accelerators were designed with an operating frequency up to 150 MHz, in order to make sure that they would not introduce any implementation bottlenecks when mapped with other hardware sub-modules. Furthermore, every design is fully pipelined, where a new instruction can be issued every clock cycle.

Finally, regarding the FPGA resource utilization, we tried to exploit as much as possible the dedicated XtremeDSP slices [57]. This way, we achieved two major goals: The first one is that, because of the hardwired XtremeDSP slices, all accelerators could easily operate at the target frequency of 150 MHz. The second one is

Table 2.12 hArtes library components specifications

Precision	Operation
single	$x+y$, $x-y$, $x*y$, x/y, $\sqrt{x}$, $x\%y$, $x>y$, $x<y$, $x>=y$, $x<=y$, $x==y$, $x!=y$, neg(x), round(x), floor(x), ceiling(x), x2int, x2short, x2char, zero(x)
double	$x+y$, $x-y$, $x*y$, x/y, $\sqrt{x}$, $x\%y$, $x>y$, $x<y$, $x>=y$, $x<=y$, $x==y$, $x!=y$, neg(x), round(x), floor(x), ceiling(x), x2int, x2short, x2char, zero(x)
int	x/y, x2fp_sp, x2fp_dp, $x\%y$
short	x/y, $x\%y$
char	x/y, $x\%y$

Table 2.13 hArtes library components specifications

Function	Explanation
round(x)	If the decimal part of x is >=0.5, then round(x) = $\lceil x \rceil$, else round(x) = $\lfloor x \rfloor$
floor(x)	floor(x) = $\lfloor x \rfloor$
ceiling(x)	ceiling(x) = $\lceil x \rceil$
x2int(x)	Keep x integer part and convert it to integer format
x2short(x)	Keep x integer part and convert it to short format
x2char(x)	Keep x integer part and convert it to char format
zero(x)	If $x == 0.0$ or $x == -0.0$ then zero(x) = 1, else zero(x) = 0

that we leave more FPGA slices available to map other sub-modules, that are automatically generated by the DWARV C-to-VHDL tool. In practise, the majority of the library elements occupies only 1% of the hArtes platform Virtex4 FPGA regular slices.

2.8.4 Linking

The hArtes linker produces a single executable, linking all the contributes coming from the different PE' compilation chains.

We use the standard GNU Linker (LD) targeted for ARM Linux and a customized hArtes Linker script (*target_dek_linux.ld*) that instructs the Linker on how to build the hArtes executable from the different PE's input sections. Each PE has an associated *text* (program section), *bss* (not initialized data), *data* (initialized data) section, plus some additional sections that correspond to platform's shared memories (if any).

The Linker and the customized linker scripts have been integrated inside the hArtes framework as last hArtes compilation pass, as shown in Fig. 2.35. The Linker script, at linking time, generates additional global GPP variables in the hArtes executable that describe the PE sections that must be loaded by the hArtes runtime. Once the executable starts the hArtes runtime uses these variables to retrieve addresses and size of the sections that must be loaded.

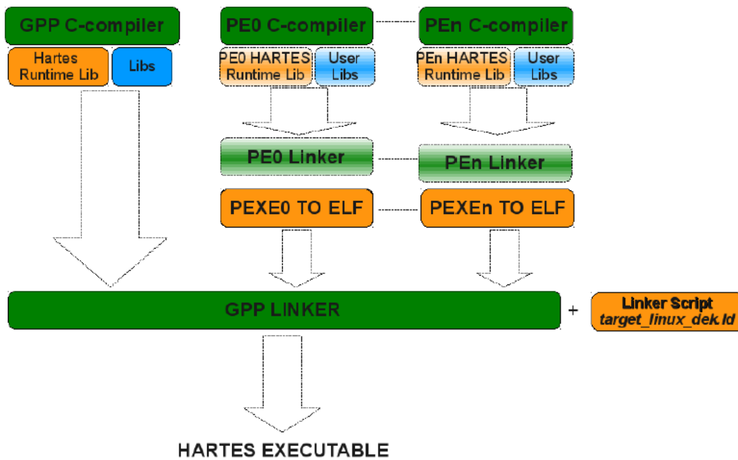


Fig. 2.35 Representation of the linking process in the hArtes toolchain

2.9 Runtime Support

The *hArtes Runtime* provides a simple device driver interface for programs to connect to the underlying Processing Elements and GPP Operating System. The hArtes runtime it is also integrated with the GPP ANSI C standard library and provides hosted I/O services to the PE (printf, fopen.).

Different functionalities have been implemented inside the hArtes Runtime to support the runtime analysis of the application on the target platform. The main features are:

- memory allocation
- profiling support
- debug support

2.9.1 Memory Allocation

Memory allocation is performed by the runtime support. It keeps trace of allocations performed by the PEs and the GPP, in order to figure out how to move and optimize data to/from GPP and PEs. It also performs basic checking on pointers passed in remote calls.

2.9.2 Profiling and Benchmarking Support

The aim of the hArtes toolchain is to optimize an application by partitioning it into pieces and mapping it onto the heterogeneous PEs of a given architecture. Profiling and benchmarking activities are thus fundamental to understand to improve harts toolchain outputs by analyzing the execution of the current status of the application.

Manual Instrumentation of the Code

The hArtes runtime provides a very accurate function to get time, that is *hget_tick*. This function can be used in the code to compute the actual time expressed in ticks (order on NS). This function cannot be used to measure time greater than 4 s (32 bit counter). Use standard timer.h functions to measure seconds.

Automatic Instrumentation of the Code

The HGCC compiler provides options to automatically instrument code, currently supporting:

- `-dspstat` that instruments all DSP remote calls.
- `-fpgastat` that instruments all FPGA calls.
- `-gppstat` that instruments all GPP calls.

Measure Power Consumption

Some HW targets have the support to measure currents. For instance the DEB board can measure power consumption via a SPI current sensor on the board.

2.9.3 Debug Support

We used **`Gnu DeBugger`**(GDB) targeted for ARM Linux, plus an additional hArtes patch to support multiple PE. This patch essentially adds an additional signal handler coming from the hArtes Runtime. This signal is treated as a standard ARM Linux breakpoint trap, interrupting the application flow execution, meanwhile the hArtes Runtime suspends the execution of other PEs (if the HW provides support). In this way a "soft" synchronization is realized and the user can inspect a stopped application.

In the current implementation we are supporting ARM + MAGIC(DSP) debugging, that is because FPGA does not support, at the moment, any kind of debugging facilities (debugging symbols, HW mechanism to suspend execution), so remote FPGA calls behave as a black box, providing the possibility to inspect only inputs and outputs before and after remote execution.

At the moment it is possible:

- To inspect the hArtes address space, including PE I/O spaces;
- to add DSP breakpoints;
- to inspect DSP program frame;
- to read/write and decode DSP registers;
- to provide GPP/DSP inter block (a breakpoint or an event on the GPP blocks the DSP and vice versa)

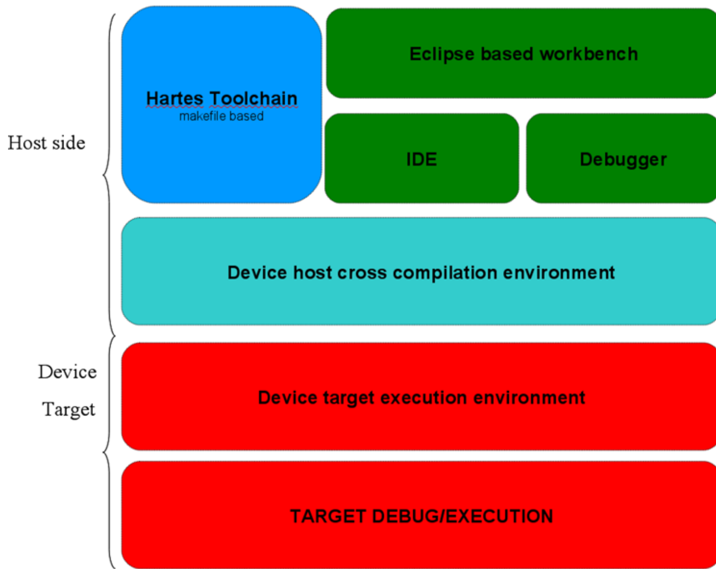


Fig. 2.36 Hartes framework

The hArtes runtime keeps trace of every thread and remote function is being executed. The user can dump the status of a running hArtes application just sending a HUP signal (see POSIX signals) to it. Exceptions or abnormal situations caught by the runtime generate an immediate dump of the process status.

2.10 hArtes Framework

The hArtes Framework makes it easy to develop Linux applications for GPP-based platforms (in our case ARM). It reduces the learning curve, shortens the development and testing cycle, and helps building reliable applications quickly.

In the hArtes view, the program flow is executed on the GPP, PEs are used to speed up computing intensive tasks. The GPP runs Linux. For the GPP the hArtes toolchain uses a standard GNU toolchain plus an additional hArtes patch, in order to support hArtes XML and pragma annotations. An overview of hArtes framework is shown in Fig. 2.36.

2.10.1 Text Based Workbench

The hArtes projects are makefile based, the makefile contains all the information needed to build an hArtes application, starting from sources. Once the hArtes framework is installed, the user has to provide application sources and a simple makefile that essentially lists the files to be passed through the hArtes toolchain.

hArtes Project Compilation The creation of an hArtes project requires a very simple very simple makefile.

The following example shows a typical hArtes makefile for a system composed of a GPP and two PEs (DSP and FPGA):

```
PROJECT = hArtesProject

## put here the common includes -I<DIR> and defines
## -D<your Define> for the project
HARTESCOMMON =
################ DSE TOOLS SETTINGS ###################
## put here the sources that you want to pass to the hArtes
## toolchain
DSECOMPILESRC =
## put here the flags for zebu (partioner)
DSEPARTITIONFLAGS =
## put here the flags for hArmonics (mapper)
DSEMAPPINGFLAGS =
########################################################


################ Synthesis TOOLS SETTINGS ###################

####### HGCC COMPILER [GPP] #######
## put here the source you want compile for the ARM
COMPILESRC =
## GPP Compiler flags (include directories, defines..)
CFLAGS =
## GPP Linker flags (include libraries, additional libraries)
LDFLAGS =

####### CHESS COMPILER [DSP]#######
## put here the source you want compile for the DSP
DSPCOMPILESRC =
## DSP Compiler flags (include directories, defines..)
DSPCCFLAGS =
## DSP Linker flags (include libraries, additional libraries)
DSPLDFLAGS    =

####### DWARV COMPILER [FPGA]#######
## put here the source you want compile for the FPGA
FPGACOMPILESRC =
## FPGA Compiler flags (include directories, defines..)
FPGACCFLAGS =

include $(GNAMDIR)/config.mak
```

The above makefile can be generated by the `hproject_create.sh` command, available once installed the hArtes framework.

The same project can be compiled in three different configurations:

- No mapping (low effort, low results)
- Manual mapping (medium effort, best results)
- Completely automatic mapping (low effort, good/best results)

No Mapping This means that the application is compiled entirely for the GPP.

This pass will provide a working reference and allow to evaluate the baseline performance.

Manual Mapping (Martes) The user does a manual partitioning and mapping, by using `#pragma` for mapping and by putting the sources on the appropriate PE section. This pass is useful to evaluate the maximum performance of the application on the system.

Completely Automatic Mapping This kind of project is entirely managed by the hArtes toolchain that will partition and will map the application on the available PEs.

This pass is useful to evaluate the performance obtained with a very low human effort.

2.10.2 hArtes Execution

This pass is very intuitive, since it requires just to copy to the target board the single executable image produced by the toolchain and then execute it.

2.10.3 Graphical Based Workbench

The hArtes Eclipse workbench is built on the standard Eclipse development environment providing outstanding windows management, project management, and C/C++ source code editing tools. We provided additional plugins to manage hArtes projects. This plugin allows the creation of hArtes projects with two possible configurations: no mapping and completely automatic mapping.

The Eclipse IDE's fully featured C/C++ source editor provides syntax checking.

- Outline view which lists functions, variables, and declarations
- Highlights syntax errors in your C/C++ source code
- Configurable syntax colorization and code formatting for C/C++ and ARM/Thumb/Thumb2 assembly
- Full change history which can be integrated with popular source code control systems, including CVS and SVN
- Graphical configuration of parameters in the source code via menus and pull-down lists

File Transfer to Target The hArtes Eclipse distribution includes a Remote System Explorer (RSE) perspective for easy transfer of applications and libraries to the Linux file system on the target.

RSE enables the host computer to access the Linux file system on hardware targets.

- FTP connection to the target to explore its file system, create new folders, and drag & drop files from the host machine
- Open files on the target's file system by double-clicking on them in the FTP view. Edit them within Eclipse and save them directly to the target's file system
- Shell and terminal windows enable running Linux commands on the target system without a monitor and keyboard
- Display of a list of processes running on the target

Window Management The flexible window management system in Eclipse enables the optimal use of the visual workspace.

- Support for multiple source code and debugger views
- Arrange windows: floating (detached), docked, tabbed, or minimized into the Fast View bar
- Support of multi-screen set-ups by dragging and dropping detached windows to additional monitors

Debugger Overview The Eclipse debugger is a powerful graphical debugger supporting end-to-end development of GPP Linux-based systems. It makes it easy to debug Linux applications with its comprehensive and intuitive Eclipse-based views, including synchronized source and disassembly, memory, registers, variables, threads, call stack, and conditional breakpoints.

Target Connection The hArtes debugger automates target connection, application download and debugger connection to *gdbserver* on supported platforms.

- The debugger connects to a *gdbserver* debug agent running on the target, using an Ethernet cable;
- A launcher panel automates the download of Linux applications to the hardware target by using a telnet or ssh connection;
- The debugger can connect to GPP Linux target. The Remote System Explorer (RSE) is used to manually transfer files to the target, open a terminal window, and start *gdbserver*.

Run Control Control the target's execution with high (C/C++) and low (assembler) level single-stepping and powerful conditional breakpoints.

All aspects of CPU operation can be controlled from Eclipse.

- Run control: run, stop, step through source code and disassembly
- Set an unlimited number of software breakpoints by double clicking in the source or disassembly views
- Conditional breakpoints halt the processor when hit a pre-defined number of times or when a condition is true
- Assignment of actions to breakpoints, allows message logging, update views, or output messages
- If the debugger detects a slow target connection it disables the system views until the user stops stepping. This enables fast single-stepping operation

System Views The hArtes Eclipse debugger provides access to the resources inside the target device, including processor and peripheral registers, code, memory, and variables.

Synchronized source code and disassembly views provide easy application debug since they provide the possibility to

- Open as many system views at the same type as necessary. Freeze them for easy comparison of their contents over time
- Color code synchronized source code and disassembly for easy debug of highly optimized C/C++ code
- View and modify C variables and C++ classes, whether local to a function or global
- View a list of current threads and the call stack for each thread. Click on a thread or a call stack entry to focus the debugger views on that frame
- Use expressions in C-syntax on any of the system views. For example, write to a memory location the contents pointed at by pointer ptr by typing =*ptr

2.10.4 *hArtes Debugging Customization*

Debugging an hArtes application it's not easy. However we added to **GDB**, the capability to debug a GPP + DSP + (FPGA) hArtes application, as indicated in the Runtime support section.

In the following chapters will be described techniques to debug an hArtes application for GPP + DSP.

For the GPP and DSP we use the GNU GDB targeted for ARM and customized for hArtes. Both local (target) or remote debugging configuration are possible.

Local debugging is performed directly on the Target HW (HHP or DEB) by running `gdb` , while remote debugging is performed by running *`gdbserver`* on the target HW and **gdb** on the local host.

New GDB Commands For the DSP new commands have been added:

- info hframe that gives information about the current hArtes frame
- mreset that performs a DSP HW reset

2.11 Conclusion

This chapter aimed at presenting a global overview of the hArtes methodology to map a C-application onto an heterogeneous reconfigurable platform composed of different processors and reconfigurable logic. The different features provided by the different tools provide a large set of options to the designer to explore the design space, from the algorithm down to the mapping decisions. The main restrictions of the actual design flow is the assumption of a shared memory model and a fork/join programming model, on which OpenMP is based.

References

1. Agosta, G., Bruschi, F., Sciuto, D.: Static analysis of transaction-level models. In: DAC 03: Proceedings of the 40th Conference on Design Automation, pp. 448–453 (2003)
2. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.: Metropolis: an integrated electronic system design environment. Computer **36**, 45–52 (2003)
3. Banerjee, U., Eigenmann, R., Nicolau, A., Padua, D.A.: Automatic program parallelization. Proc. IEEE **81**(2), 211–243 (1993)
4. Beltrame, G., Fossati, L., Sciuto, D.: ReSP: a nonintrusive transaction-level reflective MPSoC simulation platform for design space exploration. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **28**(12), 1857–1869 (2009)
5. Carr, S., Kennedy, K.: Scalar replacement in the presence of conditional control flow. Softw. Pract. Exp. **24**(1), 51–77 (1994)
6. Coware N2C: http://www.coware.com/products/
7. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proc. IEEE **91**, 127–144 (2003)
8. Ferrandi, F., Lattuada, M., Pilato, C., Tumeo, A.: Performance modeling of parallel applications on MPSoCs. In: Proceedings of IEEE International Symposium on System-on-Chip 2009 (SOC 2009), Tampere, Finland, pp. 64–67 (2009)
9. Ferrandi, F., Lattuada, M., Pilato, C., Tumeo, A.: Performance estimation for task graphs combining sequential path profiling and control dependence regions. In: Proceedings of ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2009), Cambridge, MA, USA, pp. 131–140 (2009)
10. Ferrandi, F., Pilato, C., Tumeo, A., Sciuto, D.: Mapping and scheduling of parallel C applications with ant colony optimization onto heterogeneous reconfigurable MPSoCs. In: Proceedings of IEEE Asia and South Pacific Design Automation Conference 2010 (ASPDAC 2010), Taipei, Taiwan, pp. 799–804 (2010)
11. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)
12. Franke, B., OBoyle, M.F.P.: A complete compiler approach to auto-parallelizing C programs for multi-DSP systems. IEEE Trans. Parallel Distrib. Syst. **16**(3), 234–245 (2005)
13. GCC, the GNU Compiler Collection, version 4.3: http://gcc.gnu.org/
14. Girkar, M., Polychronopoulos, C.D.: Automatic extraction of functional parallelism from ordinary programs. IEEE Trans. Parallel Distrib. Syst. **3**(2), 166–178 (1992)
15. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: a call graph execution profiler. SIGPLAN Not. **17**(6), 120–126 (1982)
16. Hou, E.S.H., Ansari, N., Ren, H.: A genetic algorithm for multiprocessor scheduling. IEEE Trans. Parallel Distrib. Syst. **5**, 113–120 (1994)
17. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
18. IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, IEEE Std 1076.6-2004, available online at http://ieeexplore.ieee.org/servlet/opac?punumber=9308
19. IEEE Standard VHDL Synthesis Packages, IEEE Std 1076.3-1997, available online at http://ieeexplore.ieee.org/servlet/opac?punumber=4593
20. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008, vol. 29, pp. 1–58 (2008)
21. Ierotheou, C.S., Johnson, S.P., Cross, M., Leggett, P.F.: Computer aided parallelisation tools (CAPTools)—conceptual overview and performance on the parallelisation of structured mesh codes. Parallel Comput. **22**(2), 163–195 (1996)
22. Ismail, T.B., Abid, M., Jerraya, A.: COSMOS: a codesign approach for communicating systems. In: CODES'94: Proceedings of the 3rd International Workshop on Hardware/Software Co-design, pp. 17–24 (1994)
23. Jin, H., Frumkin, M.A., Yan, J.: Automatic generation of OpenMP directives and its application to computational fluid dynamics codes. In: Proceedings of the Third International Symposium on High Performance Computing (ISHPC 00), London, UK, pp. 440–456. Springer, Berlin (2000)

24. Jin, H., Jost, G., Yan, J., Ayguade, E., Gonzalez, M., Martorell, X.: Automatic multilevel parallelization using OpenMP. Sci. Program. **11**(2), 177–190 (2003)
25. Johnson, T.A., Eigenmann, R., Vijaykumar, T.N.: Min-cut program decomposition for thread-level speculation. In: Programming Language Design and Implementation, Washington, DC, USA (2004)
26. Kianzad, V., Bhattacharyya, S.S.: Efficient techniques for clustering and scheduling onto embedded multiprocessors. IEEE Trans. Parallel Distrib. Syst. **17**(17), 667–680 (2006)
27. Kim, S.J., Browne, J.C.: A general approach to mapping of parallel computation upon multiprocessor architectures. In: Int. Conference on Parallel Processing, pp. 1–8 (1988)
28. Kuzmanov, G.K., Gaydadjiev, G.N., Vassiliadis, S.: The Virtex II Pro MOLEN processor. In: Proceedings of International Workshop on Computer Systems: Architectures, Modelling, and Simulation (SAMOS), Samos, Greece, July. LNCS, vol. 3133, pp. 192–202 (2004)
29. Kuzmanov, G., Gaydadjiev, G.N., Vassiliadis, S.: The Molen media processor: design and evaluation. In: WASP'05 (2005)
30. Lam, Y.M., Coutinho, J.G.F., Luk, W., Leong, P.H.W.: Integrated hardware/software codesign for heterogeneous computing systems. In: Proceedings of the Southern Programmable Logic Conference, pp. 217–220 (2008)
31. Lam, Y.M., Coutinho, J.G.F., Ho, C.H., Leong, P.H.W., Luk, W.: Multi-loop parallelisation using unrolling and fission. Int. J. Reconfigurable Comput. **2010**, 1–10 (2010). ISSN 1687-7195
32. Liu, Q., et al.: Optimising designs by combining model-based transformations and pattern-based transformations. In: Proceedings of International Conference on Field Programmable Logic and Applications (2009)
33. Luis, J.P., Carvalho, C.G., Delgado, J.C.: Parallelism extraction in acyclic code. In: Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing (PDP'96), Braga, January, pp. 437–447 (1996)
34. Luk, W., Coutinho, J.G.F., Todman, T., Lam, Y.M., Osborne, W., Susanto, K.W., Liu, Q., Wong, W.S.: A high-level compilation toolchain for heterogeneous systems. In: Proceedings of IEEE International SOC Conference, September 2009
35. Newburn, C.J., Shen, J.P.: Automatic partitioning of signal processing programs for symmetric multiprocessors. In: PACT 96 (1996)
36. Novillo, D.: A new optimization infrastructure for GCC. In: GCC Developers Summit, Ottawa (2003)
37. Novillo, D.: Design and implementation of tree SSA. In: GCC Developers Summit, Ottawa (2004)
38. Osborne, W.G., Coutinho, J.G.F., Luk, W., Mencer, O.: Power and branch aware word-length optimization. In: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 129–138. IEEE Comput. Soc., Los Alamitos (2008)
39. Ottoni, G., Rangan, R., Stoler, A., Bridges, M.J., August, D.I.: From sequential programs to concurrent threads. IEEE Comput. Archit. Lett. **5**(1), 2 (2006)
40. Pilato, C., Tumeo, A., Palermo, G., Ferrandi, F., Lanzi, P.L., Sciuto, D.: Improving evolutionary exploration to area-time optimization of FPGA designs. J. Syst. Archit., Embed. Syst. Design **54**(11), 1046–1057 (2008)
41. Qin, W., Malik, S.: Flexible and formal modeling of microprocessors with application to retargetable simulation. In: DATE 03: Proceedings of Conference on Design, Automation and Test in Europe, pp. 556–561 (2003)
42. Quinlan, D.J.: ROSE: compiler support for object-oriented frameworks. In: Proceedings of Conference on Parallel Compilers (CPC2000) (2000)
43. Ramalingam, G.: Identifying loops in almost linear time. ACM Trans. Program. Lang. Syst. **21**(2), 175–188 (1999)
44. Sarkar, V.: Partitioning and scheduling parallel programs for multiprocessors. In: Research Monographs in Parallel and Distributed Processing (1989)
45. Sato, M.: OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors. In: Proceedings of ISSS'02, pp. 109–111 (2002)

46. Schirrmeister, F.: Cadence virtual component co-design: an environment for system design and its application to functional specification and architectural selection for automotive systems. Cadence Design Systems, Inc. (2000)
47. SUIF2 Compiler System: available online at http://suif.stanford.edu/suif/suif2
48. Susanto, K., Luk, W., Coutinho, J.G.F., Todman, T.J.: Design validation by symbolic simulation and equivalence checking: a case study in memory optimisation for image manipulation. In: Proceedings of 35th Conference on Current Trends in Theory and Practice of Computer Science (2009)
49. Todman, T., Liu, Q., Luk, W., Constantinides, G.A.: A scripting engine for combining design transformations. In: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), US, May 2010
50. Vallerio, K.S., Jha, N.K.: Task graph extraction for embedded system synthesis. In: Proceedings of the 16th International Conference on VLSI Design (VLSID 03), Washington, DC, USA, p. 480. IEEE Comput. Soc., Los Alamitos (2003)
51. Vassiliadis, S., Wong, S., Cotofana, S.: The MOLEN $\rho\mu$-coded processor. In: Proceedings of International Conference on Field-Programmable Logic and Applications (FPL). LNCS, vol. 2147, pp. 275–285. Springer, Berlin (2001)
52. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K.L.M., Kuzmanov, G.K., Moscu Panainte, E.: The MOLEN polymorphic processor. IEEE Trans. Comput. **53**(11), 1363–1375 (2004)
53. Wang, W.K.: Process scheduling using genetic algorithms. In: IEEE Symposium on Parallel and Distributed Processing, pp. 638–641 (1995)
54. Wiangtong, T., Cheung, P.Y.K., Luk, W.: Hardware/software codesign: a systematic approach targeting data-intensive applications. IEEE Signal Process. Mag. **22**(3), 14–22 (2005)
55. Xilinx Inc: Virtex-5 APU Floating-Point Unit v1.01a, April 2009
56. Xilinx Inc: Floating-Point Operator v5.0, June 2009
57. Xilinx Inc: XtremeDSP for Virtex-4 FPGAs, May 2008
58. Xilinx Inc: IP Release Notes Guide v1.8, December 2009
59. Xilinx, Inc: Virtex-4 FPGA Configuration User Guide, Xilinx user guide UG071 (2008)
60. Yang, T., Gerasoulis, A.: DSC: scheduling parallel tasks on an unbounded number of processors. IEEE Trans. Parallel Distrib. Syst. **5**, 951–967 (1994)
61. Yankova, Y., Bertels, K., Kuzmanov, G., Gaydadjiev, G., Lu, Y., Vassiliadis, S.: DWARV: DelftWorkBench automated reconfigurable VHDL generator. In: FPL'07 (2007)
62. Zhu, J., Gajski, D.D.: Compiling SpecC for simulation. In: ASP-DAC'01: Proceedings of the 2001 Asia and South Pacific Design Automation Conference, Yokohama, Japan, pp. 57–62 (2001)

Chapter 3
The hArtes Platform

Georgi Kuzmanov, Raffaele Tripiccione, Giacomo Marchiori, and Immacolata Colacicco

This chapter will introduce the hArtes Hardware Platform (hHp, in short). This platform is an heterogeneous reconfigurable computing system, that has been purposedly developed within the project. The platform plays a double role within hArtes: on the one hand, it is a reference target for the complete hArtes toolchain, while on the other hand it provides sufficient computing resources for several high-performance applications in the audio domain. This chapter describes in details the ideas that lay behind the hHp architecture, its detailed implementation and how the hHp meets its requirements within the hArtes project.

3.1 Introduction

Embedded systems have grown steadily in scope complexity and diffusion over the years; today they play an increasingly prominent role in several areas of computing. Embedded systems have become not only more widespread, but also more complex and performing. As more and more complex and demanding functionalities must be supported by just one embedded system, complexity increases and at the same time diverse and conflicting requirements emerge, that in many cases cannot be met efficiently by just one architecture. For instance, traditional processors are not particularly efficient for DSP processing; on the other hand, an FPGA block is very good at handling any streaming component of an application, but DSPs and FPGA-based systems alike are an obviously poor choice if a complex friendly human interface is a must for the entire system.

One solution to this problem that naturally offers to the hardware developer is that of assembling the appropriate mix of heterogeneous building blocks, each based on a specific computational architecture and optimized for specific sub tasks of the entire

R. Tripiccione (✉)
Università di Ferrara, via Saragat 1, 44100 Ferrara, Italy
e-mail: tripiccione@fe.infn.it

K. Bertels (ed.), *Hardware/Software Co-design for Heterogeneous Multi-core Platforms*,
DOI 10.1007/978-94-007-1406-9_3, © Springer Science+Business Media B.V. 2012

application. In this approach, the target application has to be split by a corresponding set of computational tasks (let us call them threads for simplicity); each thread runs on the most appropriate sub-system and exchanges data with all other threads, as needed for the successful operation of the whole system.

The process of splitting a large application in hardware-friendly threads is an obviously complex and laborious process, since it relies on information associated to the computational structure of the application and to the hardware structure of the embedded system that may be not easily available and in any case requires a sound know-how in areas—algorithms and architecture—that are usually not fully commanded by just one developer. Even if we assume that this breakup of the target application has been successfully performed, specific program developments for each thread has to be carried out on the corresponding target hardware subsystem. This means that the development of an application for a specific heterogeneous system is a lengthy, expensive and complicated process. This is why heterogeneous systems have not been popular so far, in spite of their potential advantages.

The hArtes project plans to improve on this situation developing a set of tools able to partition the application, map it onto the appropriate mix of hardware elements and optimize for performance in a largely automated way. The platform that we present here is the workhorse for all demonstration activities within the project, and bears the responsibility of providing the performance that the hArtes software tools have to unleash. As such, the platform must provide the computing muscles needed by the applications while remaining a friendly target for the hArtes tools.

This chapter will proceed as follows: the next section outlines the Molen machine paradigm, which is the preferred scheme to reconfigure the platform for a given specific task; the architecture of the platform is described in the following section, with an extensive explanation of the requirements that the architecture itself is meant to fulfill. The final section describes the firmware and software elements that have been developed to support the operation of the platform.

3.2 The Molen Machine Organisation

The Molen programming paradigm is based on the Molen Machine Organisation (see Fig. 3.1), which defines how a general purpose processor (GPP) interacts with one or more co-processors. The components of the Molen Machine Organisation, which is an instance of tightly coupled processor/co-processor with shared memory, are the following:

- Shared Memory: in this memory, both instructions and data reside. The data memory is accessible by all the processing elements such as the FPGA and the DSP and is used for large data structures. Access to the shared data memory is managed by the Data Memory Mux/Demux.
- Arbiter: the arbiter receives the next instruction to be executed and determines by what processor it needs to be executed and will directs it accordingly.

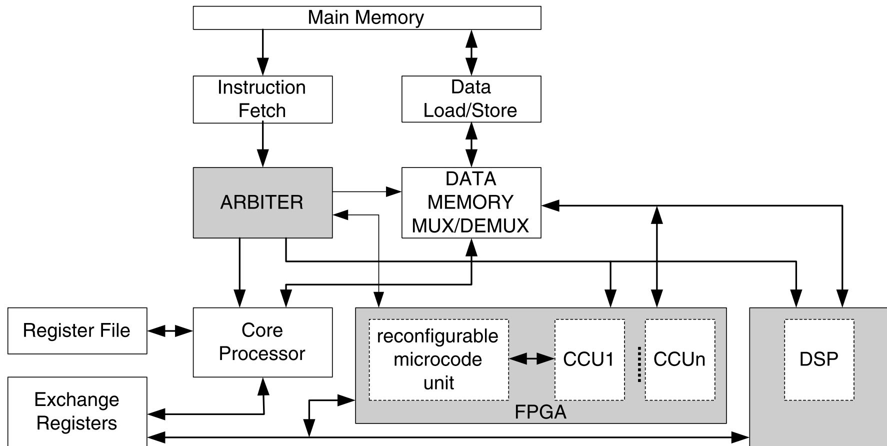


Fig. 3.1 The Molen machine organization

- Register File and Exchange Registers: the GPP has, like any other GPP, its own register file. The Exchange Registers are the interface between the GPP and the co-processors. They are used to transfer small pieces of data between the GPP and the co-processors, such as function parameters, results of computations, pointers to locations in the shared memory, etc.
- Custom Computing Units (CCU) and DSP: on the reconfigurable fabric, one or more custom computing units can be mapped. If required, they can execute in parallel. When supported by the FPGA, partial and runtime reconfiguration can be used to modify the functionality of the CCU's during the program execution.

This Molen architecture [1] can be used for various hardware configurations which in the case of the hArtes project was constituted of an ARM processor, a DSP and an FPGA. The Molen paradigm involves a one time extension of the instruction set architecture (ISA) to implement an arbitrary functionality in the CCU's or the DSP in the hArtes case. The detailed description of the IS-extension is given in [1]. In the context of the hArtes project, we have implemented the following five instructions: SET, EXECUTE, MOVTX, MOVFX and BREAK. The first instruction makes sure that the co-processors are ready. Where the DSP is loaded at the start of the application, the FPGA can be configured at runtime. The appropriate configuration logic is executed when the SET instruction is encountered. The EXECUTE then actually starts the execution by the co-processor. As explained above, exchange registers are used to provide a fast communication channel between the main processor and any of the co-processors. To this purpose, the instruction set is extended with instructions, MOVTX and MOVTX, that move data in and out of those registers.

Even though parallel execution is possible through the use of e.g. the OpenMP annotations, in essence, the Molen programming paradigm conforms to the sequential consistency paradigm meaning that synchronization points are added to guarantee the correctness of the overall sequential execution. These synchronization points are supported by the BREAK instruction where program execution is stalled until

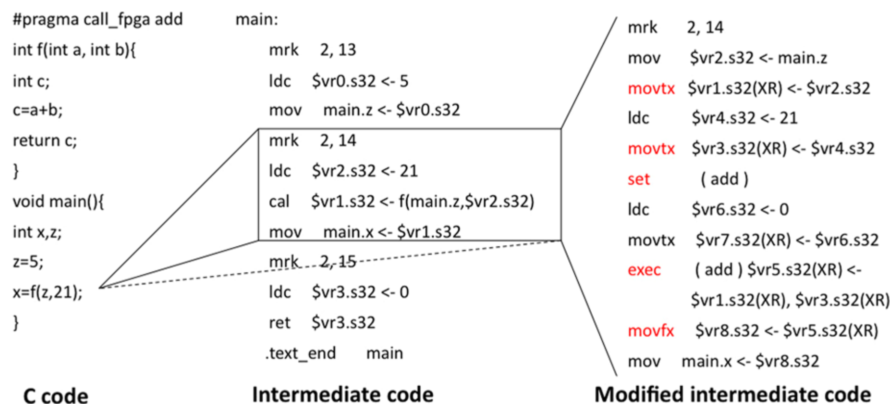


Fig. 3.2 The Molen programming paradigm

all running threads end, after which execution is resumed. Although the Molen Machine was initially proposed specifically for FPGA based hardware platforms, it can be also applied for other kind of co-processor technologies. In the case of hArtes, the hardware platform consists of an ARM or a PowerPC processor, the Diopsis DSP of Atmel, a Xilinx Virtex-4 or an Altera Stratix II FPGA.

Polymorphic Program Execution In order to understand how an application is executed on the Molen platform (see Fig. 3.1), we describe a typical execution flow. As instructions are residing in memory, they are being fetched and decoded by the arbiter. The arbiter determines whether to send the instruction to the main processor or to any of the CCUs. If a SET instruction is decoded, the corresponding configuration bitstream is downloaded into the FPGA. The SET instruction could also be used for the DSP if at runtime it could be reloaded with new instructions. However, the Diopsis DSP is preloaded at the start of the application and those instructions cannot be changed. In case an EXECUTE-instruction is decoded, the arbiter initiates a CCU operation and the control unit in the CCU starts reading the memory pointers from the XREG which are used to address the main data memory. A similar operation is performed in case the DSP is started.

At the software level, the program code is annotated by a set of pragmas, as illustrated in Fig. 3.2. One can clearly identify the position and the sequence of the Molen specific instructions in the modified assembly code generated by the Molen compiler. The essence is that the assembly code for the standard operation is substituted by a call to the hardware, executing the same operation either on a reconfigurable CCU, or on a DSP (in the hArtes context). The specific pragma types considered by the hArtes toolchain are introduced in more details in the section to follow.

3.3 Architecture of the hArtes Platform

The architecture of the hArtes Hardware Platform [4] has been shaped in order to meet two different set of objectives: we wanted the hHp to be an effective computational structure for a number of applications that are obvious candidates for embedded systems, showing the architectural advantages of an heterogeneous system and, at the same time, we wanted to show that this architecturally complex system can be effectively and efficiently targeted by the hArtes toolchain, allowing a quick and easy porting process from the underlying algorithms to the deployed working application. In this section we characterize in more details our requirements and then describe the architecture that matches them.

3.3.1 Requirements

The demonstrator applications envisaged for the platform are focused on two main fields: audio applications for advanced Car Information Systems (CIS) and immersive audio (e.g., teleconferencing). For details, see [2] and [3]. Both applications are centered in the audio domain; they require different levels of complexity, both in terms of computational load (ranging from 2 to more than 10 Gflops) and in terms of I/O requirements (ranging from just a few audio channels up to 64 high-quality, digital audio channels).

The applications that we have outlined above set a number of performance requirements on the target hardware platform:

- single precision floating point performance at the level of 1 to 10 Gflops, mostly, but not limited to, FFT and FIR filter computations;
- power efficiency (at the level of 1W/Gflops or better) is extremely important for embedded systems, so power efficient data-paths have to be used;
- memory in the range of several Gbytes. From the point of view of the application structure, shared memory would be preferred; however the performance penalty associated to a globally shared memory space has to be taken in consideration, so a message-passing structure may be a more efficient solution;
- dedicated I/O structure for up to 64 high quality input and output audio channels, with hardware support for data transfers from/to the audio channels and system memory.

The role of the hHP as the demonstration platform for the hArtes project requires that the system integrates several independent and heterogeneous subsystems, on which target applications will be mapped by the hArtes tool chain. Among the subsystems that we consider for the hHp is a configurable block, for which we anticipate very good performance and efficiency for the data streaming sections of the target applications. We add as a further requirement that all configurable elements in the system must comply with the MOLEN processor organization. The decision to implement the MOLEN paradigm on an heterogeneous system has several advantages and implies several specific problems that will be described later on.

Finally, the constraint that the system must be tested "on-the-road" for in-car applications requires that the platform is able to run almost independently with limited support from traditional PC-based computers, so it must provide standard I/Os (like USB, serial and Ethernet ports) to interface with an independent infrastructure, e.g. an on-board general purpose information system.

3.3.2 Overall Structure

The requirements discussed above directly translate into the following key features:

- integrate several elements of an heterogeneous computer architecture and provide sufficient interconnection between them to support tightly-coupled computational tasks;
- support a modular system structure, starting from a simple and cheap entry-level configuration and extending to more performing systems, offering a larger computational power (some tens of Gflops);
- define an architecture in which new processing elements (that is, additional heterogeneous components) can be incorporated in a compatible way at a later stage of the design process;
- provide adequate input-output channels, both at the general purpose level (e.g., USB and Ethernet connections) and at the level of the specific standards applicable for audio signal processing.

The hHP makes ample use of FPGA technologies, that are obvious implementation options for the configurable segment of the system; we also use the FPGA to implement several non-configurable functions that are relevant for the platform. In general, FPGAs play two key roles within the hHP. First, an FPGA based subsystem controls the data streaming between the dedicated audio channels and the processing kernels. This subsystem receives serial data from up to 64 channels, performs serial to parallel conversion and reformatting and routes the resulting stream directly to the memory interface of each processor; audio streaming from/to the memory is fully supported in hardware, leaving the number crunchers free to perform their main computational tasks. A further even more pivotal role played by FPGAs sees one such component at the heart of each processing element. Here, the FPGA plays a computational role as a configurable processor and also handles the communication channels that connect the processing element among each other, allowing to share data.

3.3.3 Detailed Hardware Description

The system is composed of a certain number (two in the present design) of independent blocks, each of which includes a general-purpose RISC processor, a DSP

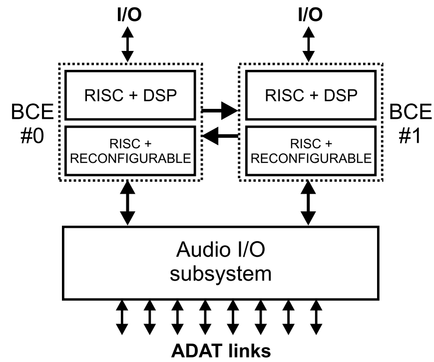


Fig. 3.3 Top level architectural structure of the hArtes Hardware Platform. The system has two independent heterogeneous and configurable processors that communicate among each other and with an audio I/O subsystems; the latter supports several ADAT channels

processor and an application-specific reconfigurable block, that also incorporates a RISC processor. We call this basic building block, consisting of RISC processors and a reconfigurable element, a Basic Configurable Element (BCE).

The top-level typical architecture of the hHP is shown in Fig. 3.3 (for the present case of a system containing two BCE). All BCEs have the same organization; they are able to run any selected thread of a large application, as partitioned by the hArtes tool chain. They are connected by a direct data link with high-bandwidth (200 Mbyte/sec).

The link is used to share data and synchronization information between the two BCEs. However, private (distributed among all its components) memory is also available on each BCE, for faster access to local data. In this way, complex applications sharing global data segments can run on the application with little porting effort. However, tuning for performance is possible, as data segments that do not need to be shared are moved to local memory banks within the appropriate BCE. Dedicated hardware for massive data streaming is also available on the system, as shown in Fig. 3.3 (the "Audio I/O subsystem" block). Up to 64 input audio channels and up to 64 output audio channels are available on $8 + 8$ ADAT interfaces, a standard that defines a digital link designed for high-quality audio transfers.

Figure 3.4 provides a detailed overview of the BCE. There are two main blocks, namely the RISC/DSP processor (the D940HF component produced by Atmel) and the reconfigurable processor based on a high-end FPGA (Xilinx Virtex4-FX100 [5]), that carries also two PowerPC 405 processing cores. The DCM board (left on the figure) and the FPGA module (shown on the right) are logically part of the same BCE but they are implemented as two separated boards, both placed on a common main board. This configuration was chosen to cut development time, since the DCM board was already available; no significant limitation comes from this solution. This choice adds flexibility to the platform itself, as modules may be replaced by a different subsystem with a different set of functions (e.g., a video processor for multimedia applications).

Each of the two elements (DCM module and FPGA board) has a private and independent memory bank ("Mem 1" and "Mem 3" in the figure) boosting overall memory bandwidth, needed to sustain a large computational throughput. Also, two

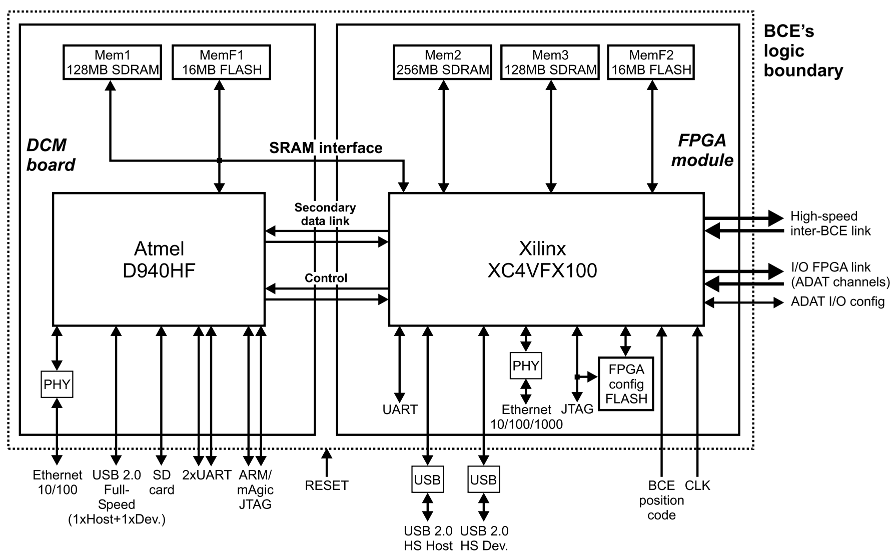


Fig. 3.4 Detailed block diagram of the Basic Configurable Element (BCE) of the hArtes Hardware Platform. The BCE is the basic building block of the platform, supporting several processing architectures. One or more BCEs work in parallel to support an hArtes application

separate FLASH memory banks are present, to keep user data and programs. The FPGA module has also a shareable memory bank ("Mem2" in the figure). This bank can be shared at different levels, i.e. among all the computing elements of the BCE itself (the D940HF component, the reconfigurable fabric of the FPGA and the PowerPC CPU inside the FPGA), and by the processors of any other BCE available on the system. This feature is supported by specific functional blocks configured inside the FPGA. Memory sharing (or any other program-controlled pattern of BCE to BCE communication) is handled by the links connecting the BCEs. They have large bandwidth (of the order of 200 Mbyte/sec), making data transfers of short data packets very efficient.

Finally, each BCE has several standard input/output interfaces (Ethernet, USB, UART), located both on the DCM board and on the FPGA module. The D940HF component on the DCM board contains two cores: one ARM processor and one mAgic DSP core (a floating-point core, optimized for DSP). The connection of both of these computational elements to the rest of the BCE goes through a common static memory interface. Although this solution is not optimal from a performance point of view, it guarantees sufficient bandwidth for the ARM/DSP cores to access the audio buffers located into the shared memory. A memory access arbiter within the FPGA controls memory requests coming from the DSP core, from the PowerPC cores and from the reconfigurable fabric (when it implements signal processing functions). A software lock mechanism has to be implemented to guarantee memory coherency, since neither the ARM/DSP cores nor the PowerPC core implement native cache coherency facilities.

The hHP main board contains the basic infrastructure to support the two BCEs, their interconnections and the I/O connectors. The key components on the main board are the two BCEs, in the shape of two FPGA modules and two DCM boards: to reduce signal lengths, the two BCE board groups are placed in a specular way. On the main board we have also all the appropriate power conditioning and distribution elements. The system clock is distributed to all daughter boards with low skew (except for the DCMs that have a private clock). This eases the data exchange on the links between the two BCEs (no needs for clock-domain resynchronization) and guarantees the synchronization of programs executing on the reconfigurable part of the BCEs. The main board has SD card connectors for the DCMs, Fast-Ethernet, USB Host/Device and serial ports. For audio applications, we have eight ADAT optical decoders and eight optical encoders. A medium-sized FPGA (Virtex4-LX40) converts the audio samples (from serial to parallel and vice-versa), forwards audio data inputs (coming from all ADAT decoders) to all the BCEs, using a simple bus, and collects audio data coming from all BCEs to be output onto the ADAT encoders. Simple control logic is available to arbitrate the use of the same output channel by different BCEs, in a first-come, first-served fashion. This FPGA contains also the logic needed to control the configuration of the ADAT encoders and decoders (in terms of sample frequency, sample bit width, etc.), and a number of FIFOs needed to stage the data streams coming from the decoders and going to the encoders. Figure 3.5 shows a picture of the present version of the platform.

3.3.4 Special Features for Audio Processing

One specific feature of the hHP is its strong support for high-level data streaming. This support includes dedicated I/O audio ports as well as firmware-enabled support that independently moves (and, if needed, re-format) audio data between memory buffers and I/O devices without processor intervention. This approach completely off-loads the CPUs on board from these tasks, so they can handle "useful" computational work.

At the basic hardware level, the platform has 8 input and 8 output ADAT interfaces. ADAT is a widely-used standard that uses optical-fibers and packs eight audio streams on each link, so our system supports up to 64 audio channels. All audio streams are collected by a dedicated FPGA that sends all data to either BCE. Any combination of ADAT input channels can be forwarded to any of the two BCEs, on the basis of appropriate configuration registers that can be set under software control. Conversely, any of the BCEs can be selected to supply data streams to any of the ADAT output channels.

Audio data, after reaching the target BCE, is processed by a dedicated engine, implemented in firmware inside the FPGA, that writes data to memory buffers. Each channel has two independent buffers (the structure uses a "ping-pong" approach); data from any incoming stream is automatically copied to the corresponding buffer; when a (ping or pong) buffer is ready, the engine raises an interrupt. In a completely

Fig. 3.5 Picture of the hArtes Hardware Platform. The two BCEs use two daughter boards each, one for the D940HF processor and one for the FPGA based infrastructure. These daughter-boards are at the center of the mother board. The ADAT interfaces and several standard I/O connectors are clearly visible at the *top* and at the *bottom* of the picture, respectively

specular fashion, the engine reads data from output buffers associated to any of the enabled output channels and moves them to the intermediate FPGA and eventually to the output ADAT links. All parameters of the structure (e.g., number of active channels, size and base addresses of the buffers) can be selected under software control.

Finally for each ADAT channel format conversion is available on-the-fly from integer to floating point (for input channels) and from floating point to integer for output channels (we remind that the ADAT standard uses 24-bit signed integers).

3.4 Firmware and Software for the hArtes Platform

As remarked in previous sections, critical features of the BCE architecture use a large recent-generation FPGA for their implementation. The system operates on a basic firmware infrastructure that provides several key functionalities of the system; this is the non-configurable partition of the system, that can be augmented at any time by partially configuring a subset of the FPGA to perform application-specific functions; the latter are prepared and compiled by the hArtes tool-chain, loaded onto the FPGA and then executed as part of the complete application. In the following

subsections we separately describe the "stable" partition of the system and the support for partial on-the-fly configuration.

3.4.1 The Basic Infrastructure

The FPGA-based section of the BCE performs several functions for the whole system. All these functions are supported by a stable firmware configuration of the FPGA itself, that is loaded onto the device at power-up. A non-exhaustive list includes the following:

- shared access to the common memory area (*M*em2) by the D940 processor and any further processor inside the FPGA;
- data transfer between the ADAT channels and the memory buffers;
- data re-formatting for the audio streams;
- message passing interface between the two BCEs;
- support for partial on-the-fly reconfiguration of the FPGA itself.

Some comments are in order:

- Probably the most important feature of the basic firmware infrastructure is the ability to route the memory read/write requests by several independent components to the appropriate memory blocks. This feature effectively builds up a central hub for all data transfer activities within the platform. Access requests are prioritized, and non-interruptible (real-time) accesses are given the priority they need.
- The control of all the audio streams is completely supported in hardware, so that none of the processor is busy for this purpose. In this way, a very large number of audio channels are supported by just one system.
- Some features have been added to boost application performances: an example is on-the-fly conversion of audio data from integer format to floating point format and vice-versa.
- Central to the operation of the platform it the support for dynamic partial reconfiguration, that is covered in detail in the next section.

3.4.2 Special Features for Software-Driven Configuration

The FPGA-based processor is reconfigurable "on-the-fly": the FPGA is divided in two logically separated domains, using partial reconfiguration features available on Xilinx FPGAs. One domain includes the PowerPC cores and all relatively stable functional blocks needed for basic system support, that is all functions described above that are required to run the processor, to access the private and shared memories, to move data through the interconnecting links and to handle all I/O activities. The second domain is reserved for reconfigurable functions. It changes its logic

structure as required by the running applications; using specific ports present on the PowerPC core it is possible, for instance, to reconfigure that part of the FPGA directly at run-time, augmenting the instruction set of PowerPC core, or implementing specific co-processor functions.

3.4.3 Software Support for the hArtes Platform

At present, the platform runs the Linux Operating System which is booted on the D940 processors; if both BCEs are in operation, each of them runs its own independent instance of the OS. The boot process of the system is planned to start from one of the SD cards connected to the DCM boards, or from one of the Ethernet ports. At any time one of the two BCEs is the master of the whole system, able to start slave processes on either BCEs. A limited set of specific software elements has been developed to support the applications running on the hardware. Key software elements are the following:

- mapping of the BCE shared memory (controlled by the FPGA) into the application addressing space: this feature enables the data exchange between the application, its audio data and the memory segment controlled by the reconfigurable part of the system;
- specific drivers to support the ADAT audio functionalities: since most of the functions are implemented in hardware, this module essentially configures the buffers that stores all the data and performs access to those audio samples;
- specific drivers to initialize and perform the dynamic reconfiguration of CCU modules;
- programming API to support inter-BCE data transfers: the communication model as seen at the application level is very similar to the one implemented by the MPI model, providing a familiar way to exchange data;
- automatic build of a single ELF file that contains all the executable elements making up a complete application, that are selectively loaded onto the ARM processor, the DSP core and the FPGA reconfigurable partition.

All in all, a very limited set of specific software modules was developed to ensure efficient operation of the platform.

3.5 Conclusion

This chapter presented the hardware platform that was built for demonstration and proof of concept purposes. It combined the Atmel Diopsis, the ARM processor with a Virtex 4 FPGA. The Molen machine organisation provided the architectural template that was implemented having the ARM as the master processor with the DSP and the FPGA as co-processing units, sharing the same linear address space. The

board was used for demonstration purposes and at the end of the project, the applications that are described in the following chapters were mapped on the hArtes board and successfully executed respecting the real time constraints.

References

1. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Moscu Panainte, E.: The MOLEN polymorphic processor. IEEE Trans. Comput. **53**(11), 1363–1375 (2004)
2. Piazza, F., Cecchi, S., Palestini, L., Lattanzi, A., Bettarelli, F., Capman, F., Thabuteau, S., Levy, C., Bonastre, J.-F., Toppi, R.: The hArtes Carlab: hardware implementation and algorithm development. In: AES 36th International Conference, 2009, Audio for Automotive, Dearborn, Detroit, MI, USA, 2–4 June 2009
3. Heinrich, G., Leitner, M., Jung, C., Logemann, F., Hahn, V.: A platform for audiovisual telepresence using model- and data-based wave-field synthesis. In: AES 125th Convention, San Francisco, 2–5 October 2008
4. Colacicco, I., Marchiori, G., Tripiccione, R.: The hardware application platform of the hArtes project. In: Field Programmable Logic and Applications. International Conference on FPL 2008, 8–10 Sept., pp. 439–442 (2008)

Chapter 4
Audio Array Processing for Telepresence

Gregor Heinrich, Fabian Logemann, Volker Hahn, Christoph Jung, Jose Gabriel de Figueiredo Coutinho, and Wayne Luk

This chapter presents embedded implementations of two audio array processing algorithms for a telepresence application as usage examples of the hArtes tool-chain and platform.

The first algorithm, multi-channel wide-band beamforming, may be used to record an acoustic field in a room with an array of microphones, the second one, wave-field synthesis, to render an acoustic field with an array of loudspeakers. While these algorithms have parallelisms and kernel functions typical for their algorithm class, they are chosen to be simple in structure, which makes it easier to follow implementation considerations.

Starting from an overview of the application and structure of the algorithms in question, several implementations on different levels of hardware abstraction are presented, along with empirical results on both the design process supported and the processing performance achieved.

4.1 Introduction

Telepresence—or audiovisual reproduction of a remote location—has long been investigated in research and real-world applications (see, e.g., [2–4]). While traditionally telepresence has been handled with the visual experience in focus and modest audio requirements, the importance of the auditory modality for transporting verbal information and spatial cues for an immersive perception of the remote environment is now generally recognized. Therefore, today in many telepresence systems, audio is now as central a modality as video.

With unobtrusiveness as a core design criterion (i.e., no headphones or other devices necessary on the part of the user), audio poses some technical challenges that are harder to cope with than its visual counterpart. Especially when telepresence

V. Hahn (✉)
Fraunhofer IGD, Fraunhoferstraße 5, 64283 Darmstadt, Germany
e-mail: volker.hahn@igd.fraunhofer.de

K. Bertels (ed.), *Hardware/Software Co-design for Heterogeneous Multi-core Platforms*,
DOI 10.1007/978-94-007-1406-9_4, © Springer Science+Business Media B.V. 2012

is used for speech communication, firstly any strong noise sources are to be reduced, a goal that needs to be balanced with the goal to create an impression of being present in the remote environment. Secondly, if communication is bi-directional, signal feedback between the locations needs to be suppressed. These issues add to the more general requirements for telepresence audio that distortions of the remote source signals should be minimal and that directions of remote sound sources in space should be mapped realistically in space when reproduced.

A viable approach to overcome these challenges is to use transducer arrays and associated processing methods, and this chapter will explore such methods as usage examples for the hArtes platform and tool-chain.

At the remote or acquisition end, a microphone array with a beamforming processor is used to filter acoustic sources in space, thus being able to suppress unwanted sound sources like from an open window or concurrent speakers. Furthermore, beamforming also allows to control the amount of reverberation. At the receiving end, sound may in principle be reproduced spatially using a stereophonic method that pans the location of the sound source between a set of loudspeakers. An alternative is, however, to render the wave field of the remote location, or, as we are spatially filtering the remote field using the beamformer, to simulate a model of the remote field using wave field synthesis that drives a loudspeaker array. Although wave-field synthesis is a more computationally intensive sound reproduction method than stereophonic approaches, it allows more stable image sources in the reproduction space. Furthermore, its parallelism structure as an array processing method is highly illustrative as an application of the hArtes platform.

In this chapter, we will explore the beamforming and wave-field synthesis algorithms to demonstrate the viability to build advanced and highly parallelized real-time immersive audio reproduction systems on the hArtes reference hardware using the hArtes tool-chain. In particular, we are interested in examining how hArtes reduces the development effort of high-end audio systems, in examining the flexibility and scalability of the hArtes platform for large numbers of parallel signals processed. For this goal, it is not necessary to add complexity by suppressing signal feedback, which requires echo cancellation algorithms. Such more complex approaches will, however, be presented in Chap. 5, and consequently, our scenario is one of unidirectional telepresence.

We will continue this chapter as follows. Section 4.2 will give a more detailed overview of the scenario that the algorithms are applied in, while Sect. 4.3 gives an overview of the algorithmic structures itself. In Sect. 4.4, we will describe how these algorithms are implemented on different target hardware, with and without the hArtes tool-chain. The results obtained from this implementation are presented and discussed in Sect. 4.5. We finish this chapter with general conclusions on the hArtes tool-chain and platform in Sect. 4.6.

4.2 Scenario and Architecture

The envisioned telepresence system is presented in Fig. 4.1. Here, the black slabs on the bottom of the video screens are the casings of the microphone and loudspeaker

Fig. 4.1 Prototype setup used for the telepresence application

arrays, respectively. There are two sides to a unidirectional audiovisual transmission. On the acquisition side, a microphone array records the utterances of speakers and other sound sources in the room selectively, while at the rendering side, a loudspeaker array reproduces the audio by projecting the recorded acoustic sources in spatial directions that re-enact the recording directions.

In order to identify the recording directions, a visual approach has been taken: By tracking faces with a video camera, speakers are recognized and followed, sending updates to the positions directly to the beamformer that runs the microphone array, similar to [1, 8]. This itself sends positional information to a wave-field synthesis system that runs the loudspeaker array.

A functional overview of the realized system can be seen in Fig. 4.2. The blocks on the bottom left and right show a Diopsis Evaluation Board (DEB) module for the beamformer and a hHP for the wave-field synthesis. The DEB contains a single Diopsis processor and peripherals for 16 I/O channels, which is sufficient for the processing in the beamformer. The beamformer may as well be executed on the hHP.

4.3 Array Processing Algorithms

Beamforming (BF) and wave-field synthesis (WFS) are array processing methods that strongly rely on the wave character of the sounds to be reproduced in space. While in the beamformer waves are superimposed in order to create directional gain patterns, i.e., to spatially filter directional waves, in wave-field synthesis the superposition is found in every point of the listening space where an acoustic wave is synthesized. In this section, we will review the algorithms with some attention to their common structure, which is of interest in the integration and implementations done with the hArtes tool-chain.

As a basis for both algorithms, consider the pressure field of a point source located at $\vec{x} = 0$, which is:

$$p(\vec{x}, t) = \frac{P}{r} \exp j(\vec{k}\vec{r} - \omega t) \tag{4.1}$$

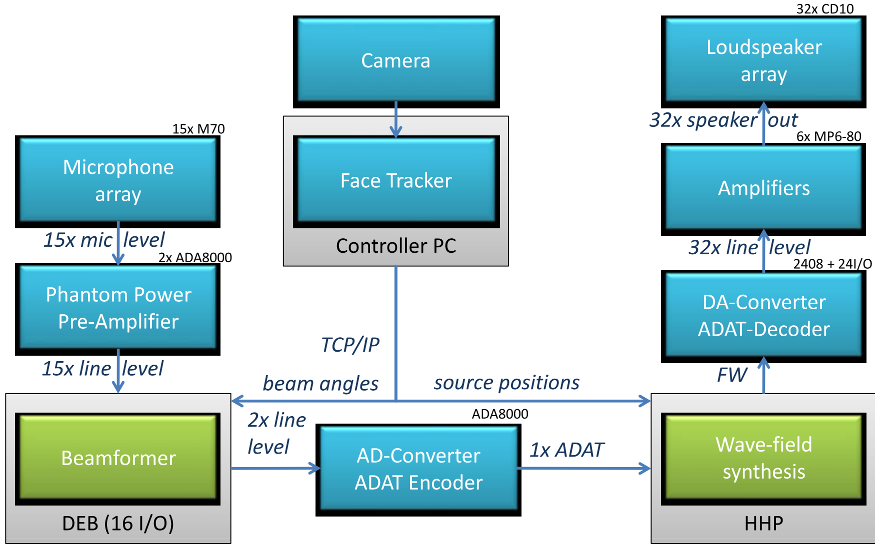


Fig. 4.2 Telepresence scenario: functional overview

where P is the pressure amplitude, $\vec{k}$ is the wave number with $k = |\vec{k}| = \omega/c$, the ratio between angular frequency ω and speed of sound c (or the wave cycles per meter), and $\vec{r} = \vec{x} - \vec{x}_0$ is the vector difference between source and observation locations $\vec{x}_0$ and $\vec{x}$, respectively. For the following review, we use the common assumptions of free-field conditions (no reflections) and omnidirectional transducers with infinitesimal size (microphones and loudspeakers are points in space) as well as a 2-dimensional geometry.

4.3.1 Multi-channel Wide-Band Beamforming

Beamforming [9, 11] is the directional filtering of spatially separated signals using sensor arrays with adaptive or fixed array processing algorithms. The output signal of such a sensor array can generally be described as a sum of the sensor signals filtered by some sensor-specific transfer function $h_n(t)$:

$$s(t) = \sum_n h_n(t) * p(\vec{x}_n, t) \tag{4.2}$$

where the operator $*$ denotes convolution and $\vec{x}_n$ the location of array element n. Figure 4.3 illustrates this for equidistant $\vec{x}_n$ on a line array. Standard beamforming approaches assume far-field conditions for the acoustic sources, and it is clear that a plane wave from a direction is amplified in the array output $s(t)$ by compensating the delay between the instants of arrival of the wave front at each sensor, which is called a delay-and-sum beamformer. To additionally suppress signals incident from other

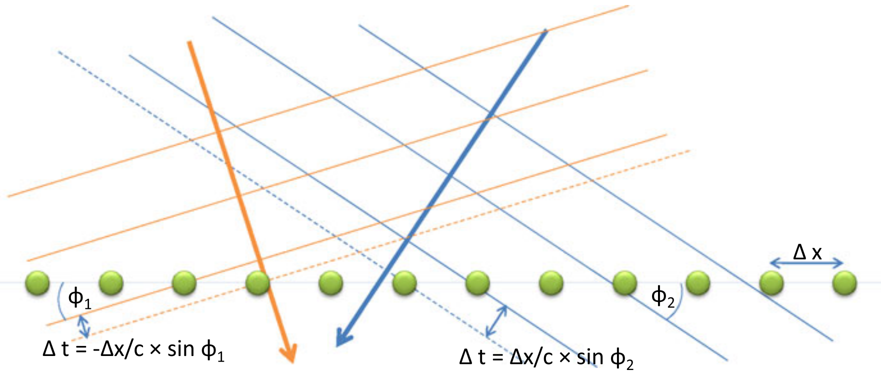


Fig. 4.3 Multi-channel beamforming: Listening in two directions

directions than the azimuth ϕ (side lobes), filter-and-sum approaches are preferred over simple delays, especially for wide-band signals like speech.

This leads to the general case of processing in the BF demonstrator application: A set of FIR filters with impulse responses $h_n(\phi, t)$ that convolve the microphone inputs $p(\vec{x}_n, t)$ and whose outputs are summed to obtain a beamformer output signal. In the scenario applied in hArtes, there are several concurrent beamformers, i.e., several impulse responses for different angles $\phi_{1,2,\ldots} : (h_n(\phi_1, t), h_n(\phi_2, t), \ldots)$.

The impulse responses $h_n(\phi, t)$ for the microphones are determined dependent on a given sensor configuration (locations $\{\vec{x}_n\}$ in space and possible source angles $\{\phi_i\}$). For our hArtes implementation, this is done in an offline manner in order to concentrate computational resources to the actual filtering: For each beam steering angle ϕ, the impulse responses $h_n(\phi, t)$ are derived as those sets of coefficients that zero the array output for any off-beam angles, $\phi + \alpha$ ($\alpha \neq 0$) over all frequencies (in practice discretized for K frequencies k and J angles α). Typically it is impossible to truly zero off-beam output, which leads to approximate solutions, most importantly least-squares approaches [9, 11] that estimate those filter coefficients $h_n(\phi, t)$ that minimize the squared error between the desired and true array response.

Least-squares beamformer designs have usually some properties that are undesired for a given application. Such issues are typically addressed by regularization approaches [6] and adding optimization constraints, such as bounding the white noise gain, enforcing unity gain at the main lobe azimuth (see, e.g., [5]) or weighting the importance of particular optimization criteria (frequencies, angles). Such regularizations have been appropriately applied in the offline algorithm.

4.3.2 Wave-Field Synthesis

Wave-field synthesis [7, 10] is the reproduction of a wave field in a listening space bounded by loudspeakers with spatio-temporal properties that are, in principle, identical to those in a recording space. In theory, there is no “sweet spot” restriction that

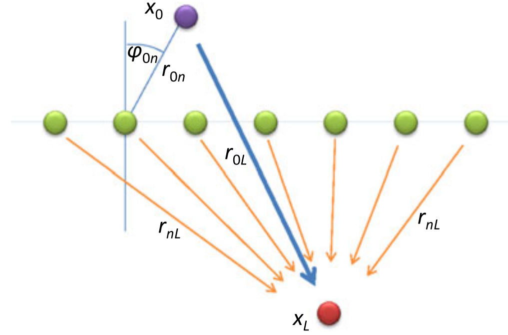


Fig. 4.4 Wave-field synthesis: Matching the original and synthesized fields

limits optimal spatial sound perception to a small area, as is the case for stereophonic approaches including surround sound.

Consider a wave from a point source at some position within the listening area $\vec{x} = \vec{x}_\ell$ as described by (4.1). The desired behaviour of WFS is to synthesize this field with a loudspeaker array with elements at positions $\vec{x}_n$, i.e., the pressure field of the point source and that of the synthesized field should be identical for each x_ℓ in the listening space and each frequency ω:

$$p(\vec{x}_\ell, t) = P/r_{0\ell} \exp j\omega(r_{0\ell}/c - t)$$
$$\stackrel{!}{=} \sum_n p_n(t) \exp j\omega(r_{n\ell}/c - t) \tag{4.3}$$

with $r_{0\ell} = |\vec{x}_\ell - \vec{x}_0|$ and $r_{n\ell} = |\vec{x}_\ell - \vec{x}_n|$. That is, at any listening position, the field of the original "primary" source should be the sum of the fields of the "secondary" sources (or, to speak in terms of Huygens' law, elementary sources). Figure 4.4 illustrates this. Synthesis of such a field using the secondary loudspeaker signals $p_n(t) = p(\vec{x}_n, t)$ is possible using a convolution of the original signal $p(\vec{x}_0, t)$ with some impulse responses $h_n(t)$, analogous to the beamformer:

$$p_n(t) = \sum_n h_n(t) * p(\vec{x}_0, t). \tag{4.4}$$

For the 2D case considered here, an approximate solution to determine the impulse responses $h_n(t)$ is by integrating a Rayleigh integral via stationary phase approximation [10], which leads to the WFS operator:

$$h_n(t) = h_0(t, \lambda) * a_n(\vec{x}_0)\, \delta(t - (\lambda r_{0n} - r_\mathrm{d})/c). \tag{4.5}$$

This equation consists of (1) a filter $h_0(t, \lambda)$ that depends on the source region $\lambda = \pm 1$ (source inside or outside the array), (2) a gain factor, $a_n(x_0) = f(r_{0n}^{-1/2}) \cos \phi_{0n}$, that depends on the distance between primary and secondary sources and the angle between their normals, and (3) a position-dependent delay term.

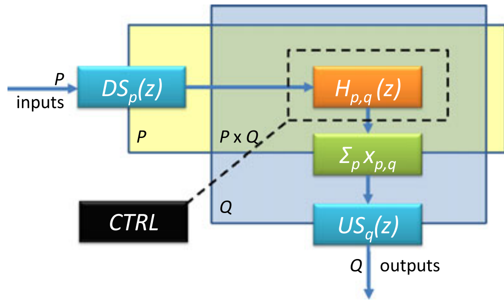


Fig. 4.5 Generic structure of the array processing algorithms

The term r_d/c is a modelling delay that ensures causal behaviour for "inner sources", i.e., sources projected into the space within the array. The possibility of such inner sources is one of the advantages of WFS because it can be used to let listeners perceive sounds directly in front of them or beside them. Technically, if $\lambda = -1$, most importantly the sign of the delay term is reversed, leading to a time-reversed wave that converges in one point in front of the array and subsequently travels as if originating from that point. For the converging part, the additional modelling delay is required.

4.3.3 Algorithm Structure

When considering several beamformers, i.e., a "multi-beamformer", and several WFS primary sources, the two array processing methods just described can be viewed as special cases of a generic processing structure with the following system equation:

$$y_q(t) = \sum_p h_{p,q}(t) * x_p(t) \tag{4.6}$$

where p corresponds to the index of a primary WFS source and a BF microphone, respectively, and q to a loudspeaker and a beamformer, respectively, with the transfer functions $h_{p,q}(t)$ derived as outlined above. Note that the number of primary sources and beamformers changes with the scene the systems are used for, whereas the numbers of transducers (microphones and loudspeakers) are constant for a given array design.

As a consequence of the (4.6), both algorithms chosen for the telepresence application scenario can be implemented using the same generic structure, as is shown in Fig. 4.5. Using notation in frequency domain, this structure can be understood as a grid of filters $H_{p,q}(z)$ whose outputs are accumulated, controlled by some parameter estimation block, *CTRL*. To simplify the visualization, the block scheme in Fig. 4.5 uses a specific notation for replicated blocks: Within the "flat" yellow and blue rectangles with denotations P and Q ("plates"), the blocks are duplicated for the respective number of times, if plates overlap, this multiplies the duplications.

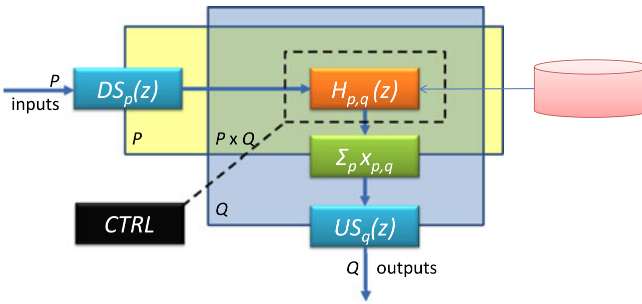


Fig. 4.6 Structure of the beamforming algorithm

For example, the filters $H_{pq}(z)$ are replicated $P \times Q$ times in the structure. Further, there are branching and joining blocks, in the structure above the summing operation over index p is a joining block that accumulates all filter outputs $H_{p,q}$ to one summed signal on the blue plate.

This generic structure incorporates the possibility to process filters at a different sampling frequency, i.e., in a subsampled domain, which is realized with the downsampling and upsampling blocks $DS(z)$, $US(z)$ and allows adjustment of the trade-off between audio quality and computational processing requirements in a more fine-grained manner than the options of the audio codecs permit. In general, it is of course advantageous to have the full audio bandwidth available, but on the other hand the available hardware does not always guarantee to process a number of channels sufficiently high to achieve the desired wave effects both for the beamformer and WFS.

The different algorithms for immersive audio processing require different specializations of this generic structure, starting at the number of typical input and output channels and the specific instantiation of the filter according to the structure of their transfer functions $H_{pq}(z)$.

Beamforming

Figure 4.6 shows the specialization of the generic algorithm structure to the beamforming case. Here a high number of inputs (microphones) and a low number of outputs (beamformers) are typical. The original filters are retained as processing blocks. Furthermore, the algorithm includes the possibility to downsample the signal for processing, as the beamforming design has a lower bandwidth due to the geometrical constraints of the microphone array while at the same time these constraints require a sufficiently high number of channels, which again requires $P \times Q$ channels to be filtered. Moreover, the filter coefficients are either calculated online using adaptive filtering algorithms or loaded from a database. In the final application, the second alternative has been chosen because the beam directions are available from an external source and the transfer functions can be calculated offline, which again increases the number of possible microphones or beams to process.

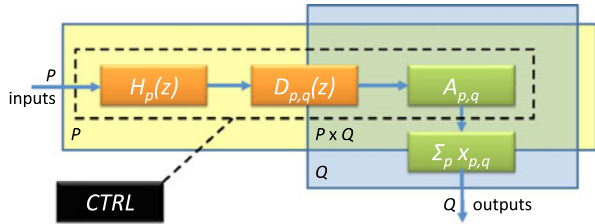


Fig. 4.7 Structure of the wave-field synthesis algorithm

From the outside, the application is controlled in terms of the directions of incidence that the speakers have relative to the microphone array. This information is sent to the algorithm via a network protocol, based on a face-tracking application that runs in the visual sub-system of the application (see Sect. 4.2).

Wave-Field Synthesis

Although roughly of the same algorithmic structure, in contrast to the beamformer the wave-field synthesis (WFS) setup typically uses a high number of outputs (speakers) and fewer input channels (primary sources to be rendered). Furthermore, according to (4.5) the structure of the transfer function between channels p and q can be simplified into an input-specific part with a full filter structure, a multi-tap delay structure that simulates the wave propagation though space and a gain stage between all input and output channels. The resulting algorithm structure can be seen in Fig. 4.7.

This processing structure allows simplification of the WFS approach compared to the beamformer because only P filters are required and the interaction between the $P \times Q$ channels is realized via delay lines. This again allows an increased number of channels for the algorithm compared to the beamformer with the same processing capabilities available on a given platform.

Analogous to the beamformer, the controller of the application receives information from an external source, including the positions of the virtual sources and loudspeakers in space. The virtual sources are constantly updated, changing the parameters of the filters, delay lines and gain factors. This information is also sent to the algorithm via a network protocol.

4.3.4 Processing Blocks

Regarding the processing required for both ends of telepresence system, their similar structure allows abstraction on the level of a small set of blocks. These blocks are described in the following. As a general assumption, processing is performed in audio frames.

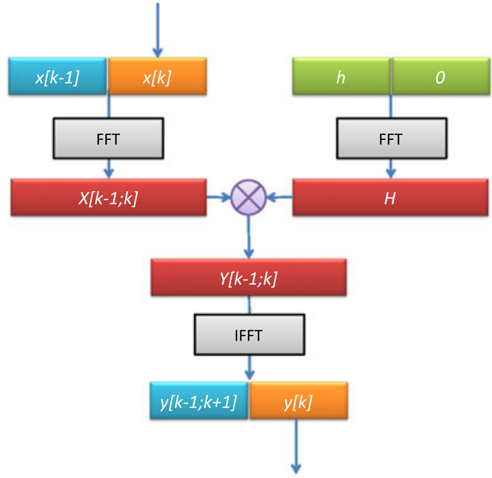
FIR Filter, $H(z)$

For impulse responses longer than about 50 samples, FIR filtering is computationally more efficient in the frequency domain, due to the beneficial scalability of the FFT. Consequently, this block consists of a frequency domain fast convolution based on the overlap-save (OLS) method. This method transforms the signal into frequency domain using the FFT, multiplies every complex-valued frequency bin of the input signal and the transfer function and transforms the result back to the time domain using the IFFT (a slight variation of the FFT). To cope with the intrinsic property of discrete Fourier transforms to result in circular convolutions (i.e., the tail of the convolved signal "wrapped around" and added to the beginning of the convolution result), the OLS method performs the transform on the concatenated previous ("saved") and current frame with twice the size of the FFT and discards the first half of the resulting output. This is illustrated in Fig. 4.8(a). In principle, this approach can be reduced in latency from one frame to one sample using methods known from multirate processing, but it turned out that in the hArtes applications, this is not required to implement the final scenarios.

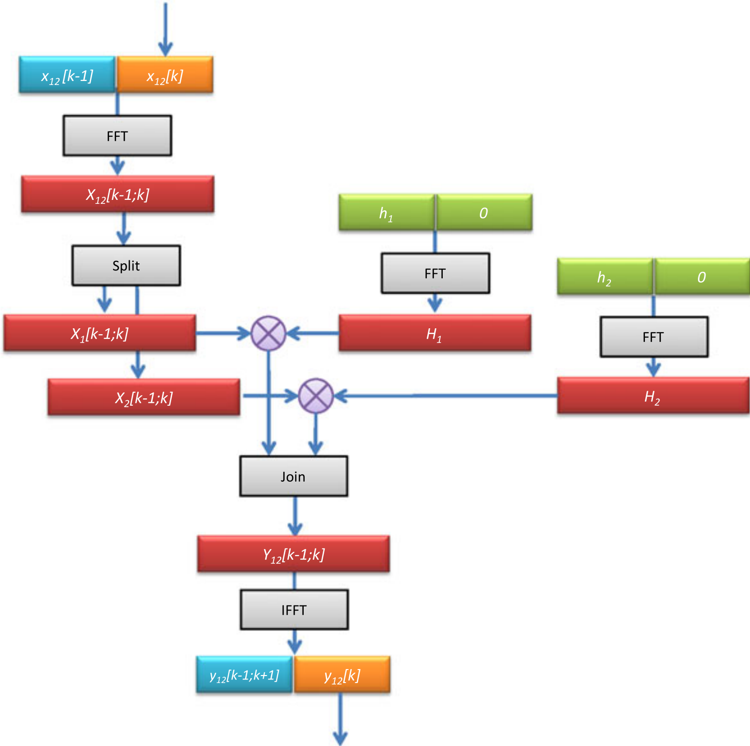
Usage of the FFT introduces the side-condition to process the audio in frame-sizes that correspond to the size of the available FFT implementations, with the most straightforward and sufficiently efficient approaches being radix-2 algorithms based on the Cooley-Tukey algorithm that require the FFT size to be a power of 2. Furthermore, because the FFT is a complex operation and real signals and impulse responses have important symmetry and periodicity properties, it is possible to perform an FFT of size N (number of complex samples) on an interleaved pair of signals of with N real samples, while the "standard" complex FFT requires interleaving of a single real signal with zeros. All that is required to achieve this dual OLS algorithm is a set of re-ordering and summing steps that split and join the complex signals in frequency domain, see Fig. 4.8(b).

Delay Line, $D(z)$

Delay lines store samples in memory to postpone their output. From an implementation perspective, delay lines are implemented efficiently using ring buffers where the write pointers are positioned according to the incoming frames and the read pointers according write positions shifted by the delay time set. In the WFS application, several outputs are based on the same input, therefore the actual algorithm used is that of a multi-tap delay line or ring buffer with several read pointers, each dependent on the output channel, as shown in Fig. 4.9. When using the delay lines with large differences in delay times as can be the case with WFS, the delays are gradually updated, leading to a smooth, Doppler-like acoustic effect. To achieve efficient memory operations, samples are copied from the read pointer in a block that may be split by buffer boundaries (considering $y_1[k]$ in Fig. 4.9, the first partial block of output is at end of the buffer, the second one at the beginning).



(a) Single channel filter.



(b) Dual-channel filter.

Fig. 4.8 Overlap-Save fast convolution for a filter block $H(z)$. *Square brackets* indicate frame indices, interleaved signals are denoted x_{12} etc., sizes of data blocks are to scale

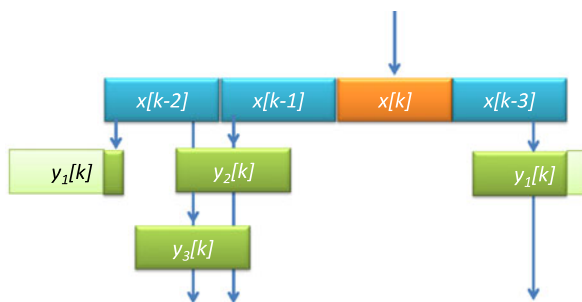


Fig. 4.9 Multitap ring buffer with wrap-around output for a delay line $D(z) = z^{-M}$

Multiply-Accumulate, A, Σ

All algorithms need to accumulate the processing result from several input channels, possibly in a weighted manner. These operations are of special interest less because of their computational complexity (which is trivial) but rather because a large volume of data is collected. That is, the multiply-accumulate block is mainly a memory bottleneck.

Subsampling, $DS(z)$, $US(z)$

Subsampling reduces the sampling rate of the audio to a fraction of the original sampling rate, allowing to process the audio based on smaller data rates and frame sizes, at the expense of audio bandwidth. In principle, multirate resampling schemes can be used that allow a wide variety of subsampling ratios. However, for the final applications, subsampling is based on integer ratios, allowing to implement the blocks using a lowpass filter to remove spectral content from the audio that would disobey the sampling theorem at the lower sampling rate and a resampling block that picks every Rth sample (with R the subsampling ratio). On the upsampling side, each subsampled signal value is repeated R times and the result lowpass-filtered to remove the high-frequency sample-and-hold artefacts.

The computationally most expensive part of these blocks consists of the lowpass filters. However, opposed to the actual processing filters, IIR structures can be easier applied, allowing more efficient operation than FIR structures, at the expense of slight phase distortions. To design the lowpasses, classical digital filter design methods are applied, in particular, the Chebysheff type I method with optimal stopband suppression at a sufficiently good passband ripple, or type II with reversed properties.

Controller, *CTRL*

The control input to both algorithms requires positional information of the sources to be recorded or rendered. In the beamformer, the angles and source IDs need to

```
while (sys_running) {
    // (0) update parameters, CTRL
    getParameters(...);

    // for each input channel pair
    for (p = 0; p < P; p += 2) {

        // (1) acquire audio channels pairwise
        getAudioPairBuffer(p + 1, inputbuffers, INTERLEAVED);

        // (2) [downsampling, DS(z)]
        downsample(inputBuffers, inputBuffers, outerBlockSize, innerBlockSize);

        // (3) [input-specific filter, H_p(z)]
        dualFilter(inputBuffers, tempBuffers, inParamPair[p], 2 * innerBlockSize);

        // for each output channel
        for (q = 0; q < Q; q++) {

            // (4) load filter parameters for current p,q set, part of CTRL
            getFilterPairParameters(p, p + 1, q, inoutParamPair);

            // (5) either filter, H_pq(z), or delay line, D_pq(z), A_pq
            {dualFilter|delay}(tempBuffers, tempBuffers2,
                               inoutParamPair, 2 * innerBlockSize);

            // (6) sum up the processed signals, sum_p x_pq
            sumInterleaved(outputBuffers[q], tempBuffers2, 2 * innerBlockSize);
        } // q ++
    } // p += 2

    // for each output channel
    for (q = 0; q < Q/2; q++) {
        // (7) [upsampling, US(z)]
        upsample(outputBuffers[q], outputBuffers[q], innerBlocksize, outerBlockSize);

        // (8) for pairs of channel buffers (don't need to deinterleave)
        putAudioBuffer(2 * q + 1, outputBuffers[q], NON_INTERLEAVED);
    } // q
} // sys_running
```

Fig. 4.10 Main loop C-language structure of the generic algorithm in Fig. 4.5

be sent to the algorithm, and in WFS, the positions and source IDs. In addition, the controllers are application-specific and cannot be fully generalized: While for the beamformer, the controller includes management of the filter database with a relatively simple lookup algorithm, the WFS application requires more sophisticated geometrical calculations to derive the delays and amplification gains for the different loudspeakers. This difference is important from the point of view of heterogeneous compute resources because while for the beamformer, the database handling may expected adequate for the GPP (loading new filter coefficients into the processing device), computation for the WFS algorithm (including geometric floating-point operations) is a task for a specialized processing device like the DSP.

4.3.5 Code Structure

Based on the block scheme in Fig. 4.5, it is possible to give a simplified algorithm structure of the main processing loop. This is presented in Fig. 4.10 and shows that the generic algorithm blocks are preserved in the C-language implementation: The main loop over audio frames starts with (0) retrieving parameters for processing the

current frame, that is, running the logic of the *CTRL* block. For each input channel pair, a loop is started that (1) retrieves a single frame and optionally (2) downsamples this pair. The input frames are retrieved in interleaved sample order to support the pair-wise operation of filters. In a subsequent optional step, the signal is (3) filtered with an input-specific set of coefficients, which effectively splits the filter $H_{p,q}(z)$ in Fig. 4.5 and is used if this saves overall computations (as in the WFS algorithm, cf. Fig. 4.7).

To complete the inner filter block $H_{p,q}(z)$ (the double plate in Fig. 4.5), for each output channel q, (4) the filter coefficients for the two relevant filters (p,q) and $(p+1,q)$ are retrieved and (5) the filters $H_{p,q}(z)$ and $H_{p+1,q}(z)$ applied to the pair of input frames. The filter excludes the input-specific part applied in step (3) and may in addition be simplified to a delay line (for WFS). The output inner loop is completed by (6) summing the filter result to the particular output.

The generic algorithm finishes with a loop over pairs of output filters that for each pair performs (7) optional upsampling to compensate the input downsampling step (2) and (8) writes back the processed audio frames to the output buffers. Output frames are non-interleaved.

4.4 Integration Approach

To implement the telepresence application, we adopted a strategy that allows having functional results as early as possible while decoupling them from the actual tool-chain development that was carried on concurrently. There are three different types of implementation: First we developed a set of implementations of the algorithms that run on general-purpose processor (GPP) architectures. Second, we implemented the algorithms on the Diopsis architecture, applying manual optimizations using joint GPP and DSP cores. And finally, we used the hArtes tool-chain to implement the algorithms, varying different degrees of automated and constrained mapping and partitioning as well as processing elements.

In terms of computational efficiency, the GPP implementations may be considered a base-line while the manually optimized implementation on Diopsis may be considered a "gold-standard", and the implementations using the hArtes tool-chain and platform are expected to have computational efficiency somewhere between them. Furthermore, GPP-based implementations may be considered a "gold standard" for development, as their realization is straight-forward and does not require special knowledge on embedded hardware APIs.

A general overview of all implementations of the audio array processing algorithms is shown in Fig. 4.11.

Opposed to the In-Car scenario described in the next chapter where Graphical Algorithm Exploration marks a red thread through algorithm development and integration, the focus in the telepresence application was C-language code exclusively, with Matlab sources used for initial proofs of concept. This provides an alternative way to implement algorithms, which is more aligned with "classical" software development in a low-level language.

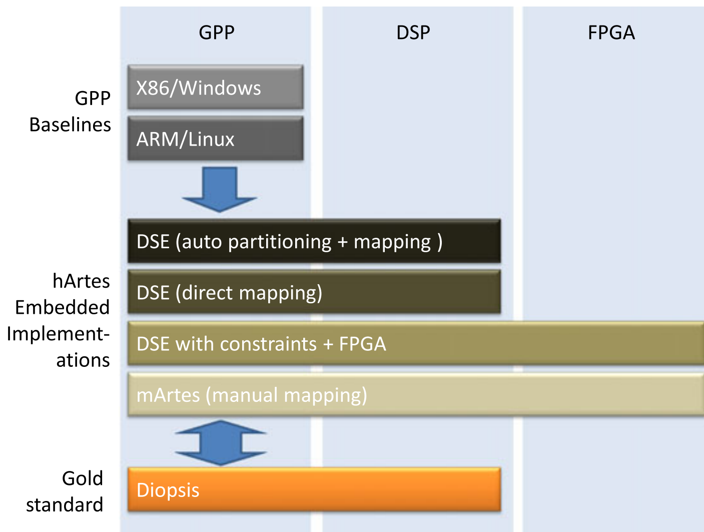


Fig. 4.11 Implementations of audio array processing algorithms

In the following sections, we will describe these implementations, starting with the reference implementations (base-line and gold standard) and subsequently the ones based on the hArtes tool-chain.

4.4.1 Base-Line and Gold-Standard Implementations

Besides the first demonstrator on a PC platform, ARM-only and Diopsis implementations of the BF and WFS algorithms were realized. This way, numerical results may be tested regardless of computational efficiency, and subsequently the hardware-specific DSP implementation ported from this code.[1]

In particular, in Table 4.1 the list of reference implementations is given, along with the numbers of input and output channels specified for the particular realizations of the algorithms. These figures are chosen either to the maximum of available processing resources (realtime operation) or to allow a reasonable comparison with hArtes implementations (offline operation). The main constraints for such modifications are memory limitations and the number of accesses to available hardware channels.

[1]During the project, in fact the opposite direction was adopted in order to obtain a technical understanding of the DSP architecture and software platform in order to understand the requirements for the hArtes tools.

Table 4.1 Reference implementations of the BF and WFS algorithms

Platform/operation mode	Beamforming	Wave-field synthesis
X86 GPP/realtime	$P = 24$, $Q > 8$, x86	$P > 32$, $Q = 72$, x86
X86 GPP/offline	$P = 16$, $Q = 2$, ARM + x86	$P = 2$, $Q = 8$, ARM + x86
ARM GPP/offline	$P = 16$, $Q = 2$, DEB (hHP)	$P = 2$, $Q = 8$, hHP
ARM + DSP/realtime	$P = 16$, $Q = 2$, DEB	$P = 2$, $Q = 32$, hHP

GPP Implementations

As can be seen from Table 4.1, the x86-based implementations (on Windows PCs) have rather high numbers of channels that can be processed. This shows the advantage of multi-core Pentium x86 machines running at 30 or more times the clock rate of the embedded systems (3 GHz vs. 80 or 100 MHz for the DSP), which cannot be reasonably compared in terms of actual performance. However, one needs to keep in mind the typical power ratings of both hardware platforms, with dissipations of well above 50 W for x86-based GPP versions versus less than one to a few watts for the embedded implementations.

Regarding the ARM GPP implementations, the parameters have been set to be comparable to hArtes implementations rather than to meet real-time constraints. Furthermore, with the ARM implementations, all floating-point operations have to be performed using the softfloat support in ARM (`--target arm-softfloat-elf`). In addition to an ARM implementation, it is possible to re-use code on almost any computing platform supported by GCC, which has been tested for PC-based Linux and Mac OS X. One advantage of this alternative is that the more comfortable Eclipse debugging environment can be used for testing.

All GPP implementations have been realized in a straight-forward manner, implementing signal processing blocks directly using standard C data types (`float`) and avoiding the use of specific library functions. In order to keep the structure of the algorithms comparable between all versions, the implementations of major processing blocks have been done interface-compatible with functions available from the Diopsis DSPLib, a performance library available for both the Diopsis and hArtes implementations, which in the scope of the array processing scenario mainly provides pair-wise FIR and IIR filters.

This interface-compatible approach not only allows to re-enact DSP implementations on GPP, it also permits complete control on the hardware and implementations to be used for evaluation from within the hArtes-based code. This control is exerted by way of pre-processor switches (`#ifdef`) and extends to GPP and all hardware targeted by the hArtes tool-chain.

Diopsis Manual Implementations and Benchmarks

Opposed to the GPP implementations, for the Diopsis platform specific domain knowledge on the algorithm and hardware had to be applied. This leads to two

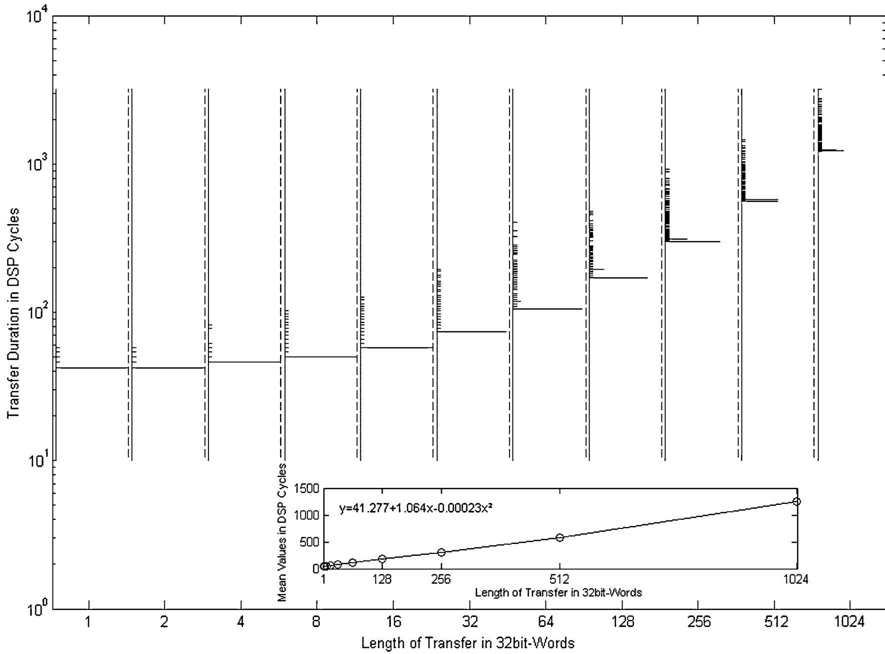


Fig. 4.12 Exemplary benchmark of memory transfer (hHP DMA read from shared memory to DSP-internal memory)

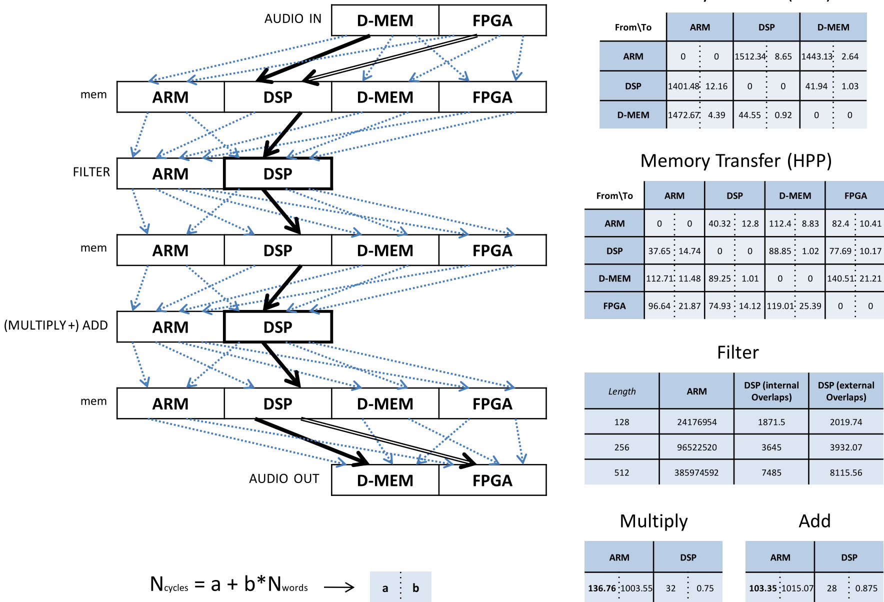
gold-standard implementations for BF and WFS that have been used to check the maximum performance possible for the given algorithms.

Regarding direct implementations of the algorithms on hHP, a part of the hArtes runtime framework already had to be used in order to cope with the custom ADAT interfaces provided on hHP, however, without any tool-chain functionality.

To obtain platform knowledge for the actual implementation, especially with respect to the high data bandwidths required for processing large numbers of audio channels, a number of benchmarking experiments have been performed. As an example for the statistical behaviour of an inter-chip memory transfer, we have chosen a DMA read operation from shared memory to DSP-internal memory. Figure 4.12 depicts the behavior of this transfer, depending on the length of the transferred array in 32 bit-words. From these measurements, we can concluded that the dependency of the duration on the array length can very closely be approximated by a linear equation, verifying theoretical results.[2] Assuming a frame size of N words per audio channel leads to:

$$\#\text{cycles} = a + Nb \times \#\text{channels}. \tag{4.7}$$

[2]In the measurements in Fig. 4.12, outliers were attributed to scheduling.



Memory Transfer (DEB)

From\To	ARM (a)	ARM (b)	DSP (a)	DSP (b)	D-MEM (a)	D-MEM (b)
ARM	0	0	1512.34	8.65	1443.13	2.64
DSP	1401.48	12.16	0	0	41.94	1.03
D-MEM	1472.67	4.39	44.55	0.92	0	0

Memory Transfer (HPP)

From\To	ARM (a)	ARM (b)	DSP (a)	DSP (b)	D-MEM (a)	D-MEM (b)	FPGA (a)	FPGA (b)
ARM	0	0	40.32	12.8	112.4	8.83	82.4	10.41
DSP	37.65	14.74	0	0	88.85	1.02	77.69	10.17
D-MEM	112.71	11.48	89.25	1.01	0	0	140.51	21.21
FPGA	96.64	21.87	74.93	14.12	119.01	25.39	0	0

Filter

Length	ARM	DSP (internal Overlaps)	DSP (external Overlaps)
128	24176954	1871.5	2019.74
256	96522520	3645	3932.07
512	385974592	7485	8115.56

Multiply

ARM (a)	ARM (b)	DSP (a)	DSP (b)
136.76	1003.55	32	0.75

Add

ARM (a)	ARM (b)	DSP (a)	DSP (b)
103.35	1015.07	28	0.875

Fig. 4.13 Embedded platform benchmark based on the generic array processing algorithm in Fig. 4.5. *Left*: optimal path (D-MEM represents shared memory, FPGA used for hHP audio I/O), *top-right*: memory transfers on DEB and hHP (linear coefficients: DSP cycles at 80 MHz $= a + Nb$). *Bottom-right*: Performance figures for filter operation (cycles vs. frame-length, internal/external overlap refers to storing the previous OLS frame in DSP internal memory or shared memory) and multiply-add (cycles $= a + Nb$)

A broader view of data flow measurements on both the Diopsis board and the hHP is presented in Fig. 4.13, which re-enacts the different data paths and processing elements that the generic array processing algorithm in Fig. 4.5 needs to run through. For memory transfers and multiply/add-operations the above linear approximation has turned out to be valid, thus the corresponding tables contain both the offset value, a, and the slope, b, which is more decisive for bulk data flow. For filters, we assume their length equal to the input signal frame size, leading to a constant input array size of N words.

According to the values from Fig. 4.13 right, the optimal path through the array processing algorithms is to keep data on the DSP, as shown in Fig. 4.13 left. Also, the best mapping in terms of processing clearly is the DSP core due to its superior computational efficiency even at trivial computational tasks (factor > 10) while running at half the ARM's clock rate.

The GPP on the other hand may be fully occupied with any kind of administrative and controlling tasks while the real-time loop is running, like synchronizing with the peripheral ADAT I/O system and Ethernet interface while performing complementary data transfers to/from the DSP. The main effort in partitioning the algorithms went into keeping the AHB bus to/from the DSP as occupied as possible

Table 4.2 Parameters of hArtes implementations of the BF and WFS algorithms

Parameter	Beamforming	Wave-field synthesis
Platform	DEB	hHP
Inputs P	16	2
Outputs Q	2	32
DSP clock rate [MHz]	100	100
Hardware sample rate [kHz]	16	48
Downsampling ratio	1	3
Frame length (downsampled)	256	512

while keeping wait states at a minimum (using non-blocking DMA transfers) and arranging the actual computational blocks in a way to cope with the limited amount of chip-internal highspeed-access memory, which only is 16 kwords on the Diopsis. The FPGA in our case is only used for controlling the audio peripherals of the WFS system running on one Diopsis on the hHP and not involved in processing without using the hArtes tool-chain.

Both algorithms could in principle be developed in a similar fashion. One way to limit the local amount of memory was to do multiplexing by channel-pair, i.e., to store incoming audio data for pairs of input channels in temporary internal buffers and process them before going on with the next pair, storing the other channel data in external memory (cf. Fig. 4.10). The crucial parameter in terms of memory consumption then remains the audio frame length.

4.4.2 hArtes Implementations

To test one of the core contributions of the hArtes project, our next goal was an investigation whether real-world algorithms like the ones described for the telepresence scenario can be efficiently developed using the hArtes tool-chain. For this, we have re-implemented the array processing algorithms starting from the ARM code described above, which resulted in implementations that are outlined in Table 4.2 and reflect high numbers of input/ouput channels to be expected on the given hardware. In the following, we will refer to these parameterizations as the hArtes WFS and BF algorithms. To realize them, the design methods from Fig. 4.11 have been used, in particular:

- *DSE (automatic)*: Design-space exploration with automatic partitioning and decision mapping.
- *DSE (DSPLib)*: DSE linking DSPLib implementations for functions that are mapped automatically to DSP.
- *DSE with constraints + FPGA*: DSE with DSPLib implementations, targeting the FIR filter kernel to FPGA.

- *mArtes*: hand-optimizing hArtes code using mapping pragmas and fine-tuned memory handling, e.g., manually calling `hmalloc_f()`, `hmemcpy()` etc.

A development strategy considered efficient is to initially omit any constraints (and thus required knowledge) on the code. This way, the tool-chain may determine an optimum set of mappings and sub-algorithms autonomously, for which subsequently manual improvements may be introduced at neuralgic points, switching tool-chain usage from purely automatic to semi-automatic. Because the tool-chain was under development while collaboratively realising hArtes algorithm implementations, we had to introduce manual decisions at an early stage, though, not fully following the ideal strategy.

Design-Space Exploration

Omission of any special developer knowledge on mapping or partitioning poses a difficulty to data-intensive multi-channel algorithms like BF and WFS and therefore constitutes a critical test to the tool-chain. Applying DSE directly in practice required some refactorings and modifications in the code compared to the pure GPP version of the algorithms, in order to optimize the stitch points of the algorithm, code locations where the algorithm may be mapped to different processing elements (PEs) and thus may incorporate data transfers.

One of the main issues was to handle of DMA calls, which are implicitly invoked at stitch points of the code. We do not consider this a real constraint for the tool-chain because the knowledge required for introducing for instance parameter qualifiers is purely on an algorithmic level. Such qualifiers are `const` and `volatile`, specifying that the output or input is not copied from or to a function when invoking it, which is crucial especially if large arrays are referenced in function parameters.

There are two additional cases where memory handling needed to be adjusted to allow the hArmonic tool (see Sect. 2.7.6) to generate efficient mappings. The first case is a temporary variable as a function argument that is neither read nor written in the caller but would be handled like a full array copy by the mapper if the call is determined as a stitch point. For instance, FFT implementations using the DSPLib API require an additional temporary array:

```
// function with read-only input and write-only output; temp array not to be copied
void fun(const float* in, volatile float* out, float* temp) {
    ...
}
```

A solution was to create a wrapper function with an array on the stack:

```
// function wrapper to map temporary variable locally
void funwrap(const float* in, volatile float* out) {
    float temp[tempsize];
    fun(in, out, temp);
}
```

As a consequence, the mapper creates the temporary variable in the appropriate memory region; in the case of the FFT, the DSP internal memory.

The second case is a reference to a large array that would be implicitly copied to the function's PE, which was the case with the impulse response database for the beamformer, a linearly indexed array of filter coefficients:

```
// number of angles, P/2 filter pairs, 2 frame lengths of complex coefficients each
float coefficients[numAngles * P * innerBlockSize * 2];
```

Dividing this into an array of angle- and microphone-specific pointers lead the mapper to select only the subarray actually required:

```
// number of angles, P/2 filter pairs, 2 frame lengths of complex coefficients each
float coefficients[numAngles][P / 2][innerBlockSize * 4];
```

Design-Space Exploration with DSPLib Constraints

For the fully automated approach described in the previous paragraph, it was necessary to implement all kernels manually to allow the decision mapping the full choice of processing devices. These kernels are the blocks described in Sect. 4.3.4. However, they have not been manually optimized for any particular processing device and are therefore sub-optimal by design. To overcome this, functions from the DSPLib performance library have been used, manually optimized implementations of standard DSP kernels including filters, vector operations, etc. In the applications, these DSPLib functions have been linked in place of the hand-implemented versions of the automatically mapped version wherever the mapping tool targeted them to the DSP. It turned out that this was exactly the constellation necessary to achieve best performance. Besides introducing this by simply including the DSPLib headers into the files in question and the DSPLib implementation library into the library path, the DSE build process remained fully automatic.

Design-Space Exploration with FPGA Constraints

Because of the inferior communication profile of the FPGA (also cf. benchmarks in Fig. 4.13), the hArmonic tool did not include the reconfigurable FPGA as part of a fully automatic solution. To test the algorithms with support for custom-configured units (CCUs) running on FPGA (VIRTEX4), additional manual mapping constraints were introduced to hArmonic to achieve a semi-automatic solution. For this, the computationally most expensive processing kernels, the FIR and IIR filters, were the primary candidates for FPGA and have been annotated using a `map VIRTEX4` pragma in their header files.

In fact, the same implementations of the algorithm blocks developed for the GPP version of the algorithms have been used as a basis for the FPGA mapping, but in order to use the C2VHDL generator DWARV, the mapped code additionally needs to obey restrictions, which includes inlining of function calls and strict ANSI variable declarations, as described in Sect. 2.8.3. These restrictions required some modifications of the FIR and IIR function implementations.

Moreover, as memory transfer to and from FPGA is slow, incorporation of processing in CCUs has some more implications. In fact, mapping the filters to FPGA may be much more efficient if the audio input channels can be directly from the FPGA, for the beamformer using the IIR filters in the downsampler and for WFS using the FIR filter as the first non-trivial processing step. This leads to a special, manual implementation of the ADAT buffer getter function.

mArtes

Finally, the hArtes framework has been used to implement the algorithm using developer expert knowledge about optimal mapping. This basically re-enacts what has been found in the manual Diopsis implementation but now uses several hArtes-specific flavours to important memory and peripheral APIs. Employing this additional layer of control over for instance DMA calls (hArtes framework functions `hdma` and `hdma_f`) may reduce inefficiencies due to function calls across stitch points. In our specific implementations, all functions have been annotated with pragmas using the respective mapping target.

4.5 Results and Discussion

This section outlines how the tool-chain actually acted in the development process. We will first report on tool-chain usage, discuss its optimization capabilities and finally give some performance figures achieved on the hArtes implementations.

4.5.1 Tool-Chain Usage

Applying the tool-chain in the different cases described above turned out to be straight-forward. There are two possibilities to work with the tool-chain. On one hand, using the project in the Eclipse environment including the interactive mapping code guidelines analysis UIs of the hArmonic mapper proved to be a effective solution to speed up error search and optimization. On the other hand, having more control on the build process is always possible by writing a dedicated `Makefile` (which is necessary for the mArtes case, as well), or by copying and modifying the file generated by the hArtes Eclipse plugin. Existing limitations that may hinder efficient development have been found to be the constraints of DWARV input, which in most cases requires some code modification, as well as the limitations of control over DMA accesses. For the audio array algorithms, memory has turned out as one of the major issues, and although not a focus in hArtes, in a future project solutions to the transport problems posed by limited DMA bandwidths and internal caches is considered worthwhile.

The general result is that the tool-chain was very well usable for the algorithms in question. Although a formal comparison of development times could not be made, it

is easily re-enactable that using the tool-chain on GPP code from an Eclipse project significantly reduces the effort to write DMA-enabled code for the DSP/ARM co-processing architecture. Not only is the necessity dropped for the developer to acquire a deep knowledge of the heterogeneous architecture. It is also unnecessary to directly use the DMA API, but rather implement this using implicit DMA via standard C function calls (with `const` and `volatile` modifiers to fine-tune data access) and `memcpy()` calls, which are then being mapped appropriately.

Regarding the semantics of function calls themselves, the usage of `const` and `volatile` qualifiers goes a long way to trim memory transfers to viable solutions. However, stride and memory ranges are important use cases to be covered by such a system and should be added. As explained above, in the example applications, the work-around modifications for these issues are simple. Furthermore, handling limitations of DSP internal memory is not directly supported by the tool-chain but rather requires some extent of platform knowledge on the part of the developer.

When it comes to FPGA mapping, things become more complicated as currently there are special steps required to prepare C-language code for mapping to reconfigurable hardware. The same is true for handling memory efficiently under the constraints of FPGA processing: As FPGA memory is slow, the best solutions memory-wise do in fact bypass the hArtes sound interface and rather read directly from the ADAT interface. This approach has something of the manual DSP approach in that developers need to get low-level access to the ADAT buffers.

4.5.2 Tool-Chain Optimization Capabilities

To evaluate how the tool-chain optimized the algorithm implementations, we mainly looked at the mapping capabilities. As general result, the tool-chain proved to be able to reach similar results to manual mapping if some modifications to the GPP implementation are made. These modifications seem to be straight-forward and it might even be possible to introduce them automatically. In most cases where the mapping between ARM and DSP was involved, the code automatically mapped between the two PEs was valid at runtime.

Regarding mapping to FPGA, the limitations seem larger, as the C to VHDL translator has some restrictions with regard to the permitted C code. It needs to be added here that C to VHDL synthesis is a much more complex task that standard C to C translation. The restrictions are, however, not a major problem since they could be easily complied with and the tool-chain was even able to hint at the specific coding problems (hArmonic with `-gxc` option). Further, as communication from and to the FPGA turns out to be slow, non-blocking memory transfers are a feature especially important in multichannel applications like the ones at hand, and the developer should optionally have control of them to finer degree to avoid filling up frame time with exclusive communication time.

In addition to these qualitative results, we analysed the automatic mapping achieved with the hArmonic tool (see Sect. 2.7.6). Table 4.3 summarizes the main results. The total number of tasks measures the number of calls in the application

Table 4.3 Evaluation of the automatic mapping approach on the beamformer and wave-field synthesis applications. The hArtes constraints correspond to the number of automatically derived constraints to make the mapping solution compatible with the hArtes platform. The speedup achieved corresponds to the ratio between the performance of the application running on the ARM and the mapping solution produced by the hArmonic tool

hArtes application	Total number tasks [mapped to DSP]	hArtes constraints	Time to compute solution (sec)	Speed-up
Beamformer	168 [38]	677	6	7.9
Wave-field synthesis	145 [24]	455	5	9.2

that can be executed in any processing element, provided that such mapping does not violate any hArtes constraint rules. The number of tasks mapped to the DSP measures the fraction of the application that is executed outside the main processing element (ARM). The speedup corresponds to the ratio between the execution time of the application running on the ARM and the performance of the same application using the mapping solution derived by the hArmonic tool.

The beamformer application has 168 tasks but only 38 tasks are automatically mapped to the DSP (Fig. 4.14(a)); the remaining tasks stay in the main processor (ARM). This mapping solution results in a speed-up of 7.9 times over an ARM-only solution, with hArmonic taking 6 seconds to complete the automatic mapping process. On the other hand, the wave-field synthesis application has 145 tasks, with 24 tasks selected to run on the DSP, achieving a 9.2 times speedup (Fig. 4.14(b)). The magnitude of these speed-ups has been influenced by various factors: the use of optimized libraries such as DSPLib, manually managing the memory footprint to improve performance, the quality of the synthesis toolbox (back-end compilers such as `hgcc` and `targetcc`), and the performance of the hArtes hardware platform. More details on the speed-up of different hArtes implementations are given in the following section.

While the overall goal of the hArmonic tool is to find a mapping solution that maximizes performance by mapping part of the application to available processing elements, not all mapping solutions are feasible. In fact, there are a number of restriction rules that forbid several mapping configurations on the hArtes platform. For this reason, the hArmonic tool automatically derives a number of constraints to prevent infeasible solutions, such as a function mapped on a non-master processor element calling a function mapped on another processing element. Figure 4.15 shows a partial log that displays the constraints that help achieve feasible solutions on the hArtes platforms. Table 4.3 shows that 677 constraints are generated for the beamformer application, whereas 455 constraints are generated for the wave-field synthesis application.

4.5.3 Performance

We further analyzed the performance achieved by the different implementations in Fig. 4.11. The main results are presented in Tables 4.4 and 4.5 for BF and WFS

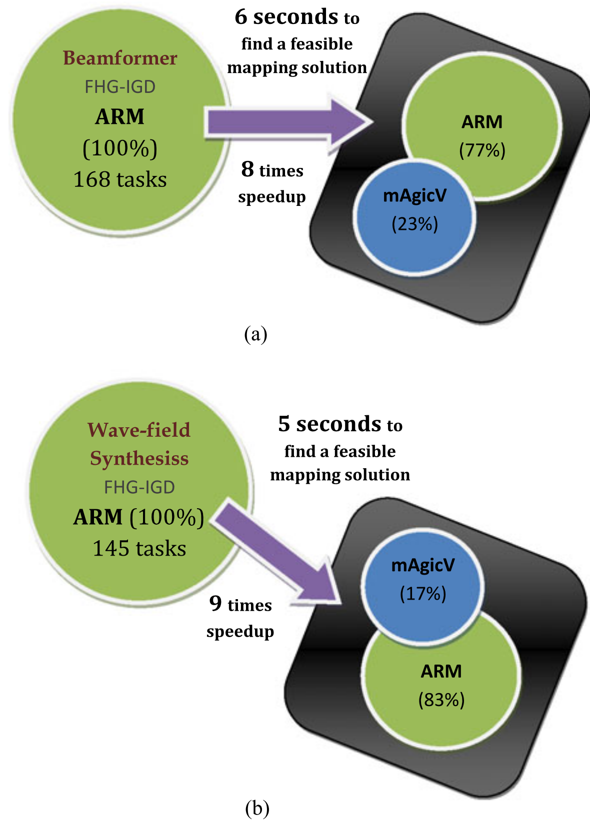


Fig. 4.14 Main results of the hArmonic tool in combination with the synthesis toolbox and the hHP: (**a**) beamformer and (**b**) wave-field synthesis

respectively. The values to consider are the occupancies in the rightmost column, with a unit value corresponding to the computed number of cycles the system needs to run in realtime with 100% load. Speed-ups between implementations can be easily computed by dividing occupancies, and in particular the speed-ups reported in Table 4.3 correspond to the ratios between "GPP" and "hArtes + DSPLib" rows.

For the beamformer algorithm, the GPP implementation needs almost 8 times longer than would be required for real-time operation. However, even the automatically mapped approach does not fully reach real-time constraints, which is due to the suboptimal implementation of the DSP kernels. In these kernels, no particular manual source code optimization has been applied, which leads to suboptimal compilation using the DSP tool-chain. This problem can be alleviated by making available to the mapper the implemention of the manually optimized DSPLib functions to use within the computing kernels, namely the filter functions (processing blocks see above). In this case, the manual mapping performance is similar to the automatic case, thus confirming that the automatic mapper was not only able to decide correctly on the mapping solution, but provided a solution with a slight edge over the manually implemented filter functions. Another observation is that neither the mArtes approach nor the direct manual Diopsis implementation provided signif-

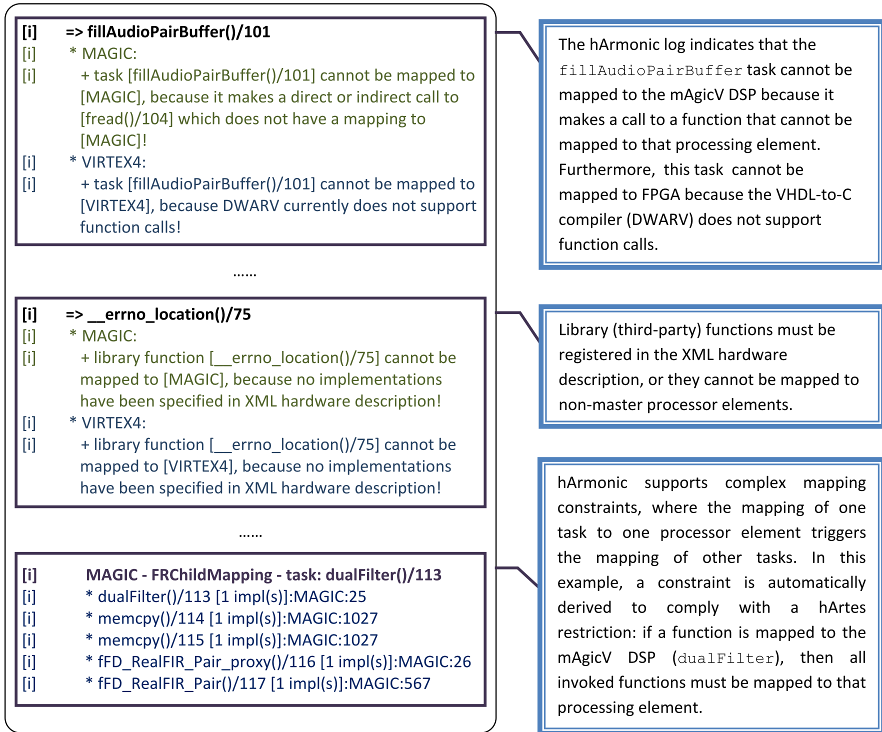


Fig. 4.15 A description of some of the hArtes constraints automatically derived by the hArmonic tool to ensure that the mapping solutions are feasible on the hArtes platform. This figure shows part of the log generated by hArmonic when processing the beamformer application. A total of 677 mapping constraints are identified for this application

icant improvements over the automatic mapping implementation (provided all use the DSPLib optimizations).

WFS yields a similar picture, generally showing that the hArtes manual and automatically mapping methods are very close (the automatic mapping being even marginally more performant). However, compared to the direct manual implementation, the performance observed was significantly inferior: None of the hArtes implementations was able to achieve similar performance figures compared to the direct manual Diopsis implementation on hHP, with occupancy values approximately 20% worse. A reason for is that memory transfer for the high number of channels to be processed makes up for a non-negligible portion of the frame time. This is consistent with the comparison of the measured transfer values between D-MEM and FPGA (where also the ADAT interface is implemented) and D-MEM and DSP in Fig. 4.13, which have approximately factor 20 in transfer time/bandwidth (from D-MEM: 21.87 vs. 1.01; to D-MEM: 21.21 vs. 1.02). As the tool-chain itself did not allow non-blocking transfers while processing, without specific precautions like double-buffering signals and explicit DMA control, the FPGA transfers fully claim

Table 4.4 Beamformer application performance (parameters see Table 4.2)

Approach	Description	Cycles	Occupancy
Realtime, 32 kHz	256 samples/frame, 100 MHz	800000	1.00
GPP	ARM only	6045903	7.57
hArtes	DSE with no DSPLib linking	2305912	2.80
hArtes + DSPLib	DSE + DSPLib linking	758868	0.95
mArtes + DSPLib	Man + DSPLib linking	764411	0.96
DSPLib manual	Gold standard impl.	743549	0.93

Table 4.5 WFS application performance (parameters see Table 4.2)

Approach	Description	Cycles	Occupancy
Realtime, 48 kHz	512 samples/frame, 100 MHz	1066666	1.00
GPP	ARM only	6598241	6.19
hArtes	DSE with no DSPLib linking	2055912	1.93
hArtes + DSPLib	DSE + DSPLib linking	712762	0.67
hArtes + FPGA	DSP + Filter in FPGA	2064202	1.94
mArtes + DSPLib	Man + DSPLib linking	724611	0.68
DSPLib manual	Gold standard impl.	600697	0.56

their toll in terms of frame time. Nevertheless, it was possible to achieve a version of the WFS application with fully automatic mapping, and the optimization of FPGA memory communication is seen as a future task beyond demonstrating the capability of the tool-chain. The FPGA used as a filter device led to similar difficulties. Here it is to note that the hardware constraints of the hHP are rather difficult to comply with, and either a future hardware platform that is specialized to multichannel audio may include this new knowledge into the next version, and/or the tool-chain may improve its capabilities in this respect.

4.6 Conclusion

In this chapter, the integration of two audio array processing algorithms with embedded hardware has been described and validation results presented. The algorithms considered, beamforming and wave-field synthesis, have been unified to a generic structure that allowed creating similar implementations. Results obtained during development of the algorithms with the tool-chain were (1) qualitative evaluations of how the tool-chain as a whole supports embedded development, (2) qualitative results on how automatic mapping distributes the algorithm over the different processing elements and (3) what performance could be achieved with these mappings.

In particular, we compared the hArtes tool-chain results with baselines and manual embedded implementations of the algorithms.

Generally, the automatic approach yielded executables for the heterogeneous platform that performed as well as the manual mapping within the scope of the hArtes framework. This fulfils the original promise of hArtes to deliver an automatic approach for targeting algorithms to heterogeneous platforms.

However, for the audio array processing algorithms at hand, the tool-chain was only able to achieve the best performance with the help of an optimized DSP library (the DSPLib). The tool-chain itself as an integrated tool is, however, not held responsible for this because it relies on the native sub-tool-chains for the different processing devices, in this case the Target mAgicV compiler. Furthermore, memory communication turned out to be a neuralgic point in the applications, and while the tool-chain made a remarkably good job at automatically mapping and abstracting memory locations and transfers across different processing devices, there are still cases where it slows down the algorithm compared to a manual implementation.

With this criticism stated to be addressed for future success, the overall usage experience of the final tool-chain was very smooth and efficient. The integration of the source code targeted at different processing elements of complex heterogeneous platforms as well as its close resemblance to platform-independent code go a long way in terms of maintaining and reviewing project structure and algorithms as well as the separation of algorithm and embedded platform knowledge. The integration with a widely-used development platforms like Eclipse is another important factor, as is the usage of standard build mechanism based on Makefiles that are ubiquitous in the C-based development world. Moreover, hArtes makes available its extensive profiling capabilities (that also its partitioning builds on) to developers and thus gives them additional support to optimize applications. Although debugging on the heterogeneous system is not yet very comfortable (this would justify a project in its own right), in practice it turned out to be useful to track particular critical points in the implementations.

Finally, starting tool-chain-based development from high-level entry points like NU-Tech and SciLab is perceived as a useful option (although for the telepresence application, NU-Tech was used as an algorithm monitoring environment rather than an entry point, and Matlab was used instead of SciLab for early proofs of concept).

We cannot quantify the exact reduction of development time achieved by using hArtes because implementation of the audio array processing algorithms was done in parallel with tool-chain development and did therefore not fully pursue the typical sequence of optimisations. However, it is clear that the hardware abstractions, mapping and profiling support provided by the hArtes tool-chain provide an enormous advantage over having to cope with fine-grained hardware and API specifications. We estimate that, if we had to implement our algorithms for heterogeneous hardware like the hHP from scratch and without platform knowledge, we may have saved up to 50% of the development effort.

References

1. Beracoechea, J.A., Torres-Guijarro, S., García, L., Casajús-Quirós, F.J.: On building immersive audio applications using robust adaptive beamforming and joint audio-video source localization. EURASIP J. Appl. Signal Process. **2006**, 40960 (2006), 12 pp.
2. Heinrich, G., Leitner, M., Jung, C., Logemann, F., Hahn, V.: A platform for audiovisual telepresence using model- and data-based wave-field synthesis. In: Proc. 125th Convention of the AES, San Francisco, Oct. 2008
3. Lombard, M., Jones, M.T.: Identifying the (tele)presence literature. PsychNology **5**(2), 197–206 (2007)
4. Lombard, M., Jones, M.T.: Defining presence. In: Biocca, F., Ijsselsteijn, W.A., Freeman, J., Lombard, M. (eds.) Immersed in Media: Telepresence Theory, Measurement and Technology. Routledge, London (2011, in press)
5. Mabande, E., Schad, A., Kellermann, W.: Robust superdirectional beamforming for hands-free speech capture in cars. In: Proc. NAG/DAGA Int. Conf. on Acoustics, Rotterdam, Netherlands, March 2009
6. Parra, L.C.: Steerable frequency-invariant beamforming for arbitrary arrays. J. Acoust. Soc. Am. **119**(6), 3839–3847 (2006)
7. Rabenstein, R., Spors, S.: Spatial sound reproduction with wave field synthesis. In: Proc. Cong. Ital. Sect. of the AES, Como, Italia, Nov. 2005
8. Shinoda, K., Mizoguchi, H., Kagami, S., Nagashima, K.: Visually steerable sound beamforming method possible to track target person by real-time visual face tracking and speaker array. In: Proc. IEEE Int. Conf. Systems, Man and Cybernetics, vol. 3, pp. 2199–2204 (2003)
9. van Veen, B., Buckley, K.M.: Beamforming: a versatile approach to spatial filtering. IEEE ASSP Mag. **5**, 4–24 (1988)
10. Verheijen, E.: Sound reproduction by wave field synthesis. Ph.D. thesis, TU Delft (1998)
11. Ward, D.B.: Theory and application of wide-band frequency invariant beamforming. Ph.D. thesis, Australian National University (1996)

Chapter 5
In Car Audio

Stefania Cecchi, Lorenzo Palestini, Paolo Peretti, Andrea Primavera, Francesco Piazza, Francois Capman, Simon Thabuteau, Christophe Levy, Jean-Francois Bonastre, Ariano Lattanzi, Emanuele Ciavattini, Ferruccio Bettarelli, Romolo Toppi, Emiliano Capucci, Fabrizio Ferrandi, Marco Lattuada, Christian Pilato, Donatella Sciuto, Wayne Luk, and Jose Gabriel de Figueiredo Coutinho

This chapter presents implementations of advanced in Car Audio Applications. The system is composed by three main different applications regarding the In Car listening and communication experience. Starting from a high level description of the algorithms, several implementations on different levels of hardware abstraction are presented, along with empirical results on both the design process undergone and the performance results achieved.

5.1 Introduction

In the last decade Car infotainment (i.e. the combination of information with entertainment features) systems have attracted many efforts by industrial research because car market is sensible to the introduction of innovative services for drivers and passengers. The need for an Advanced Car Infotainment System (ACIS) has been recently emerging, able to handle issues left open by CIS systems already on the market, and to overcome their limitations. Due to traffic congestion and growing distance from home to workplace, people spend more and more time in car, that hence becomes an appealing place to do many common activities such as listening to music and news, phone calling and doing many typical office tasks. Narrowing our focus to audio, the key role of the CIS is that it lets the driver concentrate on the road and at the same time it manages many different processing functions such as high quality music playback, hands-free communication, voice commands, speaker recognition, etc. Moreover, from the signal processing point of view, the wide band nature of the audio signal adds complexity while the requested quality calls for

F. Piazza (✉)
DIBET—Universitá Politecnica delle Marche, Via Brecce Bianche 1, 60131 Ancona, Italy
e-mail: f.piazza@univpm.it

K. Bertels (ed.), *Hardware/Software Co-design for Heterogeneous Multi-core Platforms*,
DOI 10.1007/978-94-007-1406-9_5, © Springer Science+Business Media B.V. 2012

high precision signal processing, making in-car audio processing a very open research and development field. Therefore new architectures are needed to overcome the nowadays limitations. In this context, one of the main objective of the European hArtes Project (Holistic Approach to Reconfigurable real Time Embedded Systems) is to develop an ACIS, capable to meet market requirements. It is achieved with a multichannel input (microphone array) and output (speaker array) platform managed by NU-Tech framework [33] and implemented on a real car named hArtes CarLab. In particular such system has been adopted in the hArtes project [14, 33] as a proof-of-concept in order to test and assess the project methodologies. In fact the main goal of the hArtes project is to provide, for the first time, a tool chain capable of optimal, automatic and rapid design of embedded systems from high-level descriptions, targeting a combination of embedded processors, digital signal processing and reconfigurable hardware. Three main different applications have been developed for the ACIS within the hArtes project:

- Enhanced In-Car Listening Experience: It is necessary to develop audio algorithms to improve the perceived audio quality in cars, making the listening environment more pleasant, taking into account specific features of the car cabin.
- Advanced In-Car Communication: The advanced in-car communication scenarios are based on speech and audio signal processing in order to enhance hands-free telephony, in-car cabin communication and automatic speech recognition for an efficient user interface.
- In-Car Speaker and Speech Recognition: The automatic speech and speaker recognition modules aim to provide an improved man-machine interface (MMI) for user authentication and command and control of software applications.

Section 5.2 introduces the state of the art of the three main applications. Section 5.3 is focused on the description of the audio algorithms implemented to improve the audio reproduction and to manage the communication and interaction features. Section 5.4 describes the hArtes toolchain and how has been applied for the implementation of the selected algorithms. Section 5.5 reports the experimental results for the PC-based prototype and for the final embedded platform prototype.

5.2 State of the Art for In Car Audio

In the following, the state of the art for each of the three main fields of application will be reported.

5.2.1 Enhanced In-Car Listening Experience

Despite of the advent of consumer DSP applied to sound reproduction enhancement, few applications have been developed for the in-car listening experience especially

due to characteristics of automobile environment which is not an ideal listening environment. The automobile is a well-known small noisy environment with several negative influences on the spectral, spatial and temporal attributes of the reproduced sound field [13, 26]. Specially, depending on the absorbing or reflecting interior materials, the position of loudspeakers and the shape of the car cabin, the reflected sounds attenuate or amplify the direct sound from the loudspeakers [59]. One of the most comprehensive work found in literature, regarding a complete automotive audio system is presented in [28]. The embedded system comprises different processing units: a parametric equalizer is used to correct irregularities in the frequency response due to car cabin characteristics; an adjustable delay is needed in order to equalize arrival times due to loudspeakers position; a dynamic range control is considered for compressing and amplifying the reproduced material taking into account the measured background noise level; a surround processor to artificially recreate a more appealing listening environment. Some of the previous aspects have been extensively studied singularly in other works. The equalization task has been investigated thoroughly: in [20] some fixed equalization algorithms based on different inversion approaches is presented together with a surround processor to remove and add unwanted/wanted reverberation components. With the advent of multichannel audio content, different schemes have been considered to create a compelling surround experience. In [15] several audio technologies that support reproduction of high quality multichannel audio in the automotive environment are illustrated. Although these technologies allow a superior listening experience, different open problems remain such as off-axis listening position of the passengers. In [60] a valid solution to improve the surround imaging is presented: it is based on comb filtering designed taking into account inter-loudspeaker differential phase between two listening positions. A digital audio system for real time applications is here proposed. Its aims are substantially two: to develop a complete set of audio algorithms improving the perceived audio quality in order to make the listening experience more pleasant taking into account some specific features of the car cabin; to have a modular and reconfigurable system that allows to seamlessly add, remove and manage functionalities considering, for the first time, a PC based application.

5.2.2 Advanced In-Car Communication

Monophonic Echo Cancellation

Following the formalization of the adaptive complex LMS (Least Mean Square) in [65], the use of fast convolution methods for the derivation of FDAF (Frequency Domain Adaptive Filtering) have been proposed in order to reduce the overall complexity of the adaptive filter. These methods are either based on OverLap-and-Add (OLA) or OverLap-and-Save (OLS) method. The OLS method is more generally used since it has a direct and intuitive interpretation. A first implementation of an

adaptive filter in the frequency-domain was proposed in [16]. An exact implementation of the LMS (Fast LMS) was proposed in [21] with the calculation of a constrained gradient. A sub-optimal version with the calculation of the unconstrained version was derived in [40] for further reducing the complexity, and an approximation of the constrained gradient was proposed in [61] using a time-domain cosine window. The application of FDAF to acoustic echo cancellation of speech signals have to solve additional problems due to the speech signal properties (non-stationary coloured signal) and due to the acoustical path properties (non-stationary acoustical channel with long impulse responses). The frequency-domain implementation of block gradient algorithm exhibits better performances than its time-domain counterpart since it is possible to perform independent step-size parameter normalization for each frequency bin, acting as a pre-whitening process. The major advance for its application to acoustic echo cancellation was to partition the adaptive filter in order to identify long impulse responses as implemented in the MDF (Multi-Delay Filter) algorithm in [63]. In this approach it is possible to choose an arbitrary FFT size for filtering whatever the size of the impulse response to be identified. The processing delay is by consequent also significantly reduced for transmission applications. Further improvements in terms of performances have been achieved in the GMDF (Generalized Multi Delay Filter), [3, 44], using overlapped input data leading to an increased updating rate. The residual output signal is regenerated using a WOLA (Weighted OverLap-and-Add) process. More recently, an improved version of the MDF filter has been described in [9], the Extended MDF (EMDF) which takes into account the correlation between each input blocks resulting from the filter partitioning. One of the advantages of the frequency-domain implementation is also to implement globally optimized solutions of the adaptive filter for echo cancellation and speech enhancement as in [8, 12, 31] and [32]. The overall complexity of the combined AEC with the speech enhancement algorithm can further be reduced (common FFT) and enhanced performances can be achieved using noise-reduced error signal for adaptation.

Stereophonic Echo Cancellation

When generalizing the acoustic echo cancellation to the multi-channel case, one has to deal with the cross-channel correlation. This problem has been described in [62]. Most of the studies in the field of acoustic echo cancellation carried out in the nineties were devoted to the improvements of sub-optimal multi-channel algorithms, trying to achieve a good compromise between complexity and performances, [2, 4]. More recently, a more optimal derivation of the MDF filter was proposed in [10], where the EMDF filter is generalized to the multi-channel case leading to a quasi-optimal frequency-domain implementation taking into consideration both the correlation between sub-blocks resulting from the partitioning process and the cross-channel correlations. This solution is targeting the identification of very long impulse responses with high-quality multi-channel audio reproduction. However the gain in performances has not been proven to be significant for the limited stereophonic case in a car environment for which impulse responses are shorter than in

rooms. Results showing the combination of a sub-optimal FDAF with an ASR system in a car environment have been given in [5].

In-Car Cabin Communication

Here the main objective is to improve the communication between passengers inside a car or vehicle. The difficulty of in-car cabin communication results from various factors: noisy environmental conditions, location and orientation of passengers, lack of visual feedback between passengers. This is particularly true with the emergence of van vehicles on the market. A critical scenario is for passengers located at the rear of the car while listening to the driver in noisy conditions. This topic has been recently addressed in the literature, and is also part of working groups at standardization bodies. The basic principle is to pick-up passengers voice with one or several microphones and to reinforce the speech level through loudspeakers. The main challenges are: to avoid instability of the system due to the acoustical coupling, to avoid reinforcing the noisy environment, and to keep the processing delay below an annoyance level. In [37], a characterization of the transfer functions depending on the microphone position inside a vehicle is performed, and a basic system with only two microphones and two loudspeakers is described, using two acoustic echo cancellers. The authors also proposed to include a feedback cancellation based on linear prediction. In [39, 48, 49], the proposed system is composed of an acoustic echo canceller to remove the echo signal on the microphone which picks-up the passenger's voice, follower by an echo suppression filter for removing the residual echo and a noise suppressor in order to avoid reinforcing the environmental noise. A speech reinforcement of up to 20 dB is claimed. Further improvements of the proposed system are given in [50–53]. In [23–25], the authors proposed similar systems but also performed some subjective speech quality evaluation showing that the developed system was preferred to a standard configuration at 88.6% (rejection of 4.3%) at 130 km/h on highway, and at 50.7% (rejection of 19.7%) at 0 km/h with the vehicle parked closed to a highway. These results demonstrate the interest of such system in realistic conditions. Additional intelligibility tests also have shown a 50% error reduction at 130 km/h on highway. If the components of such systems are basically related to acoustic echo cancellation and noise reduction, the major challenges in this particular application are related to stability issues, processing delay and speech signal distortion resulting from the acoustical cross-coupling.

5.2.3 In-Car Speaker and Speech Recognition

Automatic Speech Recognition

Automatic speech recognition systems have become a mandatory part of modern MMI applications, such as the automotive convenience, navigational and guidance

system scenarios, in addition to many other examples such as voice driven service portals and speech driven applications in modern offices.

Modern architectures for automatic speech recognition are mostly software architectures generating a sequence of word hypotheses from an acoustic signal. The dominant technology employs hidden Markov model (HMMs). This technology recognises speech by estimating the likelihood of each phoneme at contiguous, small regions (frames) of the speech signal. Each word in a vocabulary list is specified in terms of its component phonemes. A search procedure is used to determine the sequence of phonemes with the highest likelihood. Modern automatic speech recognition algorithms use monophone or triphone statistical HMMs, the so called acoustic models, based on Bayesian probabilistic and classification theory. Language models, which give the probability of word bigrams or trigrams, are defined using large text corpora. The information provided with the language model is advantageous especially for continuous automatic speech recognition and can substantially increase the performance of the recognition system. This explains the intrinsic difficulty of a speech-to-phoneme decoding, where only the acoustic information is available. The simplest ASR mode is the recognition of isolated words as is the case for the in-car hArtes application. Whole-word, monophone or triphone acoustic HMM models can be applied. Realistic word error rates are in the order of less than 5% with vocabularies of approximately 50 words. This ASR mode is usually applied for command driven tasks and is that best suited to the requirements of the hArtes project.

Automatic Speaker Recognition

The goal of a speaker authentication system is to decide whether a given speech utterance has been pronounced by a claimed client or by an impostor. Most state-of-the-art systems are based on a statistical framework. In this framework, one first needs a probabilistic model of anybody's voice, often called a world model and trained on a large collection of voice recordings of several people. From this generic model, a more specific client-dependent model is then derived using adaptation techniques, using data from a particular client. One can then estimate the ratio of the likelihood of the data corresponding to some access with respect to the model of the claimed client identity, with the likelihood of the same data with respect to the world model, and accept or reject the access if the likelihood ratio is higher or lower than a given threshold, selected in order to optimize either a low rejection rate, a low acceptance rate, or some combination of both.

Different scenarios can take place, mainly text dependent and text independent speaker authentication:

- In the context of text independent speaker authentication systems, where the trained client model would in theory be independent of the precise sentence pronounced by the client, the most used class of models is the Gaussian Mixture Model (GMM) with diagonal covariance matrix, adapted from a world model using Maximum A Posteriori (MAP) adaptation techniques.

- In text dependent speaker authentication, the system associates a sentence with each client speaker. This enables the use of the expected lexical content of the sentence for better modelling and robustness against replay attacks. Furthermore, in the more complex text prompted scenario, the machine prompts the client for a different sentence at every access, which should be even more robust to replay attacks, since the expected sentence for a given access in randomly chosen. On the other hand, models known to efficiently use this lexical information, such as HMMs, need more resources (in space and time, during enrolment and test) than text independent models.
- A mixed approach, which takes advantages of text-independent and text-dependent systems. In this approach, called GDW (Gaussian Dynamic Warping), the time structural information coming from the pronounced sentences are seen as a constraint of a text-independent system, allowing a balanced training of speaker specific information and text-dependent information. This characteristic allows text prompted speaker authentication systems to be built with a very light enrolment phase, by sharing the large part of the model parameters between the speakers. The main drawback of speaker authentication systems based on time structural information consists in the need of a specific acoustic model for each temporal segment of the sentence. Despite the fact that a large part of the acoustic parameters is shared between the users, the corresponding memory and computation resources are not negligible in the framework of embedded systems. A possible solution consists in building an unique model, to compute the transformation of this model for each of the temporal segments, and to store only the transformation parameters.

5.3 Algorithms Description

A detailed description of the selected algorithms will be reported for each of the three fields of application.

5.3.1 Enhanced In-Car Listening Experience

In order to achieve an improvement of the perceived sound, two different tasks can be identified: the former tries to modify the overall behavior of the system, while the latter operates on each loudspeakers channel (Fig. 5.1). Regarding the first objective, a module composed by a digital crossover network in order to split the audio signal in different bands is employed [38]. Then a parametric equalizer for each band is requested to compensate sound coloration due to several resonances in the small car environment. Each parametric (IIR) filter can produce a phase distortion: a phase processor is hence considered in order to correct the phase distortion and to enable adjustments among channels. The second objective comprises audio equalization with two different application scenarios: in the fixed case, the equalization will be

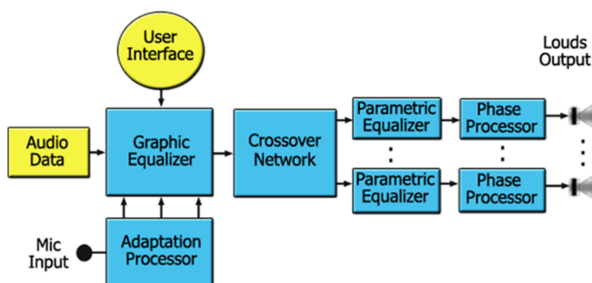


Fig. 5.1 Enhanced listening experience algorithms

done in sub-bands without considering the environment changes and for each frequency band it would be possible to modify the gain; the adaptive scenario provides for an adaptive equalization with reference to some car environment modifications (e.g., number of passengers).

In the following each constituent unit of the system will be described.

Crossover Network

In order to avoid distortions and drivers damages it is necessary to develop a Crossover Network able to split the audio spectrum into desired bands [57]. This network should satisfy some well known design objectives, as expressed in [54]. A linear phase mixed FIR/IIR crossover, satisfying such design requirements, based on a tree structure is employed [54]. The lower branches of the tree are obtained by IIR filtering in order to reduce the overall delay with respect to FIR filtering that would have required thousands of taps. IIR filters are derived from all-pass filters designed with linear phase constraint. A FIR tree is grafted on the upmost branch of the IIR one, obtaining the higher frequency channels, whose filters require less coefficients, and preserving linear phase and ensuring an overall low system delay.

Parametric Equalizer

As pointed out in [28], a Parametric Equalizer is a useful tool in order to correct frequency responses and also to amplify low frequency components masked by the background noise. The parametric equalizer is designed through high-order Butterworth or Chebyshev analog prototype filters [47], generalizing the conventional biquadratic designs and providing flatter passbands and sharper band edges.

Phase Processor

In a vehicle cabin the loudspeaker location possibilities are rather restricted. Off-center listening position is inevitable. The delay difference should be compensated through digital delay to equalize the sound arrival times from loudspeakers to the

listening position. In this context a Fractional Delay can be used to achieve a better sound alignment. Among the various approaches reported in literature, we chose to resort to a computationally efficient variable fractional delay approach, based on coefficients polynomial fitting [67]. As well as this, a unit capable of rotating the phase spectrum by a certain angle could be introduced, as a mean of phase equalization. This feature has been realized through an all-pass filter with phase specification designed with LSEE approach [29].

Equalizer

Equalization is implemented to enhance tone quality and modify frequency response. Equalizers are used to compensate for speaker placement, listening room characteristics (e.g., in automotive application, to have low frequencies emphasized in the presence of background noise), and to tailor to personal taste (e.g., with relation to particular kinds of music). This compensation is accomplished by cutting or boosting, that is, attenuating or amplifying a range of frequencies. A graphic equalizer is a high-fidelity audio control that allows the user to see graphically and control individually a number of different frequency bands in a stereophonic system. The solution we have adopted is based on an FFT approach ensuring linear phase and low computational cost due to fast FIR filtering implemented in the frequency domain [22]. With respect to efficiency, it is possible to design very efficient high quality digital filters using multirate structures or by using well known frequency domain techniques [22]. Symmetric FIR filters are usually considered for digital equalization because they are inherently linear phase and thus eliminate group delay distorsions [58]. An efficient FIR filtering implementation is achieved by using frequency domain techniques as the overlap and add method, splitting the entire audio spectrum in octave bands. A proper window, whose bandwidth increases with increasing center frequency, has been used to modulate the filter transitions and the resulting smoothness of the equivalent filter [22]. Starting from the cabin impulse response, a set of fixed curves has been derived to compensate for some spectral characteristics of the car environment. The algorithm can be extended to consider a feedback signal from a microphone near the passenger position to which each gain should be adapted. As fixed and adaptive equalization substantially share the overall architecture, only the adaptive approach will be described. A clear advantage of the proposed solution is that the fixed scheme can be easily turned to an adaptive one, necessary in realistic environment.

5.3.2 Advanced In-Car Communication

In this part the different speech and signal processing components required for improved communication and interaction through an Automatic Speech Recognition (ASR) system are described. The corresponding objectives are the following: hands-free function for in-car mobile telephony, stereophonic echo cancellation for ASR

barge-in capability while using the in-car audio system, and speech enhancement for communication and signal pre-processing for ASR. In the following, the corresponding selected algorithms are described, and some results of the integration process in the CarLab using NU-Tech software environment are given.

Monophonic Echo Cancellation

This feature is required for the realization of the hands-free function in order to cancel the acoustic echo signal resulting from the far-end speech emitted on the in-car loudspeakers. Monophonic echo cancellation can also be used to provide barge-in capability to Automatic Speech Recognition system when using speech synthesis based Human Machine Interface. The proposed solution should not only handle narrow-band speech but also wide-band speech, in order to cope with the standardized Wide-Band Adaptive Multi-Rate speech codec (AMR-WB) for mobile telephony. For performance and complexity reasons, a FDAF has been implemented. It includes a partitioning structure of the adaptive filter in order to vary independently the length of the identified impulse response and the size of the FFT, as introduced in [63]. Additionally, overlapped input data process is used to both improve tracking capability and reduce the algorithmic delay, as described in [44]. The residual output signal is then regenerated using WOLA (Weighted OverLap-and-Add).

Stereophonic Echo Cancellation

Extended FDAF algorithm to the stereophonic case has been also considered in order to cancel the acoustic echo signals coming from the in-car embedded loudspeakers when operating the ASR system while the radio is turned on. When generalizing the acoustic echo cancellation to the multi-channel case, one has to deal with the cross-channel correlation. This problem has been described in [62]. A quasi-optimal frequency-domain implementation taking into consideration both the correlation between sub-blocks resulting from the partitioning process and the cross-channel correlations has been proposed in [11]. This algorithm is particularly well-suited for long impulse responses. Due to shorter impulse responses encountered in the car environment no significant gain is expected. A simple sub-optimal extension of the monophonic FDAF has been implemented, using the global residual signal for the adaptation of each adaptive filter.

Single-Channel Speech Enhancement

Noisy speech signals picked-up in a car environment have been one of the major motivations for the development of single-channel speech enhancement techniques either for improved speech quality for phone communication or ASR applications.

Methods proposed in [18] and [19] are considered as state-of-the-art approaches especially regarding the naturalness of the restored speech signal. These algorithms implement respectively an MMSE (Minimum Mean Squared Error) estimator of the short-term spectral and log-spectral magnitude of speech. The spectral components of the clean signal and of the noise process are assumed to be statistically independent Gaussian random variables. The log-MMSE Ephraim & Mallah algorithm has been implemented together with a continuous noise estimation algorithm as described in [56]. This algorithm has been designed to deal with non-stationary environments and remove the need for explicit voice activity detection.

Multi-Channel Speech Enhancement

The use of several microphones for speech enhancement has been also studied for in-car applications. In general, the different microphones signals are first time-aligned (beam-forming), and the resulting signal is then further processed either using adaptive filtering techniques or adaptive post-filtering. For cost and complexity reasons, a four microphone array has been studied using cardioid microphone (AKG C400 BL) with 5 cm spacing. The different channels are continuously synchronized using appropriate time delays and summed together to form the output signal of the beam-former. The array is steered in different possible directions and the selected direction is the one which maximized the output power of the steered beam-former, known as the Steered Response Power (SRP) criterion. Improved performance is obtained by using the Phase Transform (PHAT) as a spectral weighting function. An evaluation of the resulting SRP-PHAT criterion can be found in [17]. Different post-filters have been implemented to further reduce non-localized noise interferences at the output of a beam-former. The first commonly used post-filter is known as Zelinski post-filter, as described in [66], which is an adaptive Wiener post-filter applied to the output of the beam-former. Various improvements have also been considered as proposed in [6, 41, 43] and [42].

5.3.3 In-Car Speaker and Speech Recognition

The automatic speech and speaker identification modules provide an improved MMI. Several functionalities could be based on these modules, in order to improve the conviviality of the user interface. A speech recognition system provides hands-free operations well suited for various in-car devices and services such as: radio equipment, multimedia management, CD player, video player and communications systems (mobile phones, PDA, GPS, etc.). Isolated/connected word recognition is a simpler task than continuous speech recognition (which allows to integrate a speech recognition system into an ACIS). Finding a command word in a continuous audio stream is one of the hardest speech recognition tasks, so we decide to use a push-when-talk approach (driver should press a button before saying the command to

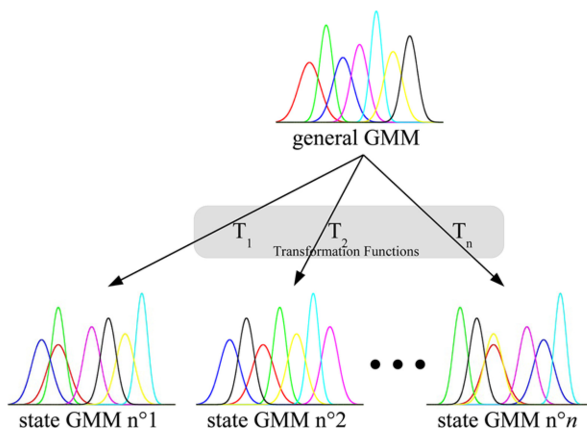


Fig. 5.2 General approach for speech recognition

recognize). Automatic speaker recognition provides user authentication (driver) for secured access to multimedia services and/or specific user profiles (seat settings, favorite radio station, in-car temperature, etc.). For embedded speech/speaker recognition, the highlights are usually the memory and computational constraints. But realistic embedded applications are also linked to several scenario constraints, giving short speech utterances, few training data and very noisy environments. Also, such systems should work with degraded speech quality. Here, two applications are targeted: isolated/connected word recognition (for command and control) and speaker identification (to load driver parameters). The ASR module has been developed using HMMs [27] and is based upon LIASTOK [35]. The development is combined with the speech and audio modules for speech pre-processing. Referring to Fig. 5.2, we begin with a GMM which represents the whole acoustic space and then derive from it the necessary HMM-state probability density functions to represent each phoneme. This approach to model acoustic space was fully presented in [36]. A word is so composed of a number of sequential phonemes. As the recognizer is phoneme-based, it allows a lexicon easily modifiable. This is deemed to sit particularly well with the hArtes paradigm. It will allow to adjust the vocabulary, according to modifications made to the ACIS or as new applications or functionalities are introduced. In addition, the lexicon can contain any number of different phonetic transcriptions per vocabulary item thus assisting recognition with different pronunciations and accents. Moreover this approach used for acoustic modeling allows easy adaptation to a speaker or to a new environment [35].

The remainder of the decoding algorithm is essentially a traditional Viterbi algorithm [64]. Likelihoods are calculated on the fly with highly efficient and sophisticated likelihood calculations. Given an audio recording the likelihood of the observed data given each model is calculated and that which produces the greatest likelihood defines the decoded or recognized word. The goal of a speaker authentication system is to decide whether a given speech utterance was pronounced by the claimed speaker or by an impostor. Most state-of-the-art systems are based on a statistical framework (Fig. 5.3). In it, one first needs a probabilistic model of anybody's voice, often called Universal Background Model (UBM or world model) and trained

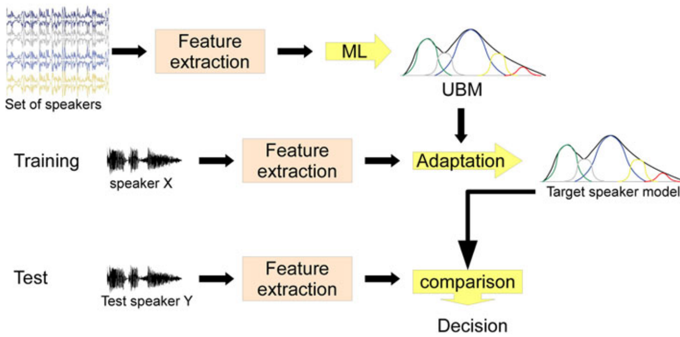


Fig. 5.3 General approach for speaker recognition

on a large collection of voice recordings of several people. From this generic model, a more specific, client-dependent model is then derived using adaptation techniques, using data from a particular client. Our work is based on ALIZE [1] toolkit and dedicated to such embedded applications. To save both memory and computation time, only a few part of UBM components are adapted and saved (other mean parameters, weight parameters and variance parameters are taken from the UBM model) like in [7] or [30]. One can then estimate the ratio of the likelihood of the data corresponding to some access with respect to the model of the claimed client identity, with the likelihood of the same data with respect to the world model, and accept or reject the access if the likelihood ratio is higher or lower than a given threshold.

5.4 Applying the hArtes Toolchain

A complete description of the hArtes toolchain application on the selected algorithms follows.

5.4.1 GAETool Implementation

The hArtes Graphical Algorithms Exploration (GAE) tool is the highest level of the hArtes toolchain based on the NU-Tech Framework [33, 46] (Fig. 5.4). It can be seen as a development environment for the interaction of DSP algorithms; the user can program his own C/C++ objects separately and then consider them as blocks, namely NUTS (NU-Tech Satellites), to be plugged-in within the available graphical design framework. Each NUTS is fully configurable to provide the needed number of inputs and outputs. Therefore GAE tool has two basic functionalities:

1. Assist the designers to instrument and possibly improve the input algorithm at the highest level of abstraction so that feedback concerning the numerical and other high-level algorithmic properties can be easily obtained.

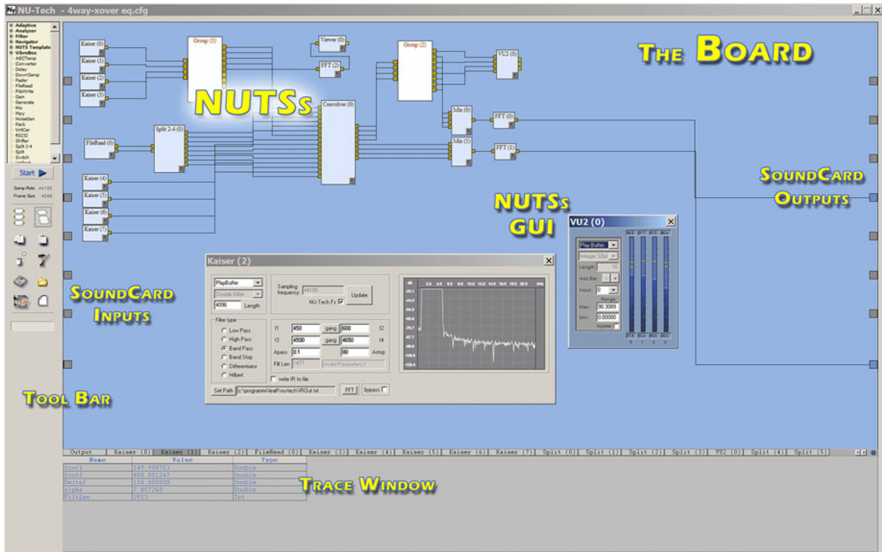


Fig. 5.4 NU-Tech Framework

2. Translate the input algorithms described in different formats and languages into a single internal description common for the tools to be employed further on. In the hArtes tool-chain, this internal description will be done in ANSI C language.

The GAE-tool accepts existing (e.g., NU-Tech library by Leaff) or new functional blocks written in C as input description. Integration of existing functional blocks allows component reuse: project-to-project and system-to-system. The GAE-tool also supports reconfigurable computing in the very initial phase allowing different descriptions of the same part of the application to let hardware/software consequent partitioning explore different implementations with respect to different reconfigurable architectures. Besides the C based description of the input application, outputs of the GAE tool contain directives provided by the user and profiling information collected during algorithm development. Possible user directives concern parallelism, bit width sizing, computing type (e.g., systematic vs. control) and all the parameters important to optimize the partitioning/mapping of the application on the available hardware. Possible profiling information collected by the GAE tool is the processor usage per single functional block with respect to the whole network of functional blocks and the list of possible bottlenecks. Exploiting NU-Tech plug-in architecture, every modules of the applications have been developed as C NUTS with associated settings window, useful to change internal parameters. Figure 5.5 shows the GAEtool integration of single channel speech enhancement.

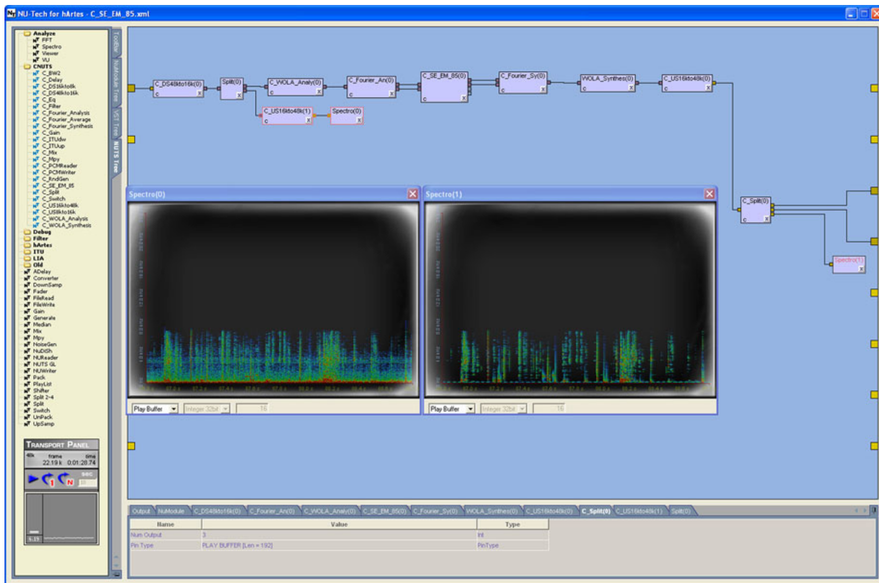


Fig. 5.5 NU-Tech integration of single-channel speech enhancement

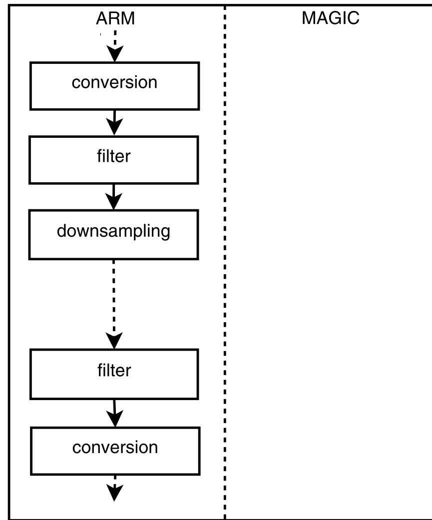
5.4.2 Task Partitioning

Zebu tool has been applied to a reduced version of the Advanced In-Car Communication application. The reduced version consists of the pre-processing and the post-processing processes required by the Stationary Noise Filter. The main kernel of the application is characterized by a sequence of transformations applied to the sound samples: Fourier transforms, filters, Weighted Overlap-and-Add, downsampling and upsampling.

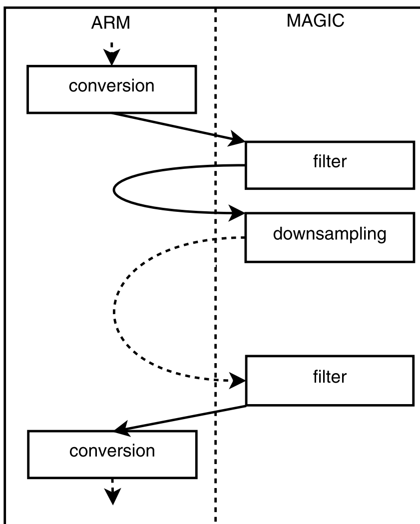
The application has been partitioned and mapped by Zebu for the execution on the Atmel's DIOPSIS board and the performance of the resulting versions of the application are measured directly on the board.

Figure 5.6(a) shows the plain execution of the kernel, without exploiting the DSP processor. Then, Fig. 5.6(b) shows the effects of applying only the mapping, that assigns most of the functions for the execution on the DSP. In this case, we obtained a speed-up equals to 4.3x. However, the data exchanged among the different functions are still allocated on the main memory since it is the master processor which at each function invocation has to pass and read back the data processed by the single function.

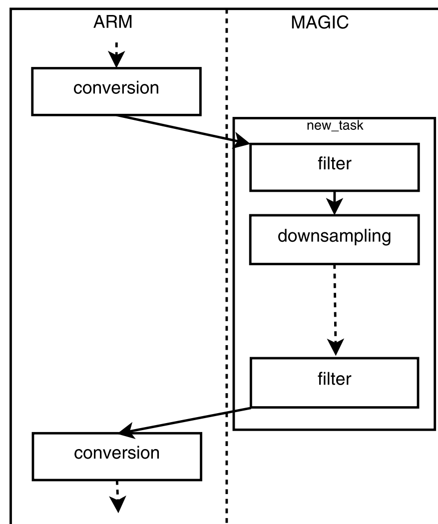
Finally, if Zebu performs also task transformations, such as sequential partitioning, it partitions the kernel of the application by grouping together the functions with highest data exchanging into a new task function `new_task`, as shown in Fig. 5.6(c). This task is then mapped on the DSP, so the functionalities mapped on the DSP are actually the same of the previous case. However, since these functionalities are now grouped in a single task, the data exchanged among the functions are



(a) Plain application



(b) Mapped application



(c) Partitioned and Mapped application

Fig. 5.6 Effects of partitioning and mapping on the kernel application

localized and do not need to be moved anymore inside and outside the DSP memory at each function call. As a result, this partitioning transformation increases the speed-up from 4.3x to 5.8x.

Table 5.1 Evaluation of the automatic mapping approach on the Audio Enhancement application and related kernels (Xfir, PEQ, FracShift, Octave), and the Stationary Noise Filter. The speedup corresponds to the ratio between the performance of the application running on the ARM and the mapping solution produced by the hArmonic tool

hArtes application	Total number of tasks	Number of tasks mapped to DSP	Time to compute solution (sec)	Speed up
Xfir	313	250	8	9.8
PEQ	87	39	3	9.1
FracShift	79	13	24	40.7
Octave	201	137	6	93.3
Audio Enhancement	1021	854	59	31.6
Stationary Noise Filter	537	100	17	5.2

5.4.3 Task Mapping

In this section, we report the results achieved with the hArmonic tool (Sect. 2.7.6) on the Audio Enhancement and Stationary Noise Filter applications using the automatic mapping approach.

Table 5.1 summarizes the main results. The total number of tasks measures the number of calls in an application, where each invoked function can be executed on a particular processing element. Note that the number of tasks is not necessarily an indication of the size of the application since each task can be of a different size granularity, however the number of tasks can affect the time to produce a mapping solution. The number of tasks mapped to DSP measures the fraction of the application that is executed outside the main processing element (ARM). The time to compute a solution corresponds to the total time required by the hArmonic tool to process each application, and the speedup corresponds to the ratio between the execution time of the application running on the ARM and the performance of the same application using the mapping solution derived by the hArmonic tool.

In addition to the Audio Enhancement application, Table 5.1 also presents the results of four of its kernels (Xfir, PEQ, FracShift, Octave) which are executed and evaluated independently. The number of tasks for the audio enhancement kernels is relatively small (from 79 to 313) and for three of the kernels, hArmonic produces an optimized mapping solution in less than 10 seconds, with speedups ranging from 9 times to 93 times.

The Audio Enhancement application has 1021 tasks and is the largest application in terms of the number of tasks evaluated by the hArmonic tool (Fig. 5.7(a)). The hArmonic tool takes 61 seconds to produce an optimized solution where 83% of the application is executed on the mAgicV DSP, and exhibits an acceleration performing 31 times faster. On the other hand, the Stationary Noise Filter has 537 tasks; hArmonic takes 17 seconds to produce a mapping solution that results in a 5 times speedup. The DSP coverage in this case is only 19%, however the task granularity of this application is understood to be on average larger than the Audio Enhancement

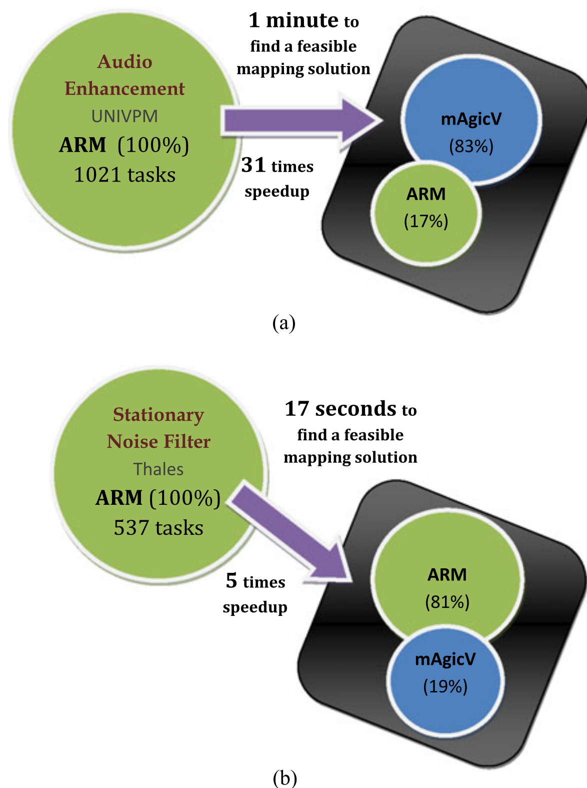


Fig. 5.7 Main results of the hArmonic tool in combination with the synthesis toolbox and the hArtes hardware platform: (**a**) Audio Enhancement application and (**b**) Stationary Noise Filter

application. Another difference between both applications is the use of DSPLib (part of the hArtes standard library), which in the case of the Stationary Noise filter is not used and therefore the mapping solution does not yield a performance improvement as large as that for the Audio Enhancement. However, since the Stationary Noise filter does not rely on the hArtes standard library, it allows the application to be compiled and run on other platforms, such as the PC. Note that the magnitude of the acceleration on both applications is influenced by many factors: the use of optimized libraries, manual management of the memory footprint, the quality of the synthesis toolbox (backend compilers such as hgcc and targetcc) and the hArtes hardware platform.

The results are achieved using the hArmonic tool in combination with the synthesis box of the hArtes toolchain on a Core 2 Duo machine with 4 GB RAM and running on the hArtes hardware platform. The synthesis box is responsible for compiling the source targetting each processing element and then linking them to form one binary. The hArmonic tool generates the source-code for each processing element in the system based on the mapping selection process. In particular, each source-code generated corresponds to part of the application and hArmonic computes the minimal subset (headers, types, variables and functions) re-

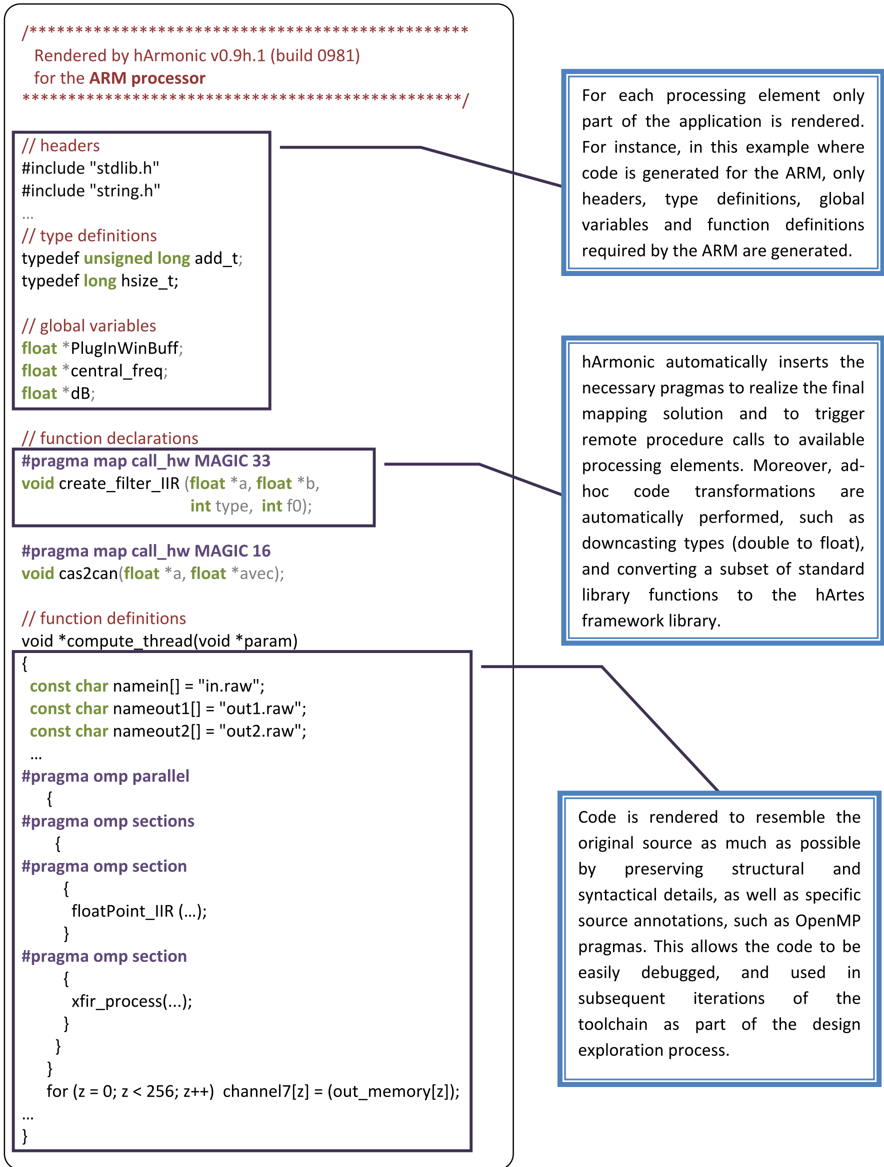


Fig. 5.8 Main features of the code generated by the hArmonic tool for each backend compiler in the synthesis toolbox. In this figure, we focus on the code generated for the ARM processor

quired to successfully complete the compilation process for each part of the application. Figure 5.8 illustrates the main features of the code generated for the ARM.

Fig. 5.9 PowerServer final setup in the car trunk

5.5 Experimental Results

Starting from a detailed description of the ACIS architecture, firstly a first evaluation of the algorithms functionality will be reported using a PC-based framework, then the final implementation on the hArtes embedded platform will described.

5.5.1 Car Lab Prototype

The selected vehicle for the demonstrator is a Mercedes R320 CDi V6 Sport car, which has been chosen because of space needed by the Carputer hardware and audio system. The car has been equipped with up to 30 loudspeakers and 24 microphones located inside the car according to algorithms development requirements (Fig. 5.9).

In particular ad hoc loudspeaker set has been designed and produced. The resulting audio reproduction system outperforms current production systems in terms of performance and allows also the exploration of a large variety of audio processing algorithms (CarLab requirement). During algorithm exploration, the system can be freely downgraded and reconfigured, switching on and off different audio channels. The system consists of a full three-way stereo subsystem for the first row of seats (6 loudspeakers in 2 sets of woofer, midrange and tweeter), a full three-way stereo subsystem for the second row of seats (other 6 loudspeakers) and a two-way subsystem (4 loudspeakers in 2 sets of mid-woofer, tweeter) for the third row (Fig. 5.10 and Fig. 5.11). One high performance subwoofer, located in the trunk, helps the subsystems to reproduce the lower audio frequencies. In this way, a complete set of loudspeakers able to cover with time coherence the whole audio frequency range is installed near each passenger. A suitable set of 9 specific loudspeakers has been developed for communication applications. It covers the full speech frequency range and has been mounted on the roof-top with almost one loudspeaker very close to each passenger ears, ensuring the transmission of a clean and high intensity sound with a relatively low absolute pressure level thus improving the signal to noise ratio and minimizing the signal energy returning into the microphones. Special effort

1-2	SubWoofer	f < 100 Hz	9- > 15	Midrange	250 Hz < f < 8 kHz
3-4-5-6	Woofer	70 Hz < f < 250 Hz	16- > 21	Tweeter	f > 4 kHz
7-8	Mid-Woofer	100 Hz < f < 4 kHz	22- > 30	Full range	f > 300 Hz

Fig. 5.10 Loudspeakers position

Fig. 5.11 Example of a midrange loudspeaker

has been devoted to contain loudspeaker nonlinear behaviors to avoid undesired effects. At high pressure level the system should remain weakly nonlinear in order to maintain signal quality and allow hArtes applications to perform correctly.

The ACIS system has to control each single loudspeaker channel for implementing high-end audio/video algorithms. Therefore every single loudspeaker has a ded-

Table 5.2 Average execution time of direct and inverse 1024-pts complex FFT for different optimization levels

	C-Code without compiler optimization (reference)		C-Code with compiler optimization		IPP-Code with compiler optimization	
1024-pts D-FFT	444993 cycles	0.1859 ms	173656 cycles	0.0725 ms	17203 cycles	0.0072 ms
Gain	1.00		2.56		25.87	
1024-pts I-FFT	450652 cycles	0.1882 ms	179449 cycles	0.075 ms	16446 cycles	0.0069 ms
Gain	1.00 (reference)		2.51		27.40	

icated high quality automotive power amplifier. Professional ASIO soundcards have been used to manage all the system I/Os. Off-the shelf components have been chosen to minimize the overall development process and focus on the demonstration of hArtes approach for applications development. Weight, size and power consumption considerations have been overcome by performances issues.

5.5.2 PC-Based In-Car Application

As previously stated, a working ACIS system has been implemented to show the capabilities of the hArtes platform. However, in order to develop a proof-of-concept, a two step strategy has been employed: first a Carputer-based CIS has been developed and installed in a test vehicle as a test bench for the in-car multichannel audio system and the in-car audio algorithms, then the final ACIS proof of concept has been designed using the entire hartes platform.

In this part, we give an overview of the hArtes integration process, through the use of NU-Tech software environment tool for demonstrating the different algorithmic solutions on a PC-based reference platform. The selected algorithms have been developed independently using standard ANSI C-language and libraries and split in elementary software components. These components are homogeneous and have been specifically structured for an efficient reusability. Some optimizations have been achieved using specific IPP (Intel Performance Primitives) library, which is well-suited for the realization of a reference real-time demonstrator. Since the developed algorithms mainly rely on FFT and IFFT calculations, we provide some performances values for 1024-points Direct and Inverse Complex Fast Fourier Transforms in Table 5.2.

Enhanced In-Car Listening Experience

Tests have been performed in order to evaluate the performances of the algorithms as NU-Tech plug-in, considering the car environment with the overall audio sys-

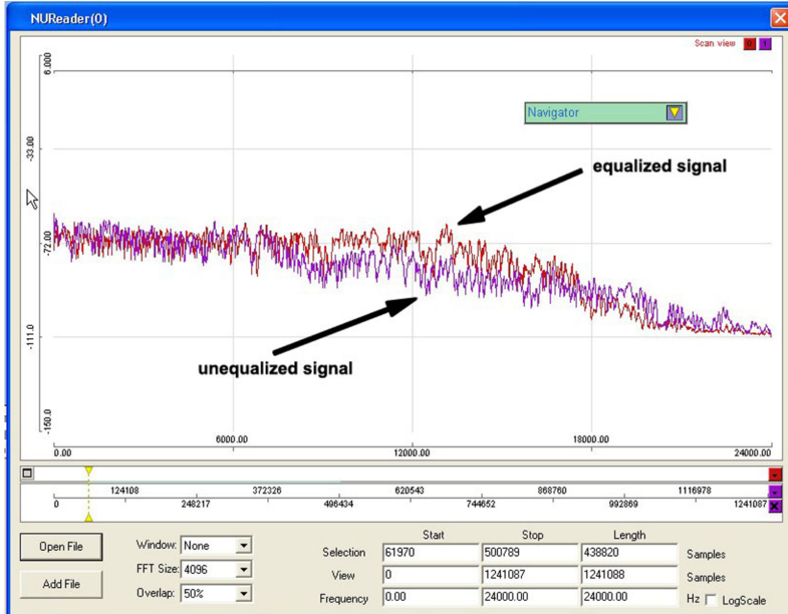


Fig. 5.12 Effect of adaptive stereo equalization on car

tem. Two different validation sessions have been considered: after having verified the algorithms functionalities by synthetic simulations, the proposed algorithms have been tested inside the hArtes CarLab reproducing audio material inside the car cockpit. Some preliminary subjective listening tests have also been conducted to assess the perceived audio quality. For the first part, each algorithm has been separately validated defining an elicited criterion to verify its correct functionality: for the sake of brevity, the entire validation of this part will not be illustrated (for more details see [55]). Concerning in-car validation, only test sessions relative to the adaptive equalizer will be reported, due to its greater significance w.r.t static algorithms. A microphone (AKG 417L) has been positioned on the roof near the driver seat and 12 loudspeakers were driven by a crossover network (Fig. 5.10, Loudspeakers 1, 3, 4, 5, 6, 9, 10, 11, 16, 17, 19, 18, 12, 13). A preliminary recording session considering white noise as input was performed. The target equalization curve was a flat one in order to better underline the equalization capability. As shown in Fig. 5.12, the frequency response of the microphone signal with the equalizer turned on presents a flatter performance and dips and peaks are attenuated w.r.t. the non-equalized signal. The corresponding equalizer at a given time is depicted in Fig. 5.13 together with the measured frequency response prototype: weights at high frequencies are thresholded to prevent excessive gains. Informal listening tests have been conducted by reproducing audio material to evaluate the perceptive effect of the overall system (Fig. 5.1): the listeners were asked to sit into the car at the driver position and to use a suitable SW interface available on the dashboard to modify the algorithm parameters and to change the audio source. Preliminary results seem

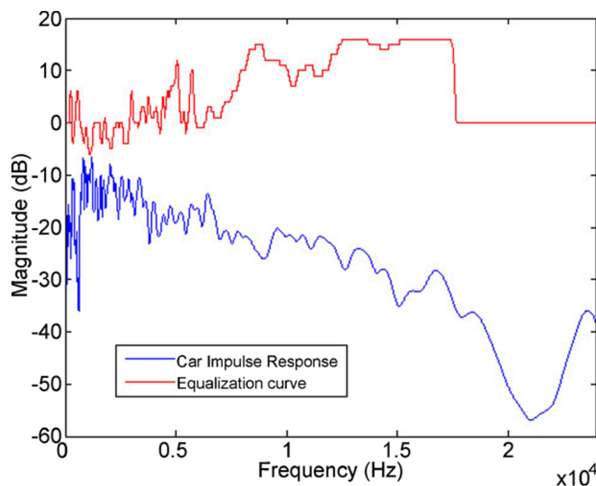


Fig. 5.13 Smoothed car impulse response prototype and equalization weights during adaptive equalization

to confirm the validity of the proposed approach since all subjects involved have reported positive comments and impressions on the global perceived sound image. Further details will be reported in future publications.

Advanced In-Car Communication

Monophonic and stereophonic echo cancellation algorithms have also been integrated in NU-Tech environment. Various experiments have been carried out with some real in-car recordings for the validation of the implemented algorithms. The Signal to Noise Ratio (SNR) improvement obtained with the single-channel speech enhancement is on average between 10 dB and 14 dB. The result of the stereophonic echo cancellation for a stereo music sequence is displayed on Fig. 5.14. The averaged Echo Return Loss Enhancement (ERLE) is between 25 to 30 dB depending on the music sequence, when using a filter length of 512 to 1024 taps at 16 kHz sampling frequency. Since the stereophonic echo cancellation is intended to be used as a pre-processing stage to Automatic Speech Recognition (ASR), no post-filter has been implemented in order to avoid distortion on near-end speech, as it can be seen on Fig. 5.14. For the monophonic echo canceller, a frequency-domain post-filter has been included in order to reduce the residual echo when used for hands-free telephony. The Impulse Response (IR) measured in the car at a microphone located on the mirror, has been considered less than 50 ms, when all the loudspeakers of the car were active, which could be considered as the worst configuration. The result of this criterion using one single frame of speech uttered at the driver's location is depicted on Fig. 5.15. When used in the final integration, the SRP-PHAT criterion is estimated on a reduced set of candidate locations in order to significantly reduce the overall complexity. Assuming that the speaker's location is known in advance, we can limit the search using some hypothesis on the angle and radial ranges. This is illustrated on Fig. 5.16.

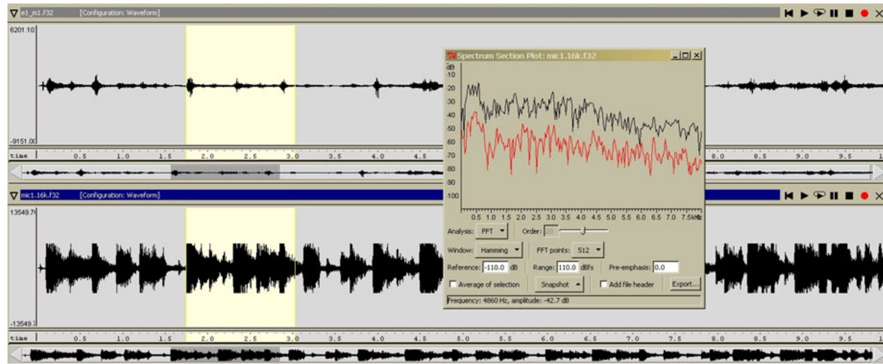


Fig. 5.14 Stereophonic Echo Cancellation of a music sequence

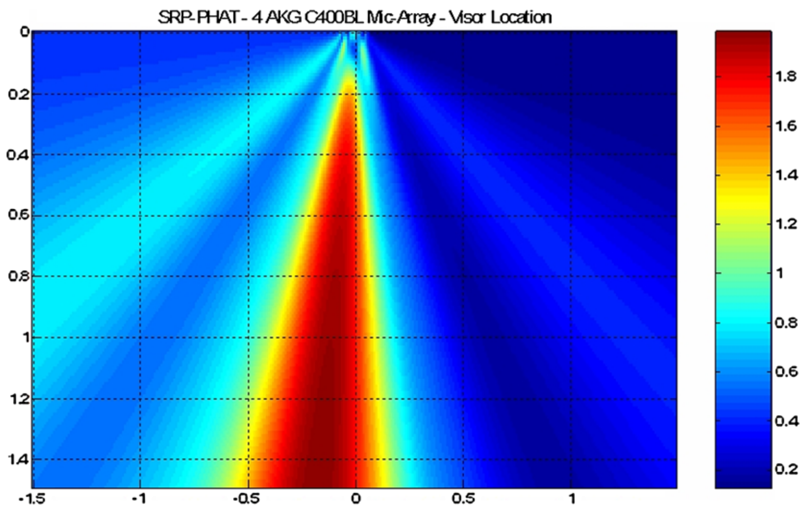


Fig. 5.15 Finally, the multi-channel speech enhancement module has been implemented in NU-Tech, using the SRP-PHAT criterion for on-line speaker localization

In-Car Speaker and Speech Recognition

Three libraries have been developed for the In Car Speaker and Speech recognition application: one for speech parameterization (which is common for speech and speaker recognition applications and aims to extract relevant information from speech material), one for speech recognition and one for speaker authentication. The two first libraries have been developed using C-ANSI language while speaker authentication modules used C++ language. Both languages allow a great integration with the NU-Tech Framework.

Speaker Verification algorithms have been used for user authentication to the CarLab system using a push-to-talk approach (the user has to say his user name and

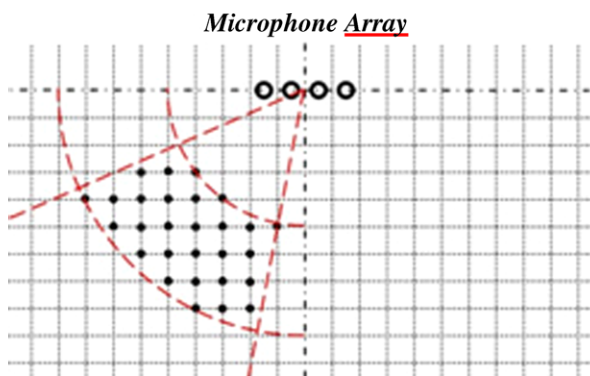


Fig. 5.16 Limited search using region-constrained estimation of SRP-PHAT criterion

password) while Speech Recognition algorithms have been used both for the authentication system (user password recognition) and for the Audio Playback Module where a list of 21 commands have to be recognized during music playback (Play, Stop, Pause, Volume Up, Exit, etc.). Recording sessions inside the car under a greater range of conditions—including real running conditions—have been performed in order to acquire 19 different users (15 males and 4 females) and significantly improve the acoustic model adopted. All algorithms used for speech and speaker recognition systems are tested and evaluated during several evaluation campaigns. For speech recognition, in similar acoustic conditions some tests on French digit recognition allow an accuracy rate higher than 97% (performing around 2300 tests). On a voice command task, with the same kind of audio records, the accuracy rate is around 95% (on 11136 tests). A complete presentation of these results could be found in [34]. In both cases, the audio data are recorded in several cars with A/C turn on/off, radio turn on/off, widows opening or not, etc. Speaker recognition algorithms are also tested and evaluated with participation on the international campaign of speaker recognition: NIST-SRE [45]. The LIA results obtained during this campaign allows to be in the TOP10 of the participants.

5.5.3 Embedded Platform In-Car Application

As described in Sect. 5.5.2, after the first step of developing a Carputer-based prototype for the validation of the system functionalities, the hArtes hardware has been integrated within the already developed system. Figure 5.17 shows how the hArtes platform has been integrated; a specific software is available on a PC for loading code on the platform and implementing the application interface with the system, based on NU-Tech platform [33]. The connection between the hArtes board and the audio system (i.e., professional soundcards) has been realized considering the ADAT protocol, while a TCP-IP connection has been used to manage the algorithm parameters through a graphical interface PC-based.

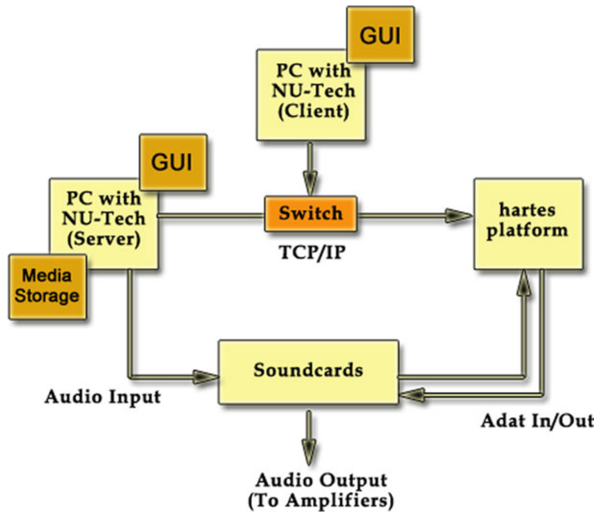


Fig. 5.17 Scheme of the Power Server System considering the hartes platform

Starting from the applications, four different approaches has been evaluated through the toolchain:

- Direct approach, considering a manual implementation on the board just to have a benchmark; two different C projects have been developed, one for ARM processor and one for mAgicV. While ARM is used to control the user interface, mAgicV is used to perform the DSP operations, to configure the audio interface, and to control the input/output audio flow.
- No mapping approach; the application is entirely compiled for the ARM processor, providing a working reference and allowing the evaluation of the basic performances for the application porting on DEB.
- Manual mapping approach (i.e., martes approach), assuming the toolchain with a manual partitioning and mapping of each part of the algorithm considering a specific hardware; in this way, it is possible to obtain the maximum performances of the application with respect to the system.
- Automatic mapping and partitioning (i.e., DSE approach), considering the overall functionality of the toolchain (i.e., automatic task partitioning and mapping). The hArtes tools find automatically the optimal partitioning/mapping, in order to evaluate performances with a very low human effort.

Performances of each algorithm will be considered in terms of workload and memory usage taking into account the Diopsis evaluation board (i.e., DEB) and the entire hArtes platform (i.e., hHp). In the case of workload requirement, it has defined as follows

$$Workload = \frac{T_a}{T_s}\% \tag{5.1}$$

where T_a is the time required to perform the entire algorithm within a single frame while T_s is the time required for realtime application for the same frame size, i.e., considering the frequency sampling of the algorithm and the clock cycles.

For the sake of brevity, just the results of some algorithms of the entire In-Car application will be reported in this section.

Enhanced In-Car Listening Experience

Using the entire toolchain, as described in Sect. 5.4, the whole Audio Enhancement application has been developed and uploaded in the hArtes Platform. A similar configuration of the PC based prototype has been considered; also in this case, the final configuration has defined by two Equalizers, four Crossover Networks of three channels, and twelve Phase Processor to cover four audio front composed by a tweeter, a midrange and a woofer (i.e., twelve total channels). Then a low-pass filtering and a high-pass filtering have been considered respectively for the central channel and for one subwoofer.

Regarding the hardware mapping of each part of the entire algorithm, two different solutions can be presented. Taking into account a manual mapping in the case of the direct implementation and the martes approach, the overall application has been divided on each part of the platform as depicted in Fig. 5.18. Considering the automatic mapping, each part of the entire application is mainly mapped as in the martes approach, except for some vectors allocations and I/O processing that are mapped in the ARM processor and for the IIR filtering that is mapped in the DSP.

For the workload analysis, (5.1) has been considered deriving T_s for a sampling frequency of 48 kHz, a frame size of 256 sampling and a clock cycle of 100 Mhz. At each frame a min and max values of the time required (T_a) is derived: the final parameter is calculated considering a temporal mean of each value. A logarithmic scale has been considered to better underline the results. Obviously, having good performances is equivalent to have a workload less than 100%, in order to have a real time application.

Considering Table 5.3, it is possible to assess the performances of the entire toolchain in terms of workload, where No-mapping approach refers to the entire application running on ARM, overlooking other processors, and DSE refers to the automatic approach. The final results are very similar to those obtained considering a manual implementation both for the martes and for DSE approach (Fig. 5.19). This is due to a good decision mapping applied by the user since the multichannel application lends itself to an easy partitioning. Moreover, considering the No-mapping approach that describes the natural way to write a code, it is clear a great advantage considering the toolchain: this confirms the validity of the methodology.

A great advantage could be found considering also concurrent applications; in this case, taking into account the martes approach, IIR filtering applications performed over the FPGA have been parallelized with respect to the DSP elaboration.

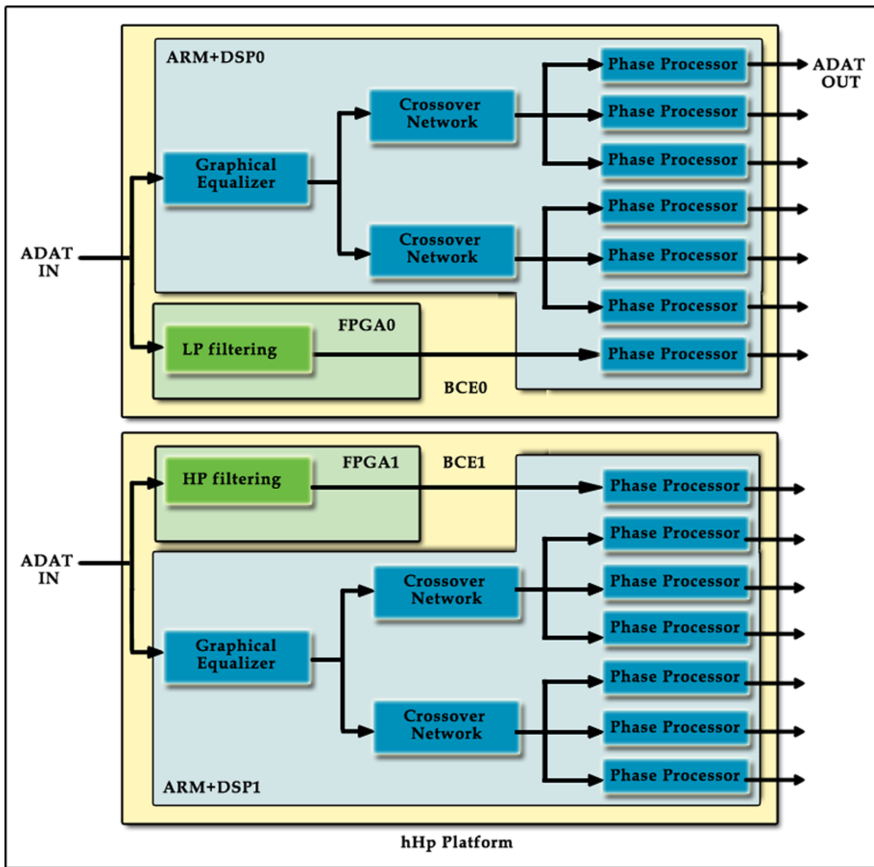


Fig. 5.18 Scheme of the overall Audio Enhancement In-Car application developed for the hHp platform

Table 5.3 Performances of the martes and DSE approach in terms of workload for each algorithm with relation to the direct implementation

Algorithms	Workload			
	Direct implementation	No-mapping	mArtes	DSE
Graphic Equalizer	7.30	873.6	7.52	7.69
Crossover Network	7.48	76.8	8.07	8.30
Phase Processor	5.76	323.5	5.94	6.06
IIR Filter	–	31	16.25	10.7

As we can see from Fig. 5.20, it is clear a great reduction of the overall workload of about 15%. Considering separately the performances of the three processor ARM, DSP and FPGA, each workload and memory requirement can be shown. Figures 5.21(a) and 5.21(b) demonstrate that the DSP performs most intensive opera-

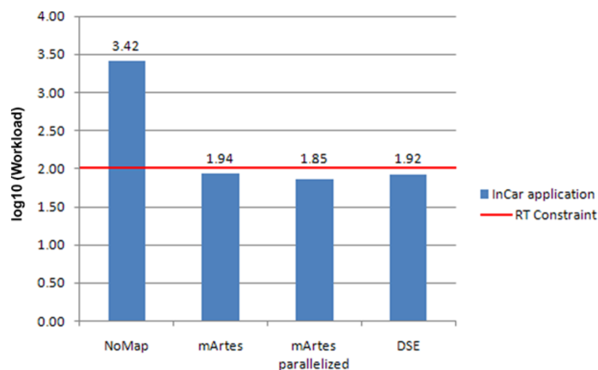


Fig. 5.19 In Car application performances considering different approaches

tion requiring the greater workload while the ARM processor requires more memory due especially to the I/O management. FPGA processor shows less performances since the performed algorithm is less intensive than the others.

Therefore the validity of the toolchain has been assessed and the following remarks can be done comparing it with the manual implementation:

- The manual implementation requires a deep knowledge of the embedded platform (i.e., audio input/output management, memory allocation/communication (Dynamic Memory Acces) and parallelization using thread programming) and the implementation performances are strictly related to the subjective capability. Using the hArtes toolchain, all these aspects can be avoided.
- On the other hands, the toolchain provides an automatic mapping of the application in case of lack of developer experience but it could be considered as a proof where a first assumption of manual mapping is done.

All these aspects imply great advantages in terms of development time: starting from an idea implemented and tested on a PC using ANSI C language, it is possible to realize directly an embedded application representing a first prototype and then reducing the time to market.

Advanced In-Car Communication

The advanced In-Car set of applications has been first integrated in the CarLab demonstrator in order to verify and improve their behavior in real environment. This work has been done through the creation of a set of 26 NU-Tech CNUTSs allowing the execution of the algorithms on any PC computer equipped with low latency audio soundcard. The stationary noise filter, the echo canceller and the beam-former were successfully integrated and easily combined together to provide enhanced In-Car experience. The processing power required for the most complete applications is really important and being able to use them on embedded systems was really challenging. The CarLab multi-core computer was able to handle our most complete solution and the challenge of hArtes project was to be able to execute the exact

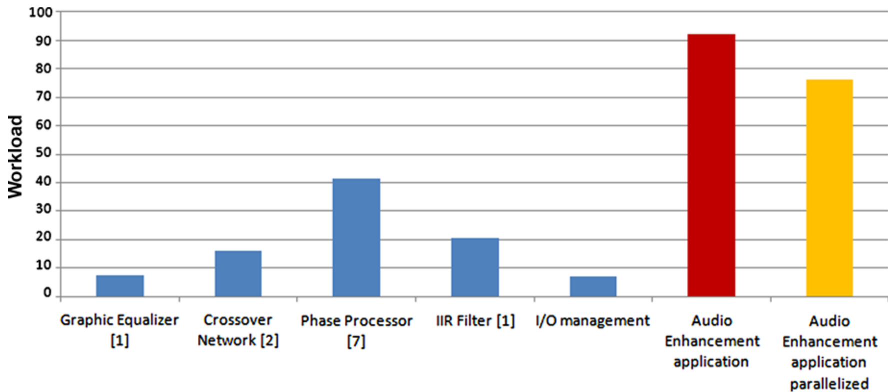


Fig. 5.20 Workload of the overall Audio Enhancement system considering a parallelization of the applications

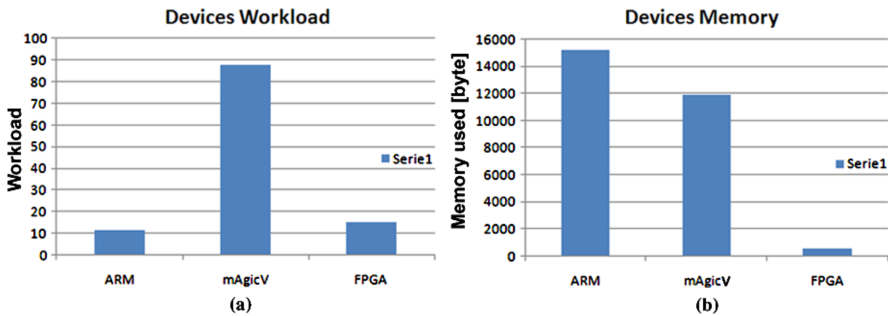


Fig. 5.21 (**a**) Devices workload and (**b**) Devices memory requirements for the Audio Enhancement system in case of parallelized applications

same application on the provided hardware without having to completely redesign the application to fit with the hardware constraints. The hArtes hardware selected for the Advanced In-Car Communication is the DEB that is based on the ATMEL Diopsis, low power, heterogeneous System on Chip providing two different processors: one General Purpose Processor and one Digital Signal Processor. The mAgic DSP is a Very Long Instruction Word class processor which provides a symmetrical architecture optimized for Fourier space calculation in float representation. This architecture is dedicated to achieve complex algebra by providing two data path each composed with a pipeline of 5 operators. The biggest advantage of such processor is its impressive efficiency to calculate Fast Fourier Transform requiring a really low amount of energy. It is one of the architecture on the current DSP market which requires the less number of cycles to process it. The properties of such processor allowed us to select one of our high performance application as a candidate to be executable in real time. We selected the Stationary Noise Filter in the GSM configuration and demonstrated its implementation using all the compilation techniques proposed by the hArtes tool chain. The required processing power of that only ap-

plication is high enough to consume most of the available power on this architecture so we decided to keep it as small as possible so we have a chance to make it working in real time on the hardware. One of the constraint that we wanted to keep all along the project is the portability of the source code, so we decided not to use the hand optimized libraries of the DSP to do signal analysis. This decision allowed us to maintain our ability to modify and test the code on a PC, using for example the high level algorithm exploration tools like the NU-Tech software but made us waste the resources offered by the architecture. This trade off between development easiness and final efficiency is a real concern in THALES business areas, because the development cost of a software represents a significant amount in the cost of a product. The implementation we chose then allowed us to be able to migrate easily part of the software from the PC to the GPP and finally to the DSP. This way of handling software on this kind of heterogeneous hardware is one of the most efficient because we are able to design an algorithm on the PC, quickly validate its usability on the embedded GPP using the full feature operating system it provides and then finally use the DSP to handle the final application and reach the Real-Time performance. The ATMEL Diopsis heterogeneous hardware introduced some limitations in that portability because of two technical issue that we encountered. The ARM926EJ-S is a 32 bit processor used in a little endian mode that do have access to internal and external memories without any kind of restriction in term of data alignment or data size. This great connectivity is unfortunately not true for the mAgic DSP. The mAgic is a 40 bit processor with 16000 words of 40 bit memory available. The first technical issue is the small amount of memory that we are able to use without having to manually use the DMA controller to access the external memory. The second limitation is the data alignment inside of the 40 bit memory. Numbers with float representation do exploit the 40 bits while numbers with integer representation only exploit 32 bits. This means that data transmitted from the GPP to the DSP have to be converted in order to be in the appropriate format and imply that we can not transmit at the same time float and integer data because the hardware doesn't have a way to know which part of the transmitted data is to be converted or not. These two limitations made us completely rewrite the source code of the application in order to split the code into several modules. The low amount of available data memory made us split the calculation and send it to the DSP in several times. The goal was to send a small enough amount of data to the DSP, make it process them and get the result back. Doing that for each part of the algorithm and making the GPP control that process allowed us to be able to overcome the memory size limitation. Furthermore, in order to be able to send the full amount of data required by a module in one request, we needed to send one homogeneous block of data containing numbers all using the same representation (float or int) in order to avoid the data representation conversion problem. This was achieved by using only float in the whole application, even for integers numbers, and by taking care of the floating point operators to make them approximate the values correctly. After identifying these limitations and understood how deep was their impact on the tool chain, we wrote some good practices and showed to all partners an easy way to handle the two problems so they could speed up their developments. In the end we managed to obtain a manual mapping of the

application split into modules. Each modules data are sent one by one to the DSP and processed one after another. One of the first big success of the hArtes tool chain was that it provided a way to select manually which functions of the source code we wanted to be executed on the DSP by the means of a C language extension: the pragmas. The tool chain then processed both GPP and DSP source code in order to automatically take care of the Remote Procedure Call implemented on the DSP. The tool chain automatically replaced the function call on the GPP by a synchronization primitive that do send the data to the DSP, launch the appropriate function and wait for the result. On the DSP side, the processor waits for data and for procedure number to be able to work. Once that first integration phase was done, we took time to refine the algorithm in order to reach the real-time performance. This step required us to redesign some parts of the algorithm in order to reduce the required processing power by replacing some implementations in the algorithms by more efficient ones. The next integration work was to compare the manual mapping application against the automatically mapped one in order to validate that the hArmonic tool was able to find an appropriate repartition source code between GPP and DSP. At the end of the project, the hArmonic tool succeeded to reproduce the exact same mapping without having as an input any kind of information on how the application was architectured or designed. The only resolution of the dependencies between the various parts of the source code made the tool decide a coherent mapping really close to the one we did manually. This implementation allowed us to reach a really high performance, close to the one we obtain using manual mapping. The Ephraim and Mallah Stationary Noise Filter has been implemented using several techniques allowing the comparison of each of them in terms of implementation and performance.

The above graph shows the execution time of 3 versions of the application considering three successfull compilation techniques. The vertical axis is time expressed in sec and each graph represents the statistical repartition of the execution time of the filter. The vertical line shows the min/max values and the box delimits the 25%/75% quartiles. The Real-Time constraint is located at 16000 sec so if the worst case execution times (the highest location of the vertical line) is below it, the application is fast enough. The first implementation we made was using no mapping at all. All the source code of the application was executed on the GPP. This solution was the most easy to obtain because the GPP do provide a complete Linux operating system, with a lot of libraries and debugging tools. It was a necessary step in our integration work because it allowed us to evaluate the processing power of the global architecture. The ARM GPP is not designed to process some floating point calculation and is required to emulate this kind of operations by the mean of a software library thus the resulting performance is really poor. This step allowed us to check that our implementation wasn't more than 10 times slower than the wanted performance so the speedup we obtain using the DSP has a chance to reach the real-time performance. The Fig. 5.22 shows that the GPP mapping of the latest version of the algorithm is 4 times slower than the real time. Another interesting point is the repartition of the execution times. The vertical bar delimits the maximum and the minimum execution time of the algorithm on the GPP. The Linux Operating system is clearly subject to interrupts and the indeterminism of it is really high so the execution time fluctuates

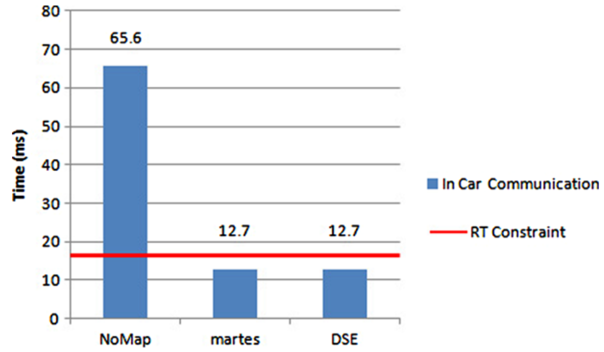


Fig. 5.22 In-Car communication performances in terms of time elapseds

between 58 ms and 88 ms. The second implementation using manual mapping is the most efficient one. The average execution time is 12.7 ms while the required execution time is 16 ms. This final implementation is below the Real-Time performance and allowed us to obtain some nice demonstrators directly using the audio input/output of the hardware. One interesting benefits of this implementation is the repartition of the execution time. The DSP operating system is much more determinist than the GPP one and we obtain a much more reliable behaviour. The third implementation using the automatic mapping proves by its results that the automatic mapping is correct. The tools successfully chose the appropriate mapping and the resulting application is as performant as the manual mapped one. The average execution time is 12.7 ms. The big advantage of hArtes toolchain is to provide an efficient way to explore the mapping that we can put in place on the hardware to make an application running and also to divide automatically the code between the processing elements. This new generation of compilers will probably be the basis of the tools that we will be looking for in order to build software for the heterogeneous hardware and even for the manycore architectures which are already on the market and they will probably reach the low power market very soon.

5.6 Conclusion

Starting from the development of a CIS Carputer-based system to test and validate the multichannel audio system and algorithms inside the cars, a development of the final ACIS proof-of-concept has been carried out. Starting from the CNUTS implementation of the algorithms, different approaches have been realized to test within the hArtes toolchain: a direct implementation to be used as the final target for the toolchain optimization tools; no-mapping approach considering the entire application running on ARM (GPP option), providing a working reference and allowing the evaluation of the basic performances for the application porting on DEB; a manual mapping (mArtes approach) where each function of the algorithms is manually divided among processors as a term of comparison for the toolchain partitioning tools; an automatic mapping/partitioning (DSE approach) to test the functionality

of the entire toolchain (optimization and partitioning). Several tests have been done in order to evaluate the performance achievement considering different toolchain versions and different hardware platforms in terms of memory allocation and workload requirements. Good results were achieved with the toolchain comparing it with the reference examples. Considering the martes approach, it allows having the same results of the direct implementation introducing a good achievement in terms of development time required to realize an embedded application since a deep knowledge of the platform is not required. It is clear that there is a great advantage also considering the DSE approach comparing it with the direct implementation and No-mapping approach respectively; the automatic approach (DSE) is the first step of the implementation suggesting a mapping solution of the target application allowing users to very quickly override some decisions and get better performance. Some adjustments of the toolchain is certainly requested, moreover this approach represents a first step on the development of advanced compilers to easy generate complex solutions. As a matter of fact, the toolchain functionality allows to implement an application without a deep knowledge of the final embedded platform, reducing considerably the development time.

References

1. ALIZE software: http://mistral.univ-avignon.fr/
2. Amand, F., Benesty, J., Gilloire, A., Grenier, Y.: Multichannel acoustic echo cancellation. In: Proceedings of International Workshop on Acoustic, Echo and Noise Control, Jun. 1993
3. Amrane, O.A., Moulines, E., Grenier, Y.: Structure and convergence analysis of the generalised multi-delay adaptive filter. In: Proceedings of EUSIPCO, August 1992
4. Benesty, J., Amand, F., Gilloire, A., Grenier, Y.: Adaptive filtering algorithms for stereophonic acoustic echo cancellation. In: Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (1995)
5. Berthault, F., Glorion, C., Capman, F., Boudy, J., Lockwood, P.: Stereophonic acoustic echo cancellation and application to speech recognition: some experimental results. In: Proceedings of International Workshop on Acoustic, Echo and Noise Control (1997)
6. Bitzer, J., Simmer, K.U., Kammeyer, K.D.: Multi-microphone noise reduction by post-filter and super-directive beam-former. In: Proceedings of International Workshop on Acoustic, Echo and Noise Control, pp. 100–103 (1999)
7. Bonastre, J.F., Morin, P., Junqua, J.C.: Gaussian dynamic warping (GDW) method applied to text-dependent speaker detection and verification. Eurospeech, pp. 2013–2016 (2003)
8. Boudy, J., Capman, F., Lockwood, P.: A globally optimised frequency-domain acoustic echo canceller for adverse environment applications. In: Proceedings of International Workshop on Acoustic, Echo and Noise Control (1995)
9. Buchner, H., Benesty, J., Kellermann, W.: An extended multidelay filter: fast low-delay algorithms for very high-order adaptive systems. In: Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing, April, vol. 5, pp. 385–388 (2003)
10. Buchner, H., Benesty, J., Kellermann, W.: Generalised multi-channel frequency-domain adaptive filtering: efficient realization and application to hands-free speech communication. In: Signal Processing, Mar., vol. 85(3), pp. 549–570 (2005)
11. Buchner, H., Benesty, J., Kellermann, W.: Generalised multi-channel frequency-domain adaptive filtering: efficient realization and application to hands-free speech communication. Signal Process. **85**(3), 549–570 (2005)

12. Capman, F., Boudy, J., Lockwood, P.: Acoustic echo cancellation and noise reduction in the frequency-domain: a global optimization. In: Proceedings of European Signal Processing Conference (EUSIPCO) (1996)
13. Cecchi, S., Palestini, L., Peretti, P., Moretti, E., Piazza, F., Lattanzi, A., Bettarelli, F.: Advanced audio algorithms for a real automotive digital audio system. In: Proc. of the 125th Audio Engineering Society Convention, Oct. 2008
14. Cecchi, S., Primavera, A., Piazza, F., Bettarelli, F., Ciavattini, E., Toppi, R., Coutinho, J.G.F., Luk, W., Pilato, C., Ferrandi, F., Sima, V., Bertels, K.: The hArtes CarLab: a new approach to advanced algorithms development for automotive audio. In: Proc. of the 129th Audio Engineering Society Convention, Nov. 2010
15. Crockett, B., Smithers, M., Benjamin, E.: Next generation automotive research and technologies. In: Proc. of the 120th Audio Engineering Society Convention, May 2006
16. Dentino, M., McCool, J., Widrow, B.: Adaptive filtering in frequency domain. In: Proceedings of the IEEE, December, vol. 66-12 (1978)
17. DiBiase, J.H.: A high accuracy low latency technique for talker localization in reverberant environments using microphone arrays. Ph.D. thesis, Brown University, Rhode Island, May 2000
18. Ephraim, Y., Malah, D.: Speech enhancement using a minimum mean-square error short-time spectral amplitude estimator. IEEE Trans. Acoust. Speech Audio Signal Process. **32**(6), 1109–1121 (1984)
19. Ephraim, Y., Malah, D.: Speech enhancement using a minimum mean-square error log-spectral amplitude estimator. IEEE Trans. Acoust. Speech Audio Signal Process. **33**(2), 443–445 (1985)
20. Farina, A., Ugolotti, E.: Spatial equalization of sound systems in cars. In: Proc. of 15th AES Conference Audio, Acoustics & Small Spaces, Oct. (1998)
21. Ferrara, E.R.: Fast implementation of LMS adaptive filters. In: IEEE transactions on acoustics, speech and signal processing, Aug. 1980
22. Ferreira, A.J.S., Leite, A.: An improved adaptive room equalization in the frequency domain. In: Proc. of the 118th Audio Engineering Society Convention, May 2005
23. Haulick, T.: Signal processing and performance evaluation for in-car cabin communication systems. In: ITU-T Workshop on Standardization in Telecommunication for Motor Vehicles, Nov. 2003
24. Haulick, T.: Speech enhancement methods for car applications, the fully networked car. A Workshop on ICT in Vehicles, ITU-T Geneva, Mar. 2005
25. Haulick, T.: Systems for improvement of the communication in passenger compartment. In: ETSI Workshop on Speech and Noise in Wideband Communication, Sophia Antipolis, 22–23 May 2007
26. House, N.: Aspects of the vehicle listening environment. In: Proc. of the 87th Audio Engineering Society Convention, Oct. 1989
27. Jelinek, F.: Continuous speech recognition by statistical methods. Proc. IEEE **64-4**, 532–556 (1976)
28. Kontro, J., Koski, A., Sjoberg, J., Vaananen, M.: Digital car audio system. IEEE Trans. Consum. Electron. **39**(3), 514–521 (1993)
29. Lang, M., Laakso, T.: Simple and robust method for the design of allpass filters using least-squares phase error criterion. IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process. **41**(1), 40–48 (1994)
30. Larcher, A., Bonastre, J.F., Mason, J.S.D.: Reinforced temporal structure information for embedded utterance-based. Interspeech (2008)
31. Lariviere, J., Goubran, R.: GMDF for noise reduction and echo cancellation. IEEE Signal Process. Lett. **7**(8), 230–232 (2000)
32. Lariviere, J., Goubran, R.: Noise-reduced GMDF for acoustic echo cancellation and speech recognition in mobile environments. In: Vehicular Technology Conference, vol. 6, pp. 2969–2972 (2000)
33. Lattanzi, A., Bettarelli, F., Cecchi, S.: NU-Tech: the entry tool of the hArtes toolchain for algorithms design. In: Proc. of the 124th Audio Engineering Society Convention, May 2008

34. Levy, C.: Modèles acoustiques compacts pour les systèmes embarqués, Ph.D. thesis (2006)
35. Levy, C., Linares, G., Nocera, P., Bonastre, J.F.: Embedded mobile phone digit-recognition. In: Advances for In-Vehicle and Mobile Systems (2007), Chapter 7
36. LIASTOK software webpage: http://lia.univ-avignon.fr/fileadmin/documents/Users/Intranet/chercheurs/linares/index.html
37. Linhard, K., Freudenberger, J.: Passenger in-car communication enhancement. In: Proceedings of European Signal Processing Conference (EUSIPCO) (2004)
38. Lipshitz, S.P., Vanderkooy, J.: A family of linear-phase crossover networks of high slope derived by time delay. J. Audio Eng. Soc., **31**(1–2), 2–20 (1983)
39. Lleida, E., Masgrau, E., Ortega, A.: Acoustic echo control and noise reduction for cabin car communication. In: Proceedings of Eurospeech, Sep., vol. 3, pp. 1585–1588 (2001)
40. Mansour, D., Gray, A.H.: Unconstrained frequency-domain adaptive filter. In: IEEE Transactions on Acoustics, Speech and Signal Processing, Oct. 1982
41. Marro, C., Mahieux, Y., Simmer, K.U.: Analysis of noise reduction and dereverberation techniques based on microphone arrays with post-filtering. IEEE Trans. Speech Audio Process. **6**(3), 240–259 (1998)
42. McCowan, I.A., Bourlard, H.: Microphone array post-filter based on noise field coherence. IEEE Trans. Speech Audio Process. **11**, 709–716 (2003)
43. Meyer, J., Simmer, K.U.: Multi-channel speech enhancement in a car environment using wiener filtering and spectral subtraction. In: Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 21–24 (1997)
44. Moulines, E., Amrane, O.A., Grenier, Y.: The generalised multi-delay adaptive filter: structure and convergence analysis. IEEE Trans. Signal Process. **43**, 14–28 (1995)
45. NIST-SRE website, http://www.nist.gov/speech/tests/sre/2008/index.html
46. NUTS Software Development Kit 2.0 (Rev 1.1): http://www.nu-tech-dsp.com
47. Orfanidis, J.S.: High-order digital parametric equalizer design. J. Audio Eng. Soc. **53**(11), 1026–1046 (2005)
48. Ortega, A., Lleida, E., Masgrau, E.: DSP to improve oral communications inside vehicles. In: Proceedings of European Signal Processing Conference (EUSIPCO) (2002)
49. Ortega, A., Lleida, E., Masgrau, E., Gallego, F.: Cabin car communication system to improve communications inside a car. In: Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing, May, vol. 4, pp. 3836–3839 (2002)
50. Ortega, A., Lleida, E., Masgrau, E.: Residual echo power estimation for speech reinforcement systems in vehicles. In: Proceedings of Eurospeech, Sep. 2003
51. Ortega, A., Lleida, E., Masgrau, E.: Speech reinforcement system for car cabin communications. IEEE Trans. Speech Audio Process. **13**(5), 917–929 (2005)
52. Ortega, A., Lleida, E., Masgrau, E., Buera, L., Miguel, A.: Acoustic feedback cancellation in speech reinforcement systems for vehicles. In: Proceedings of Interspeech (2005)
53. Ortega, A., Lleida, E., Masgrau, E., Buera, L., Miguel, A.: Stability control in a two-channel speech reinforcement system for vehicles. In: Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (2006)
54. Palestini, L., Peretti, P., Cecchi, S., Piazza, F., Lattanzi, A., Bettarelli, F.: Linear phase mixed FIR/IIR crossover networks: design and real-time implementation. In: Proc. of the 123th Audio Engineering Society Convention, Oct. 2007
55. Piazza, F., Cecchi, S., Palestini, L., Peretti, P., Bettarelli, F., Lattanzi, A., Moretti, E., Ciavattini, E.: Demonstrating hArtes project approach through an advanced car information system. In: ISVCS Int. Symposium on Vehicular Computing Systems, Trinity College, Dublin, Ireland, 22–24 Jul. 2008
56. Rangachari, S.: A noise estimation algorithm for highly non-stationary environments. Speech Commun. **48**, 220–231 (2006)
57. Regalia, P., Mitra, S.: Class of magnitude complementarity loudspeaker crossovers. IEEE Trans. Acoust. Speech Signal Process. **35**, 1509–1516 (1987)
58. Schopp, H., Hetzel, H.: A linear phase 512 band graphic equalizer using the fast Fourier transform. In: Proc. of the 96th Audio Engineering Society Convention, Jan. 1994

59. Shivley, R.: Automotive audio design (a tutorial). In: Proc. of the 109th Audio Engineering Society Convention, Sep. 2000
60. Smithers, M.: Improved stereo imaging in automobiles. In: Proc. of the 123rd Audio Engineering Society Convention, Oct. 2007
61. Sommen, P.C.W.: Frequency-domain adaptive filter with efficient window function. In: Proceedings of ICC-86, Toronto (1986)
62. Sondhi, M.M., Morgan, D.R., Hall, J.L.: Stereophonic acoustic echo cancellation—an overview of the fundamental problem. IEEE Signal Process. Lett. **2**(8), 148–151 (1995)
63. Soo, J.-S., Pang, K.K.: Multidelay block frequency domain adaptive filter. In: IEEE Transactions on Acoustics, Speech and Signal Processing, Feb. 1990
64. Viterbi, A.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans. Inf. Theory **2-13**, 260–269 (1967)
65. Widrows, B., Cool, J.M.C., Ball, M.: The complex LMS algorithm. In: Proceedings of the IEEE, Apr., vol. 63-4 (1975)
66. Zelinski, R.: A microphone array with adaptive post-filtering for noise reduction in reverberant rooms. In: Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing, April, pp. 2578–2581 (1988)
67. Zhao, H., Yu, J.: A simple and efficient design of variable fractional delay FIR filters. IEEE Trans. Circuits and Syst. II, Express Briefs **53**(2), 157–160 (2006)

Chapter 6
Extensions of the hArtes Tool Chain

Ferruccio Bettarelli, Emanuele Ciavattini, Ariano Lattanzi, Giovanni Beltrame, Fabrizio Ferrandi, Luca Fossati, Christian Pilato, Donatella Sciuto, Roel J. Meeuws, S. Arash Ostadzadeh, Zubair Nawaz, Yi Lu, Thomas Marconi, Mojtaba Sabeghi, Vlad Mihai Sima, and Kamana Sigdel

In this chapter, we describe functionality which has also been developed in the context of the hArtes project but that were not included in the final release or that are separately released. The development of the tools described here was often initiated after certain limitations of the current toolset were identified. This was the case of the memory analyser QUAD which does a detailed analysis of the memory accesses. Other tools, such as the rSesame tool, were developed and explored in parallel with the hArtes tool chain. This tool assumes a KPN-version of the application and then allows for high level simulation and experimentation with different mappings and partitionings. Finally, *ReSP* was developed to validate the partitioning results before a real implementation was possible.

6.1 Runtime Support for Multi-core Heterogeneous Computing

Even though advanced runtime support for multi-core heterogeneous platforms such as the hArtes platform was not explicitly part of the technical program of the hArtes program, some effort was spent to study in what directions runtime systems should be extended to efficiently manage the available resources. The research was conducted along 2 lines: if we want to provide an abstraction layer such that developers do not need to be aware of the underlying hardware heterogeneity, what functionality should the runtime system support? The second line of research looks at the runtime placement of kernels where area fragmentation and low task rejection are among the objectives. We propose and evaluate a series of on line placement algorithms.

Technology advances shows a movement towards general purpose reconfigurable computers and as a result serving multiple applications which are running concurrently on the same machine is an obvious requirement. To address this requirement,

V.M. Sima (✉)
TU Delft, Delft, The Netherlands
e-mail: v.m.sima@tudelft.nl

K. Bertels (ed.), *Hardware/Software Co-design for Heterogeneous Multi-core Platforms*,
DOI 10.1007/978-94-007-1406-9_6, © Springer Science+Business Media B.V. 2012

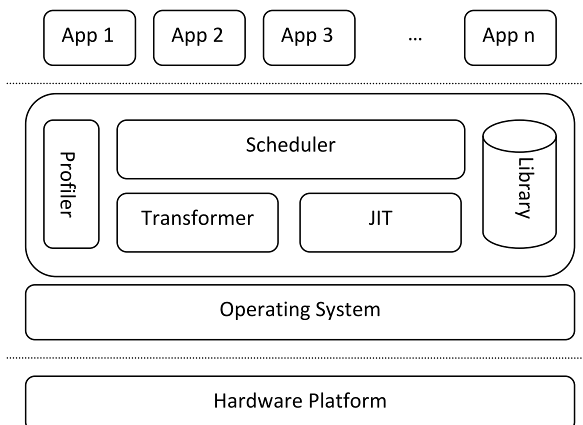


Fig. 6.1 Run-time environment

a run-time environment is needed which is responsible to fully operate the system and address all the conflicting issues between competing applications. Furthermore, the run-time system has to offer the programmers with APIs and system primitives which abstract away the platform dependent details and provides a consistent programming model.

Besides the common services that a general purpose operating systems offers, the run-time system has to manage the reconfigurable resources. It should perform the FPGA area allocation and share it between the applications. Besides, it has to provide APIs required for executing tasks on the FPGA which includes APIs for reconfiguration, execution and memory transfers [35]. To do that, we propose a virtualization layer above the operating system [34] as depicted in Fig. 6.1.

6.1.1 Scheduler

One of the most important issues in multi-tasking reconfigurable computers is the sharing of the limited FPGA resources between competing tasks. Therefore, the scheduler has to decide when and where a specific task has to be executed. It has to deal with conflicting objective such as performance, power consumption or cost. As a result, different scheduling policies can be employed targeting one or more of these objectives.

The data required by the scheduling algorithm can be obtained from a runtime profiler or a design time profiler. A good source of information for the scheduling algorithm is the Configuration Call Graph (CCG). CCG provides the runtime system with information about the execution of the application. Each CCG node represents a task. The edges of the graph represent the dependencies between the configurations within the application. The nodes of the CCG are of three types: operation in-

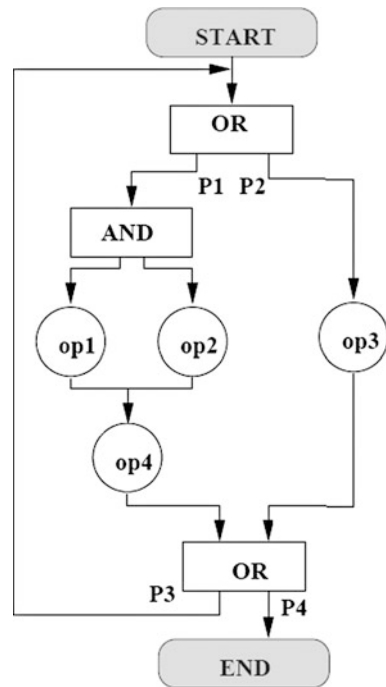


Fig. 6.2 A sample CCG

formation nodes, parallel execution nodes (AND nodes), alternative execution nodes (OR nodes). A sample CCG is shown in Fig. 6.2

It could be a nice attempt to improve the objective by guessing which tasks are likely to be needed soon and better satisfy the objective. Then, either pre-configuring such tasks or if they are already configured, not to replace them with other tasks. Using the CCG, we can extract useful information such as the distance to the next call and the frequency of the calls in future which can be used as the replacement decision parameters [36]. The distance to the next call represents the number of task calls between the current execution point and the next call to the same task in CCG and the frequency of calls in future is the total number of calls from the current execution point to a task in the CCG. Deciding based on the distance to the next call means we want to remove the task which will be used furthest in future. Similarly, considering the frequency of the calls in future as the decision parameter means the replacement candidate is the task which will be used less frequent in the future.

6.1.2 Profiler

The runtime profiler analyzes the code and stores statistics about computational intensity and number of referrals as well as the memory bandwidth being used

of different parts of the code, the purpose of which is to allow the runtime system to identify hotspots, which if implemented in hardware minimize the execution time of the running application [37]. Since the runtime profile has to run in parallel with the actual code, it must be very lightweight and very low overhead otherwise its use is not reasonable. Based on methods of gathering information, Profilers can be categorized into two major groups, Instrumentation-based and Sampling-based.

Instrumentation-based Profilers work by inserting instrumentation code into the application to be profiled. The instrumentation code can be injected either at compile time, link time or run time. The advantage of such Profilers is that they are very accurate and more portable than other kinds of profilers, but on the downside they have relatively large overhead.

Sampling-based profilers periodically take samples of program's Program Counter value. Therefore, the gathered data is a statistical approximation. They are not as accurate as Instrumentation-based Profilers, but have much lower overhead, because no extra instruction is inserted in original code. As a result, they are relatively non-intrusive, and allow the profiled program to run almost at its normal speed.

In Instrumentation-based profilers, the instrumentation code can cause cache misses and processor stalls and hence reduce accuracy of the profiler. Sampling based profilers are relatively immune from such side effects, due to the fact that, no extra code is injected in the original program. Sampling profilers, normally also exploit the hardware performance counters of modern processors, to look for instructions causing more instruction cache or data cache misses, or pipeline stalls.

6.1.3 Transformer

The transformer has to modify the binary by augmenting the calls to the software version of the task with calls to the hardware implementation. Furthermore, the transformer has to safely resolve transfer the parameters required by the hardware implementation to a part of memory which can be easily accessed by the hardware. After the results have been calculated, the appropriate return values have to be safely sent back to the calling thread.

6.1.4 JIT

A just-in-time compiler can be used to compile the tasks for which there is no implementation in the library. The compiler converts binary to bit stream. Any just in time compiler can be used here nevertheless there is no efficient one yet available.

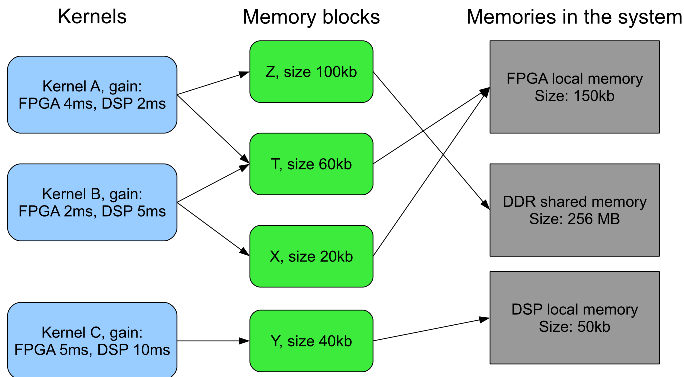


Fig. 6.3 Kernels and memories

6.2 Memory Optimizations in the hArtes Toolchain—AMMA Algorithm

One of the most important issues in a heterogeneous architecture is memory mapping. This is also true for hArtes as, even if the GPP can access all the memory, this is not necessarily true for the PE-s. More than that the memory bandwidth and latency depend on which PE is trying to access a specific memory. The simplest solution to those problems is to transfer the memory to the local memory of the PE, before starting the computation on that PE. The downside of this solution is that each kernel call will introduce a memory transfer, which might result in a slowdown of the application if the transfer time is comparable to the execution time.

Another, better solution, proposed also in [39], is to allocate the memory directly in the local memory of the PE. The GPP processor is still able to access it efficiently through it's cache, but before the actual kernel execution, the only thing needed is to flush the cache for that specific memory range.

Let's assume, as an example, that we have the kernels and memory blocks depicted in Fig. 6.3. For each kernel we show the time improvement that would be obtained if it would be run on a specific PE without additional memory transfers. For each block we give the size of that block in kilobytes. For example if blocks Z and T would be allocated to the FPGA local memory, kernel A would be run on the FPGA and the total execution time of the application would be 4 ms less. But if just kernel Z would be allocated on the FPGA local memory, a transfer of memory block T would be necessary, and the improvement of 4 ms would not be obtained.

The problem that we address with the AMMA allocation algorithm is the following. Given a list of kernels, the gain that could be obtained if run on a specific PE and the list of the memory blocks used by those kernels, decide where to allocate the memory, so that the maximum gain is obtained. For our example the optimal solution is to allocate block Z to the main DDR, blocks T and X to the FPGA local memory and block Y to the DSP memory. The gain obtained would be 12 ms. This solution is specific to the memory sizes shown in the figure.

Doing the analysis of usage of memory blocks is not trivial at compile time. As applications grow more complex, this can be hard even with intimate knowledge of the application. So, we propose a method that does this analysis at runtime, by using two modules: a tracking module and an execution module.

In the following sections we will describe each of the above modules in detail.

6.2.1 The Tracking Module

The purpose of the tracking module is to keep a list of allocated memory blocks. Besides the normal heap, where new memory blocks are tracked by the runtime heap allocator, this list includes stack variables and global variables. Using the information obtained by the tracking module, it is possible at later times, to allocate the blocks in different memories, based on the gains that would be obtained by eliminating the transfers.

For each type of block different mechanism are used. Global variables are changed to pointers, which are initialized at the start of program execution. For stack, functions are added to keep a list of memory blocks used by the stack of functions. In this way, when a certain memory address is used by a kernel, it can be traced to which function it belongs to. This instrumentation is done only when necessary, i.e. only when the address of the stack variables are used as parameters to other functions.

6.2.2 The Execution Module

This module has the role of updating the gains that would be obtained by running a specific kernel on a certain PE. The total gain depends on the speedup for each PE and on the number of execution of that kernel. It is also possible to compute the gain based on the value of parameters, but this wasn't necessary for the kernels observed in the applications used in the hArtes project.

6.2.3 The Allocation Algorithm

After having the list of memory blocks, how are they used by the kernels, and what gains can be expected from each kernel, it has to be decided to which memory we allocate each kernel. We propose 2 algorithms to solve this problem.

AMMA Algorithm The main idea of the algorithm is to give a score to each memory block, based on the gain of the kernels that use it. Then we sort the blocks based on these scores and we allocate the blocks in the memory for which the score is the highest. For our examples the scores are presented in Table 6.1.

Table 6.1 Memory block scores (in ms)

	FPGA	DSP
Z	4	2
T	6	7
X	2	5
Y	5	10

Fig. 6.4 AMMA postcode

```
p number of local memories
a[m] array of memory allocations
k[n] array of kernels
score[m*p] = computescores(a,k)
sort(score,descending)

alloc[p] = 0
for i = 0 to m*p
  j = score[i].memory
  if(a[score[j].id].allocated == false
   and alloc[j] + score[j].size < size[j])
    a[score[j].id].allocated = j
    alloc[j] + = score[j].size
```

By applying the algorithm the first block allocated will be Y (allocated to DSP memory). After that, T will have the highest score so will be allocated to FPGA memory and in the end X will also be allocated to FPGA memory.

The pseudo-code of this algorithm is given in Fig. 6.4.

AMMAe Algorithm There are 2 problems with AMMA algorithm:

- It doesn't take into account that for the gain to be obtained, all the memory blocks used by a kernel have to be allocated in the same memory.
- It will allocate more blocks, even if not all the blocks needed for one kernel will fit in the remaining memory.

To solve this 2 issues we propose another formula to compute the gain for each block and memory. The formula is based on the gain that would be obtained by one kernel, divided by the amount of the memory still needed by that kernel. The effect of this is that after one memory block is allocated to a memory, the other blocks in the same set have a grater score and will be allocated sooner to that block.

6.2.4 Results

The algorithm was applied on the x264 application. Two kernels—satd_wxh and sad—ere identified which used a number of memory blocks in multiple combinations. The sizes of memory blocks was between 256 bytes and 49 kbytes. The improvement obtained by applying the AMMA algorithm was of 14%. For an infinite amount of BRAM the maximum possible improvement would have been of 18%.

The overhead introduced by the tracking and execution module are summarized in Table 6.2.

Table 6.2 Execution overheads of tracking the memory

Video	1	2	3	4	5	6
Dynamic memory	0.38%	0.10%	0.10%	0.10%	0.10%	0.24%
Stack memory	0.36%	0.58%	0.33%	1.07%	0.10%	0.52%

6.3 Hardware Support for Concurrent Execution

In this section, we will describe a set of hardware designs which enhances the original Molen architecture [40] with concurrent execution capability. In addition, we will explore the online FPGA resource management question and several related placement and scheduling algorithms will be depicted.

6.3.1 Using Xilinx APU

In order to support multi-threads in the Linux operating system, the PowerPC (**PPC**) should not to be stalled by any instruction. In the APU version of the Molen architecture, we added two user-defined instructions (UDI0FCMCR_IMM_IMM_IMM and UDI2FCMCR_GPR_IMM_IMM) which are used to start CCUs and check computation results respectively. In addition, the hardware logic which guarantees the non-stalled execution of PPC has been designed. Furthermore, in order to directly use CCUs created by DWARV [46], a CCU wrapper was made in order to adapt the original CCU to the APU interface standard. The mechanism of the APU version consists of the following steps: (a) when the PPC receives the first instruction, the PPC passes it to the APU, then the PPC continues to execute next fetched instructions; (b) the APU decodes this instruction and passes it the CCU wrapper, then the wrapper sends the "start_op" signal to the CCU; (c) every time when the CCU wrapper receives the second instruction, it will immediately return the content of "CCU_status" register to the general purpose register 3 and create a "APU_done" pulse, in this way, the second instruction just stall the PPC for few cycles; (d) if the return value in register 3 is -1, the CCU operation is completed and the results are ready. Otherwise, the CCU is still in processing.

6.3.2 Microcode-Controlled Memory Parallelized Access

In the previous Molen implementation, multiple CCUs run sequentially. In the APU version, the memory access time is share among CCUs. CCUs will be controlled by microcodes and each CCU will occupy the memory interface only when required. The difference for memory access between the original Molen and the APU extension is shown in Fig. 6.5. In Fig. 6.5(a), three CCUs access memories sequentially,

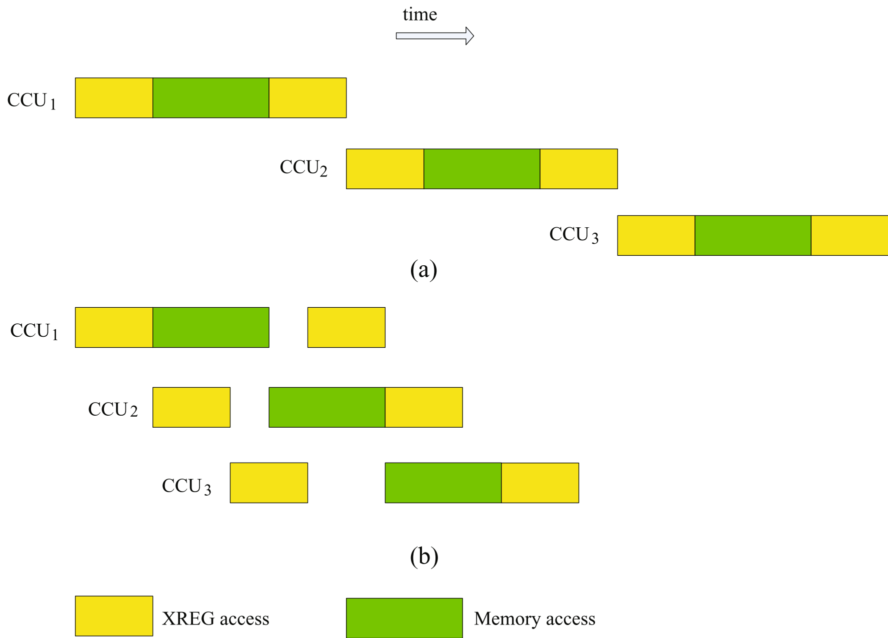


Fig. 6.5 Memory access

Table 6.3 Status of CCU and the microcode

CCU		Microcode	
Request XREG read	001	Start	111
Request memory process	010	XREG read	100
Request XREG write	011	Memory read	010
		XREG write	101
		Stall	000

which reflects the situation in the previous Molen implementation. In Fig. 6.5(b), each CCU only occupies the memory interface when needed. Each CCU has a fixed priority and it needs to request the sequencer for the memory access. The sequencer will stall the CCU or allow its request according to the availability of the required memory and the CCU's priority. Table 6.3 lists the status of a CCU and the microcode used to control it.

6.3.3 Interrupt Support for Molen

The Molen architecture is extended to support interrupt as shown in Fig. 6.6. This interrupt mechanism consists of the following four steps. Firstly, if an interrupt sig-

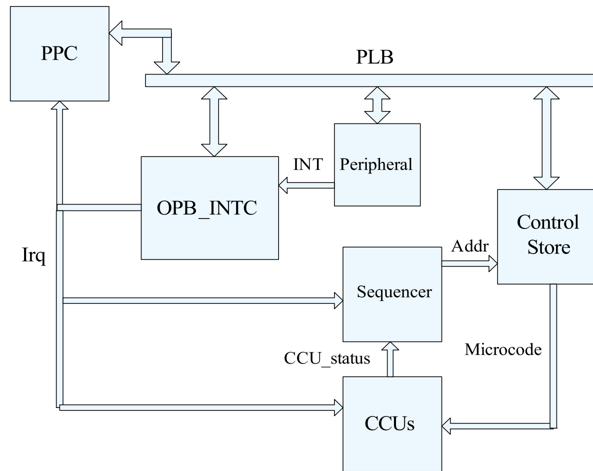


Fig. 6.6 The Molen structure with interrupt support

nal is triggered by a peripheral, the "OPB_INTC" sets the "Irq" to "high". Secondly, the running CCUs receive the "Irq" signal, then each CCU is stalled and the current microcode address is stored. The microcode will be reloaded after the interrupt is handled. Status and temporal data of each CCU will also be stored. Meanwhile, the PPC receives the "Irq" and call the interrupt responding functions. Thirdly, after handling the interrupt, the "Irq" is set back to "low", then the CCUs will be recovered to the interrupted point. Finally, interrupted CCUs continue their executions.

6.3.4 Runtime Partial Reconfiguration Support for Molen

Reconfigurable computing is a computing paradigm to gain both the flexibility of a general purpose computer and the performance of application specific integrated circuits (ASICs) by processing part of the application using flexible high speed reconfigurable fabrics like FPGAs. However, the major drawback of reconfigurable computing is the configuration overhead since the functionality of the reconfigurable fabric has to be changed frequently. To tackle this overhead, partial reconfigurable capability has been added to currently modern Field Programmable Gate Arrays (FPGAs), which means not the whole FPGA is reconfigured, but only those parts that have to be changed. To exploit this capability, we need to access the reconfiguration port of FPGAs (in Xilinx FPGA is the Internal Configuration Access Port (ICAP) [44, 45]).

In [40], Stamatis et al. introduced the MOLEN $\rho\mu$-coded processor which allow entire pieces of code or their combination to execute in a reconfigurable manner with additional three new instructions. The reconfiguration and execution of the reconfigurable hardware are performed by $\rho\mu$-code. The MOLEN polymorphic processor [40] is a Custom Computing Machine (CCM) based on the co-processor architectural paradigm. It resolves some shortcomings of reconfigurable processors,

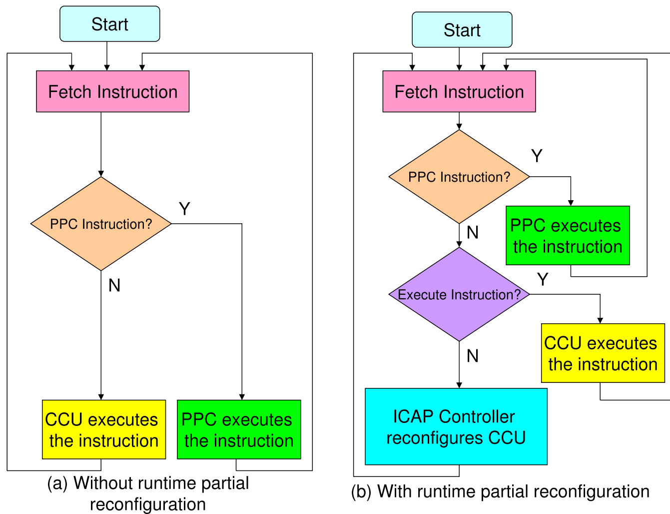


Fig. 6.7 Simple general operation of Molen

such as: opcode space explosion, modularity and compatibility problems. More detail can be seen in [13, 40]. This idea was prototyped in [13] and supported with special compiler (Molen compiler) [25]. Many applications have already exploited this idea, such as: multimedia [42], cryptography [4], bioinformatics [8], Java machine [5], etc. However many applications have exploited the Molen prototype, the runtime configuration feature is not yet supported by the current prototype [10]. In this section, we present an extension of the Molen polymorphic processor prototype utilizing a Virtex 4 FPGA of Xilinx [44] to support runtime partial reconfiguration.

Molen prototype without runtime partial reconfiguration capability has two main processing units as illustrated by flowchart in Fig. 6.7(a). The software instructions (PPC instructions) are executed by the PowerPC (PPC), while the hardware instructions (CCU instructions) are operated by the Custom Computing Unit (CCU). The extension of Molen prototype [13] with runtime partial reconfiguration capability is presented by flowchart in Fig. 6.7(b). Accessing ICAP is a mandatory to get benefits of runtime partial reconfiguration, therefore we need to add ICAP controller to the Molen prototype. The hardware instructions for execution are performed by the CCU, while the runtime partial reconfiguration instructions are executed by the ICAP controller.

An example of runtime partial reconfiguration instruction ("set A3" instruction) is illustrated in Fig. 6.8. The bitstream (BS) of partial reconfiguration is stored in repository. In this example, there are three BSs (BS1 to BS3) in the repository as illustrated in Fig. 6.8(a). The first address in each BS contains its BS length. In this example, there are two partially reconfigurable regions (PRR) in the reconfiguration fabric. The initial condition of reconfigurable fabric before "set A3" instruction contains BS1 (in PRR A) and BS2 (in PRR B) as exemplified in Fig. 6.8(b). Suppose we want to reconfigure PRR A to become BS3, in this case we have to send "set A3"

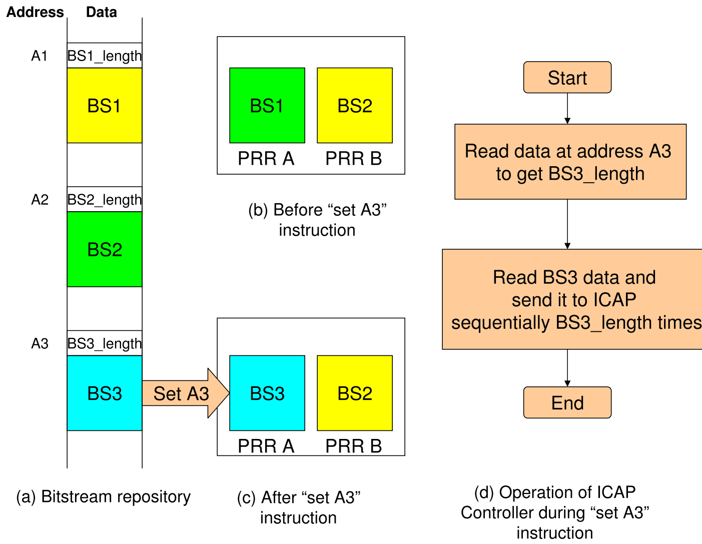


Fig. 6.8 An example of runtime partial reconfiguration instructions

instruction to the Molen prototype. The ICAP controller executes the instruction by transferring BS3 from repository to the reconfigurable fabric during runtime after reading the BS length (BS3 length) as illustrated in Fig. 6.8(c) and (d).

The detail description of the ICAP controller integration to Molen is presented in Fig. 6.9. After fetching the instruction from repository/memory, the fetch unit sends the instruction to the Arbiter. The Arbiter decides to whom this coming instruction will be delivered. If the instruction is the software instruction, the Arbiter sends the instruction to the PPC for execution. If the instruction is the hardware instruction for execution, the Arbiter sends start signal to $\rho\mu$-code unit. The start signal and the content of microinstruction register (MIR) are directed to the CCU interface and then to the chosen CCU based on MIR content. After receiving start signal from the CCU interface, the CCU executes the instruction. The end op signal from the CCU is used to tell the CCU interface that the instruction has been processed by the CCU. This end op instruction is directed to $\rho\mu$-code unit and then finally to the Arbiter to continue processing next instruction.

If the instruction is the hardware instruction for reconfiguration, the Arbiter sends start signal and BS start address to the ICAP controller. After receiving these signals, the ICAP controller executes the instruction by transferring the BS from memory to FPGA internal SRAM cells through ICAP. The end op signal from the ICAP controller is directed to the Arbiter after the reconfiguration and then the new instruction will be prepared again by the fetch unit.

To implement the prototype, we use Xilinx ISE 8.2.01i PR 5 tools (for synthesis and simulation), Xilinx Platform Studio 8.2.01i PR 5 (for designing the Molen core) and Xilinx EAPR tools (for implementation) [43]. After designing the prototype we have simulated it to make sure that the design works properly. The implementation was downloaded and verified using target chip XC4VFX100-10FF1517 on the

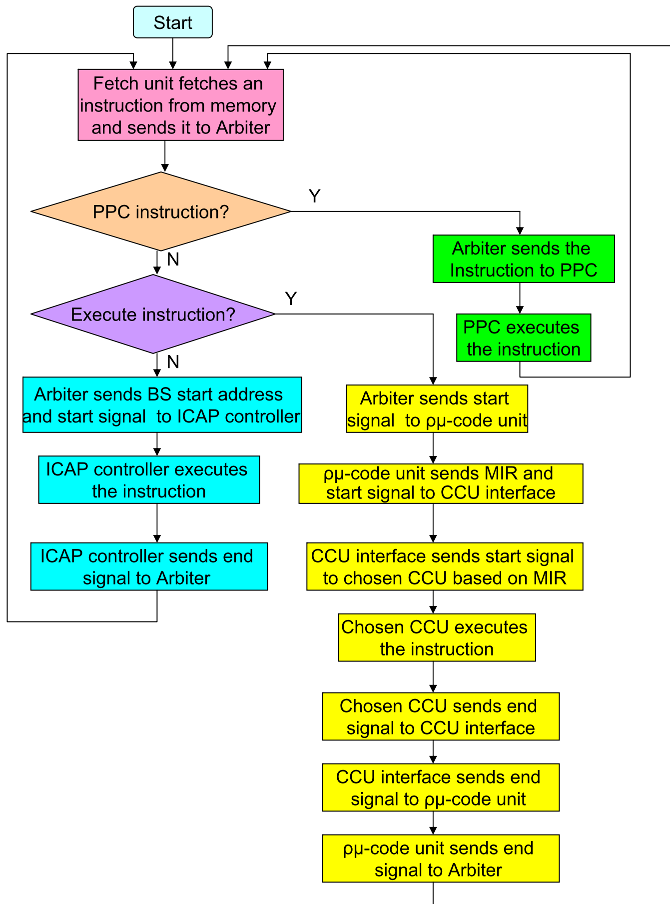


Fig. 6.9 Integration of the ICAP controller to Molen

hArtes board. The set instruction was tested using several runtime partial bitstreams that already stored in the repository. Based on the simulation and real hardware verification on the board, the prototype can support runtime partial reconfiguration correctly. The ICAP controller consumes 136 slices (0.3% of the target chip) and works at up to 188 MHz clock rate.

6.3.5 Online Scheduling and Placement Algorithms

The partial reconfigurability of FPGAs makes systems able to accelerate various applications dynamically as well as easily adapt to changing operating conditions. One of the key challenges is to provide the appropriate runtime management of the configurable resources which is referred to as reconfigurable hardware operating

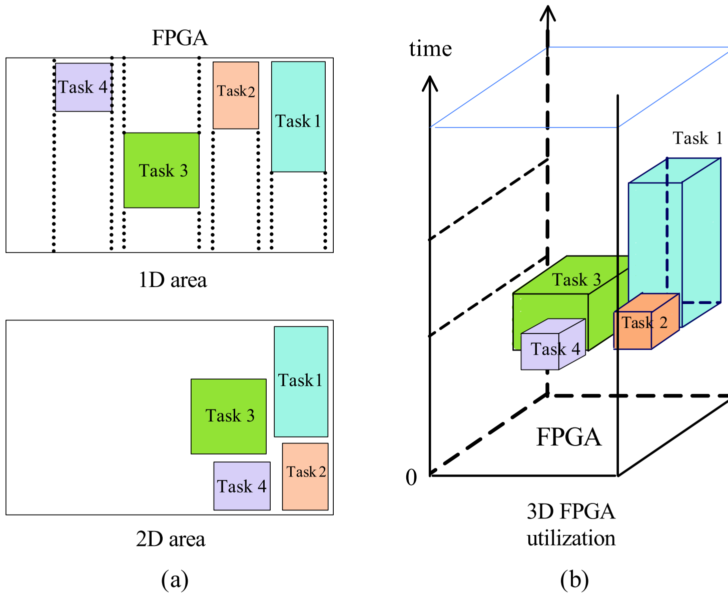


Fig. 6.10 Models

system (RHOS). The RHOS manages all configurable resources to optimize the resource usage and achieve speedup of overall systems. The reconfigurable resources are assigned to tasks based on various heuristics. If the assignments are only based on the availabilities of the resources at certain moment, it is usually called online task placement. The heuristics used in online task placement focus on the spatial requirements which usually results in the minimized spatial resource usage (such as configurable and routing resources) on the FPGA at a moment. If the time requirements of tasks are further taken into account, it is categorized into online task scheduling which mainly focus on minimizing the complete application execution time. For online task placement algorithms, FPGA area models are used to represent the spatial resources. According to the FPGA manufactory technology, the 1D area partitioning model (1D area) and 2D area partitioning model (2D area) are mostly used as shown in Fig. 6.10(a).

For online task scheduling algorithms, the spatial FPGA area models used for the task placement cannot directly represent the utilization of resources in the temporal axis. Therefore, the 3D FPGA utilization model as shown in Fig. 6.10(b) is applied and each task is assigned to a cube where the horizontal surface represents the spatial requirements and the vertical direction stands for the temporal requirements.

Immediate Fit (IF) [17]

The fixed FPGA partitioning model brings fast task allocation but with large resource wastage. On the contrary, the flexible one uses configurable resources more

efficient but requires longer time to find the suitable resources. The IF takes the advantages of both models and avoids their disadvantages. In IF, the FPGA area is pre-partitioned according to applications' requirements and a set of resource redistribution operations is applied during applications' execution. As a result, the IF achieves fast task allocation time as well as less resource wastage. In addition, the redistribution operations bring IF the ability to automatically adapt to applications' requirements at runtime.

PQM-IC [15]

In previously proposed online task placement algorithms, the performance is usually measured by resource wastage and rejection rate separately. However, none of them can reflect the algorithms' efficiency accurately. The PQM-IC takes into account the two factors as well as the rejected tasks themselves, thereby measuring the algorithms more subjectively. In the PQM-IC, in total six equations are provided and the results can be mapped on the complex-number plain for further analysis.

Flow Scanning (FS) [16]

The FS algorithm aims to find the complete set of maximum free rectangles on the FPGA. Previously proposed algorithms model an FPGA as a 2D matrix. Each cell in the matrix has meaningful information which is processed in order to find all maximum free rectangles. This type of algorithms introduces the extra information as well as longer processing time. For the same purpose, the FS only needs to process the allocations of already placed tasks, which results in shorter algorithm's execution time compared to others.

Communication Aware Scheduling Algorithm (CASA) [18, 19]

Thanks to IF and FS algorithms, our proposed scheduling algorithm takes into account not only tasks' spatial and temporal requirements of configurable resources, but also the communication issue as well as single configuration port limitation. In addition, the algorithm further support application-driven heuristics and the task reuse approach.

Intelligent Merging (IM) [22]

IM algorithm consists of three techniques (MON, PM, DC) and one strategy (CBP). To reduce algorithm execution time, IM is equipped with Merging only if needed (MON) technique that allows IM to merge blocks of empty area only if there is no available block for the incoming task. To terminate merging process earlier, IM is

armed with Partial Merging (PM) technique to give it an ability to merge only a subset of the available blocks. To further reduce algorithm execution time, IM can directly combine blocks using its Direct Combine (DC) technique. To increase the placement quality, Combine Before Placing (CBP) strategy always directly combines NERs to form a bigger NER before placing a task when possible.

Intelligent Stuffing (IS) [21]

Adding alignment status to free space segments is the main idea of IS algorithm. This status helps IS to make better decisions on task placement position to facilitate placing tasks earlier. IS is designed with two operating modes: speed and quality. In the speed mode, the algorithm execution time is more important than the quality of the solution; while the quality mode is designed for higher utilization of the resources.

Quad-Corner (QC) [23]

The existing strategies tend to place tasks concentrating on one corner and (or) split free area into many small fragments. These can lead to the undesirable situation that tasks cannot be allocated even if there would be sufficient free area available. As a consequence, the reconfigurable device is not well utilized. Furthermore, rejected tasks has to wait for execution or to be executed by the host processor, hence the application will be slowed down. To tackle these problems, QC spreads hardware tasks close to the four corners of the FPGA. Since QC spread tasks close to the corners, it can make a larger free area in the middle of the device for allocating future tasks. As a result, both the reconfigurable device utilization and the system performance will be increased.

3D Compaction (3DC) [24]

Inefficient algorithms can place tasks in positions where can block future tasks to be scheduled earlier, termed "blocking-effect". To tackle this effect, 3DC is equipped with 3D total contiguous surface (3DTCS) heuristic. The 3DTCS is the sum of all surfaces of an arriving task that is contacted with the surfaces of other scheduled tasks. It has two contiguous surfaces: horizontal and vertical surfaces. The horizontal surfaces is designed for avoiding "blocking-effect"; while the vertical surfaces is built for better packing tasks in time and space.

6.4 Extension of DWARV: Exploiting Parallelism by Using Recursive Variable Expansion

Loops are an important source of performance improvement, for which there exist a large number of compiler optimizations. A major performance can be achieved

through *loop parallelization*. Reconfigurable systems beat general purpose processor (GPP) despite the higher frequency of GPP by utilizing extensive parallelism inherent in reconfigurable systems. A major constraint to achieve extensive parallelism is due to the dependencies. There are many loop optimization techniques which reorganize the program to align the flow of the program along the direction of the data dependencies to achieve close to dataflow performance. *Recursive Variable Expansion* (RVE) is a technique which outperforms the dataflow. It removes the loop carried data dependencies among the various statements of the program to execute every statement in parallel. This can achieve maximum parallelism assuming we have unlimited hardware resources on a FPGA. Although area on FPGA is increasing with Moore's law, still it is impractical to assume that we have unlimited available resources. We have applied RVE in two different ways to achieve parallelism beyond dataflow and also keeping the memory bandwidth and area requirement within limits. First, we will introduce RVE and then we will briefly describe the two ways it is applied to different class of problems.

6.4.1 Recursive Variable Expansion

Recursive Variable Expansion (RVE) [26] is a technique which removes all loop carried data dependencies among different statements in a program, thereby making it prone to more parallelism. The basic idea is the following. Let G_i is any statement in some iteration i depending on the outcome of some statement H_j in any iteration j due to some data dependency. Both of the statements can be executed in parallel, if the computation done in H_j is replaced with all the occurrences of the variable in G_i which create the dependency with H_j. This makes G_i independent of H_j. Similarly, computations can be substituted for all the variables which create dependencies in other statements. This process can be repeated recursively till all the output expressions are a function of known values and all data dependencies are removed. These output expressions can be computed efficiently independent of results from any other expression. The success of this technique is based on the number of operators which are associative in nature. This transformation is explained by example in Fig. 6.11. When RVE is applied to code in Fig. 6.11a, it transforms to Fig. 6.11b. Generally, RVE can be applied to a class of problems, which satisfy the following conditions.

1. The bounds of the loops must be known at the compile time.
2. The loops does not have any conditional statement.
3. Data is read at the beginning of a kernel from the memory and written back at the end of the kernel.
4. The indexing of the variables should be a function of surrounding loop iterators and/or constants.

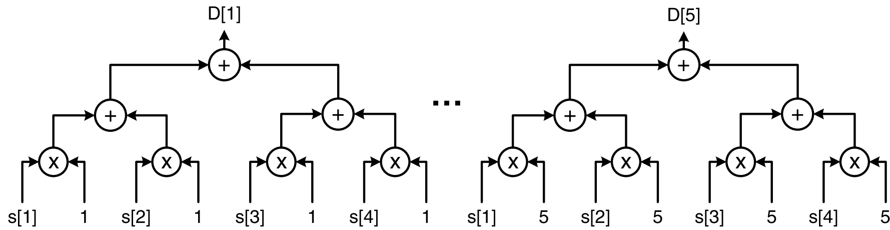
In this way, the whole expanded statement in Example 6.11b can be computed in any order by computing a large number of operations in parallel and efficiently using recursive doubling [12] as shown in Fig. 6.11c.

```
for i=1 to 5
  D[i]=0
  for j=1 to 4
    D[i]=D[i]+s[j]*i
  end for
end for
```

(a) example code

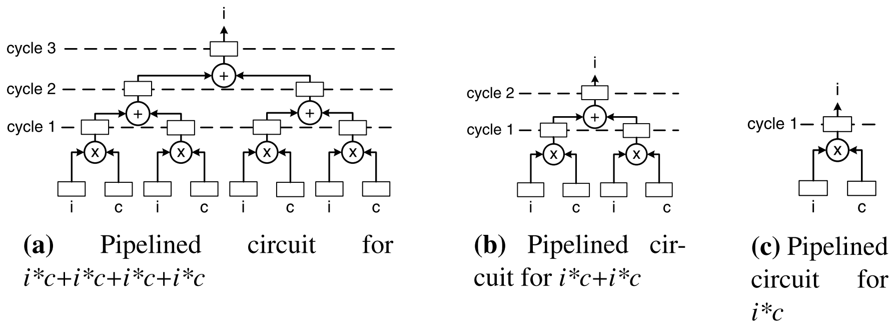
```
D[1]=0
D[1]=D[1]+s[1]*1=s[1]*1
D[1]=D[1]+s[2]*1=s[1]*1+s[2]*1
D[1]=D[1]+s[3]*1=s[1]*1+s[2]*1+s[3]*1
D[1]=D[1]+s[4]*1=s[1]*1+s[2]*1+s[3]*1+s[4]*1
...
D[5]=s[1]*5+s[2]*5+s[3]*5+s[4]*5
```

(b) After RVE



(c) Circuit for example

Fig. 6.11 Simple example



(a) Pipelined circuit for *i*c+i*c+i*c+i*c*

(b) Pipelined circuit for *i*c+i*c*

(c) Pipelined circuit for *i*c*

Fig. 6.12 Generic expression and its pipeline circuit for example in Fig. 6.11

In the next section, we will briefly describe the automatic and flexible pipelining for RVE, which is useful for kernels which does not produce exponential number of terms when RVE is applied. Many image and signal processing application lies in this class. By pipelining, we show that we are able to get performance for DCT comparable to hand optimized code.

Pipelining means that a same circuit is used by various computation in different time, therefore the circuit has to be generic enough to be used by those various computation. If we look at circuits in Fig. 6.11c, it computes all the outputs (D[1], D[2], . . . , D[5]) separately. We can save area and compute all the outputs by just making a circuit for one output and computing the rest of the outputs using the same circuit at different stage in circuit at the same time by pipelining. For that, we need to insert intermediate registers to store the intermediate values of all the stages in progress as shown in Fig. 6.12a, provided it meets the area and mem-

Table 6.4 Time and hardware to compute DCT

	Time (ns)	Speedup	Slices
Pure Software [26]	62300	1	–
Xilinx DCT core [28]	373.8	167	1213
DCT full element [28]	527.9	118	9215
DCT one-third element [28]	723.6	86	2031

ory constraints. However, if the memory or area constraints are not met, we can divide the expression further and further into some smaller repeated equivalent sub-expressions such that when a circuit is to be made for any of those sub-expressions, it satisfies the area and memory constraints. The other smaller candidates are shown in Figs. 6.12b and 6.12c, which can be chosen according to the memory and area requirement.

In [28], we have devised an efficient automatic and flexible pipeline design algorithm for kernels on which RVE is applied. This algorithm select a suitable candidate for pipelining in $O(L^2)$, where L is the size of the expanded expression.

Since many variables are repeated in RVE, there are many redundant operands that can be removed without sacrificing the speed, which further reduces the area requirement.

The pipelining algorithm for RVE is implemented, which automatically finds the optimal candidate for pipelining and feed the variables that need to be transferred after every cycle. We use a Molen prototype implemented on the Xilinx Virtex II pro platform XC2VP30 FPGA, which contains 13696 slices. The automatically generated code is simulated and synthesized on ModelSIM and Xilinx XST of ISE 8.2.022 respectively. We have also implemented *pure software* implementation, which is compiled using GCC 4.2.0. The compiled code are simulated for IBM PowerPC 405 processor immersed into the FPGA fabric, which runs at 250 MHz [26]. The integer implementation of DCT is used as benchmark to demonstrate the effectiveness of our pipelining algorithm. The results of the automatically optimized and different pipeline sizes for the DCT are compared with pure software and the hand optimized and pipelined DCT core[1] provided by Xilinx on the same platform. All the implementations take 8-bit input block elements and output DCT of 9-bit.

Table 6.4 shows the results for different implementation of DCT after synthesis. The *full element* in the experiment refers to expanded expression for one of element in DCT and *one-third element* refers to one third of the expanded expression of the element, which is the next largest repeat.

Table 6.4 shows that *full element* DCT is 118 times faster than the pure software implementation and it also fits on the FPGA. If there is less area available on the FPGA, then the next candidate *one-third element* is 86 times faster than *pure software* implementation. The fastest and more efficient implementation is *Xilinx DCT core*, which is 167 times faster than *pure software* implementation. The reason is

[1] https://secure.xilinx.com/webreg/clickthrough.do?cid=55758.

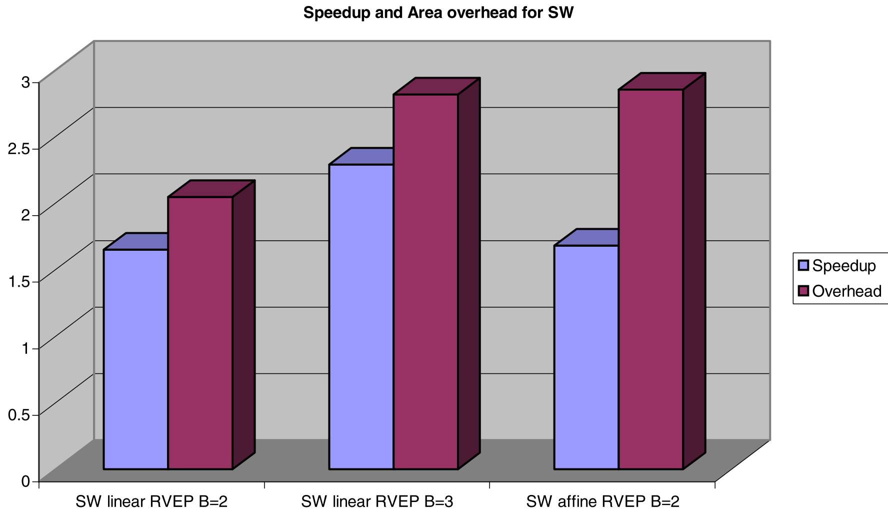


Fig. 6.13 Graph to show speedup and area overhead w.r.t. dataflow

that the *Xilinx DCT* core is hand optimized by knowing the properties of 2D DCT. Our automatic optimization does not take advantage of the knowledge of the properties of 2D DCT. It takes the unoptimized code of 2D DCT, follows some generic steps to apply the RVE and then design a flexible deep pipeline trying to satisfy the area and memory constraints.

6.4.2 RVE for Dynamic Programming Problems

Dynamic programming (DP) is a class of problems, which when applied naively can generate exponential number of terms, which cannot be computed efficiently even if exponential number of resources are used. Therefore, we have restricted RVE and mixed it with dataflow to restrict the growth of terms which achieves better speedup than dataflow approach [27, 29, 30] at the cost of extra area on FPGA. Since dynamic programming problems have overlapping subproblems as its characteristics, expanded terms are futher reduced by eliminating the redundant terms.

As mentioned earlier, the RVE is restricted and applied in smaller blocks we call it blocking factor B. We have termed our best RVE implementation for DP problems as RVEP, which stands for *RVE with precomputation*, which further increases the parallelism. In [30], RVEP is applied to well known Smith-Waterman (SW) problem with varying degree of blocking factor and for SW with different scoring schemes. The speedup and area overhead as compared to current best implementation, which is dataflow approach is shown in Fig. 6.13.

The graph in Fig. 6.13 shows that RVEP accelerates the SW computation more than twice by utilizing more than twice area on the FPGA for blocking factor $B = 3$.

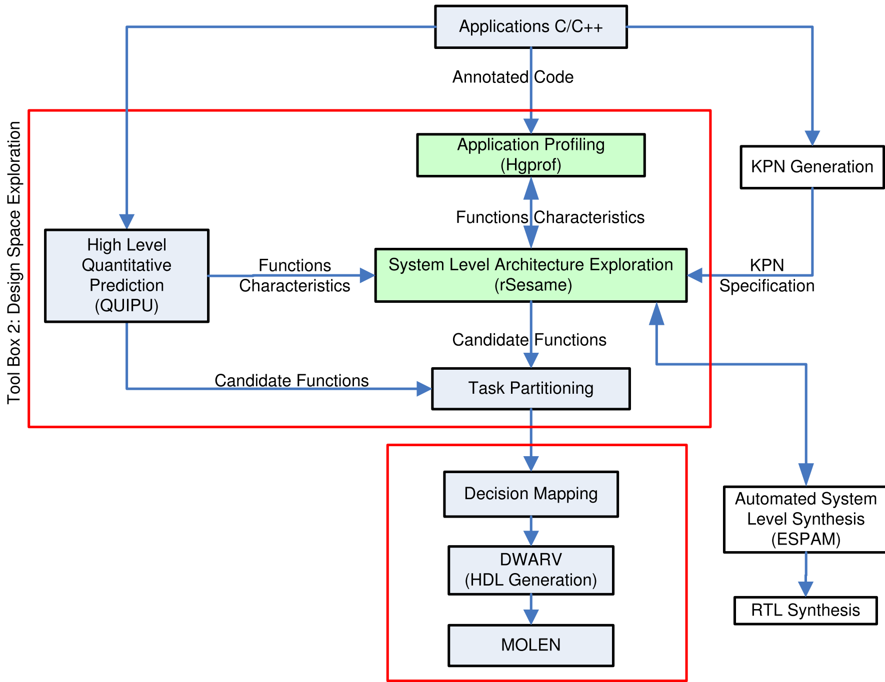


Fig. 6.14 Application profiling and system level architecture exploration in the context of hartes toolchain

It also shows that when the blocking factor for RVEP is increased, it increases the speedup at the cost of more area overhead.

6.5 High Level Profiling and Architectural Exploration in Hartes

6.5.1 Introduction

Profiling and system-level architecture exploration is a part of hArtes design space exploration (DSE) toolbox. The toolbox provides an optimal hardware/software partitioning of the input algorithm for each reconfigurable heterogeneous system considered. A set of profilers, cost estimators and system-level exploration tools employ specific metrics to evaluate a particular mapping of a candidate application on the particular reconfigurable heterogeneous system with respect to performance, hardware complexity, power consumption, etc. Figure 6.14 shows the profiling and system-level architecture exploration within the context of DSE toolbox of hartes toolchain. The applications are profiled using a high level profiling tools. The profiling identifies several characteristics of the tasks such as execution time, number of calls, etc. Based on these characteristics, a set of candidate functions can be

identified, which will be considered for the system level architecture exploration. These tasks are explored further for to evaluate different application to architecture mappings, HW/SW partitioning, and target platform architectures. Such exploration should result in a number of promising candidate system designs which can assist designers in various decision making. This is an iterative process and results with a set of candidate functions for hardware/software partitioning. In the next section of the report, we will describe this process more in detailed.

6.5.2 Application Profiling

Profiling provides necessary information to collect and analyze execution traces of the application program and to point out the performance hot spots, memory hot spot, etc in the application program. In embedded software, determining the relative execution frequency of program is essential for many important optimization techniques such as register allocation, function inlining, instruction scheduling, and hardware/software partitioning. Tools seeking to optimize the performance of the embedded software therefore should focus first on finding the critical code segment of the program. Towards this goal of hardware software partitioning, a partitioning tool should focus first on finding the most critical program hotspot and understanding the execution statistics of these code segments.

6.5.3 System Level Architecture Exploration

In the context of heterogeneous reconfigurable systems, to make early design decisions such as mapping of an application onto reconfigurable hardware, it is essential to perform architecture exploration. System level architectural exploration allows for quick evaluation of different application to architecture mappings, HW/SW partitioning, and target platform architectures. Such exploration should result in a number of promising candidate system designs which can assist designers in various decision making.

Toward this goal, we use Sesame framework [33] as a modeling and simulation framework for system level exploration. For this purpose, we have extended the Sesame framework to support partially dynamic reconfigurable architecture [38]. Sesame can be used to estimate the performance of the system and to make early decisions of various design parameters in order to obtain the most suitable system that satisfies the given constraints. Sesame adheres to a transparent simulation methodology where the concerns of application and architecture modeling are separated via an intermediate mapping layer. The framework is depicted in Fig. 6.15. An application model describes the functional behavior of an application and an architecture model defines the architectural resources and constraints. For application modeling, Sesame uses the Kahn Process Network (KPN) model of computation [11], which consists of functional application code together with annotations Read (R),

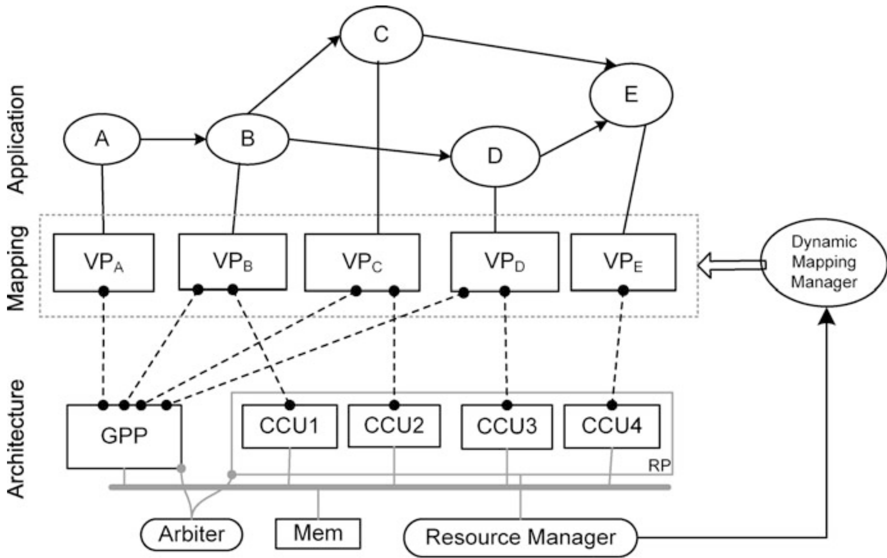


Fig. 6.15 rSesame framework

Write (W) and Execute (EX) that generate events describing the actions of the process. These events are collected into event traces that are mapped, using an intermediate mapping layer, onto an architecture model. The mapping layer consists of Virtual Processors (VPs) whose main purpose of the is to forward the event traces to components in the architecture model according to a user-specified mapping. In the architecture model, the architectural timing consequences of the events are modeled. Interconnection and memory components model the utilization and the contention caused by communication events. To this end, Sesame uses (high-level) architecture model components from the IP component library for creating any kind of architectural model. Sesame allows for quickly evaluating the performance of different application to architecture mappings, HW/SW partitioning, and target platform architectures. The result of the sesame simulation can be given as an input to the task partitioning unit in the hartes tool chain. Furthermore, its output e.g. system level platform description, application-architecture mapping description, and application description can also act as input to any other automated system level synthesis tools such as ESPAM [9] (see Fig. 6.14).

6.6 Memory Access Behavior Analysis

Inspecting the behavior of an application is a crucial step in carrying out effective optimizations for application development on heterogeneous architectures. In particular, the actual pattern of memory accesses of an application reveals significant information about data communications both qualitatively and quantitatively. Therefore, development of support tools is becoming prevalent for application behavior

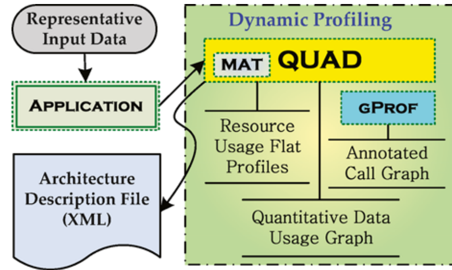


Fig. 6.16 Dynamic profiling famework overview

analysis from different perspectives. Profiling refers to analyzing the behavior of an application to identify the types of performance optimizations that can be applied to an application and/or the target architecture. General profilers such as *gprof* [7] can provide function-level execution statistics for the identification of application hot-spots. However, they do not distinguish between computation time and communication time. As a result, one can not use them to identify potential system bottlenecks regarding problems related to memory access behavior of an application.

6.6.1 Dynamic Profiling

Figure 6.16 depicts an overview of the dynamic profiling framework. Dynamic profiling process focuses on run-time behavior of an application and, therefore, is not as fast as static profiling. Furthermore, dynamic profiling requires representative input data in order to provide relevant measurements. *gprof* is used to identify hot-spots and frequently executed functions, while QUAD is responsible for tracing and revealing the pattern of memory accesses. The annotated call graph not only reveals the caller/callee relationship between functions, but also provides some dynamic information about the execution-time contribution of each function in an application. All the dynamic data extracted from memory accesses are dumped into flat profiles during the execution of an application. The raw data is later processed and the information is presented by Quantitative Data Usage (QDU) graph. The final output will be stored in an architecture description file in XML format that can be easily utilized by other DSE tools in the tool chain. Figure 6.17 depicts part of a sample architecture description file that has been used by QUAD to store bindings information. UMA shows the number of Unique Memory Addresses used for data communication between producers and consumers.

6.6.2 QUAD Tool Overview

QUAD (Quantitative Usage Analysis of Data) is a memory access tracing tool that provides a comprehensive quantitative analysis of the memory access patterns of

Fig. 6.17 Sample profiling elements stored by QUAD in an architecture description file

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ORGANIZATION SYSTEM "architecture.dtd">
<ORGANIZATION>
  <PROFILE>
    <QUAD>
      <BINDING>
        <PRODUCER> x264_validate_parameters </PRODUCER>
        <CONSUMER> x264_macroblock_cache_load </CONSUMER>
        <DATA_TRANSFER> 28800 </DATA_TRANSFER>
        <UMA> 464 </UMA>
      </BINDING>
      ...
    </QUAD>
    ...
  </PROFILE>
  ...
</ORGANIZATION>
```

an application. It primary serves for the detection of actual data dependencies at function-level. Data dependence is calculated in the sense of producer/consumer binding. In particular, QUAD reports which function is consuming the data produced by another function. The exact amount of data transfer and the number of UMA used in the transfer process are revealed. Based on the efficient Memory Access Tracing (MAT) module implemented in QUAD, which tracks every single access (read/write) to a memory location, a variety of statistics related to the memory access behavior of an application can be measured, e.g. the ratio of local to global memory accesses in a particular function call.

In contrast to the existing data dependency detection tools that mainly focus on the discovery of parallelization opportunities, QUAD does not necessarily target parallel application development. Even though QUAD can be employed to spot coarse-grained parallelism opportunities in an application, it practically provides a more general-purpose framework that can be utilized in various application tuning and optimizations for a particular architecture by estimating effective memory access related parameters, e.g. the amount of unique memory addresses used in data communication between two cooperating functions. Moreover, QUAD detects the actual data dependency which involves a higher degree of accuracy compared to the conventional data dependency. In other words, QUAD dynamically records only the real data usage that occurs during application execution. QUAD can be used to estimate how many memory references are executed locally compared to the amount of references that deal with non-local memory. As inefficient use of memory remains a significant challenge for application developers, QUAD can also be utilized to diagnose memory inefficiencies by reporting useful statistics, such as boundaries of memory references within functions, detection of unused data in memory, etc.

6.6.3 QUAD Implementation

QUAD falls into the category of Dynamic Binary Analysis (DBA) tools that analyze an application at the machine code level as it runs. QUAD is implemented using Pin [20], a runtime Dynamic Binary Instrumentation (DBI) framework. Instrumentation is a technique for injecting extra code into an application to observe

its behavior. Using Pin has the benefit of working transparently with unmodified Linux, Windows and MacOS binaries on Intel ARM, IA32, 64-bit x86, and Itanium architectures. Pin's instrumentation transparency also ensures that the original application behavior is preserved. Currently QUAD is only available for Intel-based machines, however, regarding its layered design and implementation, porting to different architectures is feasible as long as there exists a primitive tool set on the target system that can provide the basic memory read/write instrumentation capabilities.

6.7 TLM Simulation of MPSoCs

Multi-Processor Systems on Chip (MPSoCs) are becoming the prevalent design style to achieve short time-to-market for high-performance embedded devices. MPSoCs simplify the verification process and provide flexibility and programmability for post-fabrication reuse. MPSoCs are usually composed of multiple off-the-shelf processing cores, memories, and application specific coprocessors. This leads to larger and larger portions of current applications being implemented in software, and their development dominates the cost and schedule of the whole system. These issues were already present in "classical" embedded systems, but they have been amplified in MPSoCs which are more difficult to model and analyze, due to explicitly concurrent computation. These architectures must be optimally defined to find the best trade-off in terms of the selected figures of merit (e.g. performance) for a given class of applications. This tuning process is called *Design Space Exploration (DSE)*.

In hArtes, to quickly evaluate different design solutions (in particular concerning application partitioning) before implementation, we adopt *ReSP* (*Re*flective *S*imulation *P*latform) [2], a framework allowing the easy generation of high-level models of a hardware system, their simulation, and the analysis of the software execution.

ReSP is a Virtual Platform targeted to the modeling and analysis of MPSoC systems, using a component based design methodology [3], providing primitives to compose complex architectures from basic elements, and built on top of the SystemC [1] and TLM libraries [31]. The framework also exploits Python's *reflective*, *introspective*, and *scripting* capabilities, augmenting SystemC with full observability of the components' internal structure: this enables run-time composition and dynamic management of the architecture under analysis.

6.7.1 Virtual Platform

ReSP builds on SystemC and the OSCI Transaction Level Modeling library [47] to provides a non-intrusive framework to manipulate SystemC IP models. ReSP is based on the concept of reflection, which allows the user to access and modify every

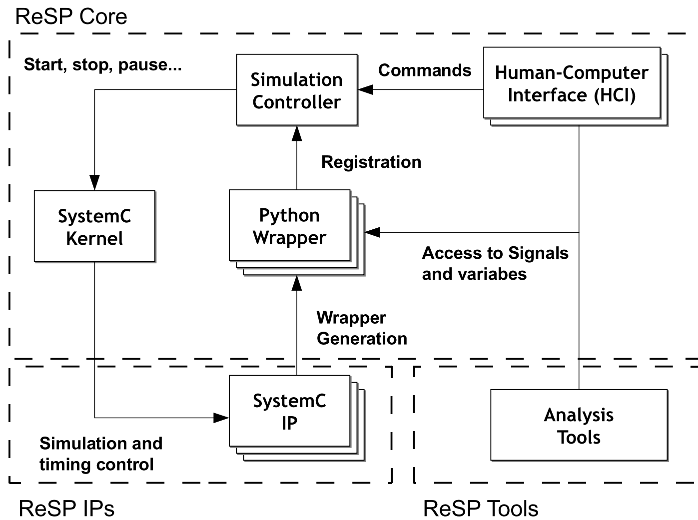


Fig. 6.18 The architecture of the ReSP system simulation platform

C++ or SystemC element (variable, method, signal, etc.) specified in any component. In the following, the terms *platform* and *framework* will be used to indicate the overall ReSP architecture, while *component* describes any top-level SystemC module included into the framework's database (e.g. an ARM processor, a cache, etc.). Figure 6.18 shows ReSP's structural elements:

ReSP Core The simulation core of ReSP is the unmodified OSCI standard SystemC kernel, augmented by a Python wrapper. As Python inherently supports reflection and introspection, it enables run-time, unrestricted access to SystemC variables (and methods, classes, etc.) and the execution of arbitrary function calls. Another noticeable advantage given by the use of Python lies in the *decoupling between the simulator and the SystemC models*: whilst still performing fast compiled simulation, ReSP does not need to have any a-priori knowledge about the components' structure and there is no need to modify the simulator's or the tools' code when components change. This mechanism also allows the declaration of IP modules in Python for quick prototyping at the cost of simulation speed.

The Simulation Controller is a set of Python classes that translate commands coming from the user into SystemC function calls. As an example, it is possible to run, step, pause, or stop the simulation at runtime, something not natively present in SystemC. This concept was introduced in [32] and it is now widely used. The *Human Computer Interface* (HCI) is an Application Programming Interface (API) to perform external control functions on the simulation core. This architecture allows multiple interfaces (such as command line or graphical ones) to be built. The SystemC kernel and the Simulation Controller are run in one execution thread with the rest of the Python wrappers. The HCI and the support tools are executed in a separate thread, and periodically (depending on the tool's needs) synchronized with the

SystemC kernel. This way, the user has full asynchronous control of the simulation (the status of the components can be queried and/or modified even while simulation is running), without consistency loss.

ReSP IPs ReSP can intergrate any SystemC component with minimal effort. In fact, no modifications are needed because ReSP can automatically generate the necessary Python wrapper using only the IP's header. This favors IP reuse (whether internally developed or externally acquired) and component-based design, i.e. the description of new hardware architectures by composition of already existing components. ReSP provides a large library of pre-defined components:

- processors cores, written using a Transactional Automatic Processor generator;
- interconnection system, such as shared memory buses or networks-on-chip;
- memory systems, including simple memories and caches;
- miscellaneous components, such as UARTs and interrupt controllers.

All these components are necessary for building a virtual platform and for running real-world operating systems such as `eCos` and `RTEMS`. Using ReSP's automatic wrapper generation, the library can be extended with user-defined SystemC (or even Python) IPs.

ReSP Tools The introduction of reflection paves the way for the development of a set of tools to perform system-level analysis. Any operation that requires observability can be performed through the Python wrappers. For example, it is possible to include advanced network traffic analysis (latency, throughput. etc.) by observing the network traffic, or to add power modeling to the system by extracting switching activities from the system at runtime. The biggest advantage given by the use of Python lies in the decoupling among the simulator itself and the SystemC models; the simulator does not need to have any a-priori knowledge about the components' structure: there is no need to change the simulator's code even if some components are modified. ReSP comes with a set of pre-defined tools spanning from software debuggers, to power modeling frameworks.

6.7.2 Routine Emulation

When developing parallel programs, the identification of performance bottlenecks is an increasingly important and challenging task, due to the complex relationship between application, operating system, and underlying hardware configuration. In order to gather timing details as soon as possible in the development process we need high level models of such intricate relationships. Accurate simulation also requires taking into account the effect of the operating system.

In order to take software design, development, analysis, and optimization into account, ReSP provides *routine emulation* capabilities; this is a technique that transfers control from the simulated environment (i.e. the software running inside the

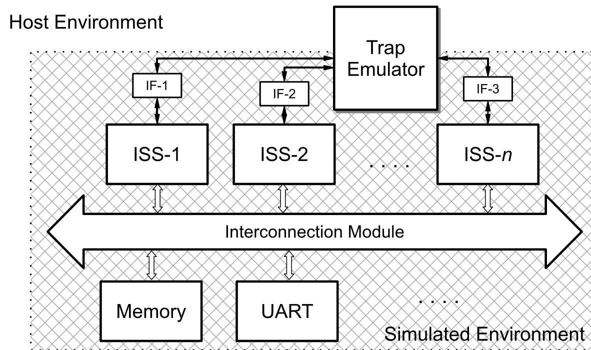


Fig. 6.19 Organization of the simulated environment including the Routine-Emulation module

ISS) to the simulation environment (i.e. ReSP) when specified software routines are encountered. The behaviour of the system in such cases is completely user-defined, and can range from simple breakpoints to the execution of the “trapped” routine in a hardware module.

An overview of the routine emulation mechanism is presented in Fig. 6.19: each ISS communicates with one centralized Routine Emulator (RE), the component responsible for forwarding calls from the simulated to the host environment.

The routine emulator is based on the `GNU/BFD` (Binary File Descriptor) library, used for associating routine names to their address in the executable file containing the application being simulated. When a new routine is scheduled for emulation a new entry is added to the emulator hash-map: this map is indexed by the routine address (determined thanks to the use of the `BFD` library) and it contains the functor implementing the emulated functionality, i.e. the functionality being executed in the simulator space instead of the original routine. At runtime, before the issue of a new instruction, the current Program Counter is checked (using the Processor Interface as explained below) and, if there is a match with one of the addresses in the hash-map, the corresponding functor is executed.

Figure 6.20 presents a detailed example of the mechanism for the `sbrk` system-call: as the library function *malloc* executes, it calls `sbrk`; the routine-emulator checks if the current program counter is to be trapped, and as it recognizes `sbrk`, it stops the ISS execution and performs the appropriate actions.

6.7.3 Concurrency Emulation

Parallel applications are needed to fully exploit MPSoCs, meaning that the system will be composed of different, concurrent paths of execution (threads). Those execution flows are usually managed by an operating system. To avoid running an OS on ReSP’s ISSs, it is necessary to model concurrency management primitives, such as thread creation, destruction, synchronization management, etc. For this purpose, the Routine Emulator was extended for the emulation of concurrency management routines with an additional unit, called *Concurrency Manager* (CM): in

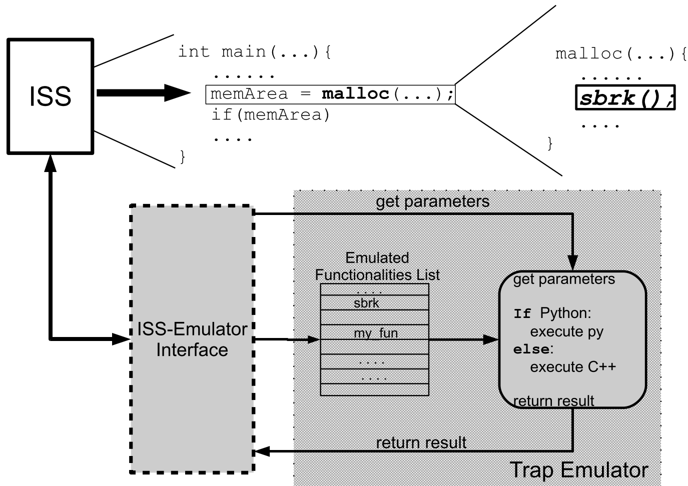


Fig. 6.20 Internal structure and working mechanisms of the function routine emulator

this case the routine emulator intercepts calls for thread creation, destruction, synchronization etc. We then created a placeholder library containing all the symbols (i.e. the function identifiers) of the POSIX-Thread standard, with no implementation present. This ensures that programs using the `pthread` library can correctly compile. During execution, all calls to `pthread` routines are intercepted and forwarded to the CM. Recent versions of the GNU C Compiler (GCC) (after version 4.2) enable compiling multi-threaded programs written using OpenMP, as adopted in hArtes. The compiler translates OpenMP directives into calls of the `libgpomp`, which, in turn contain direct calls to standard POSIX-Thread routines. As such, after the compilation process, an OpenMP annotated program consists of a standard multi-threaded program based on PThreads. Therefore, if the application software is compiled with a recent GCC (at least version 4.2) it is also possible to successfully emulate OpenMP directives enabling, for example, a quick evaluation of its partitioning or mapping.

To prove the usefulness of the methodology for co-design, we analyzed the impact of OS latencies on a set of 8 parallel benchmarks chosen from the OpenMP Source Code Repository [6]. Operating System primitives were divided into 6 classes: thread creation (*th-create*); synchronization initialization, e.g. mutex, semaphore, and condition variable creation (*sync-init*); mutex locking and unlocking (*mutex*); semaphore waiting and posting (*sem*); memory management (*memory*); and general I/O (*io*). Since it is possible to associate custom latencies with emulated routines, we can observe how different values (corresponding to different system configurations) affect the system.

Figure 6.21 shows the average behavior of the 8 benchmarks when the number of cores, as well as the number of OpenMP threads, ranges from 2 to 16 in powers of 2. On each of these hardware configurations, the benchmarks were run with exponentially increasing latencies for each class, yielding a total of 1344 simulations.

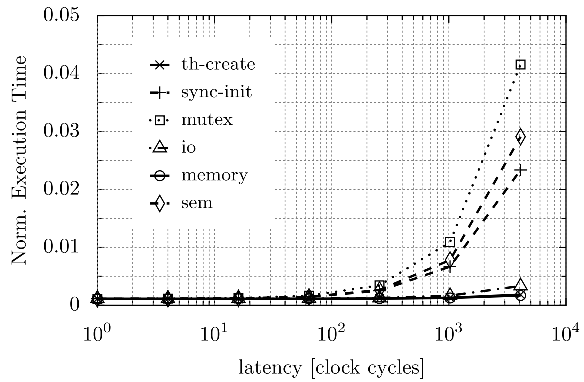


Fig. 6.21 Average execution time on all the 1344 runs for 8 benchmarks and for all the tested number of processors

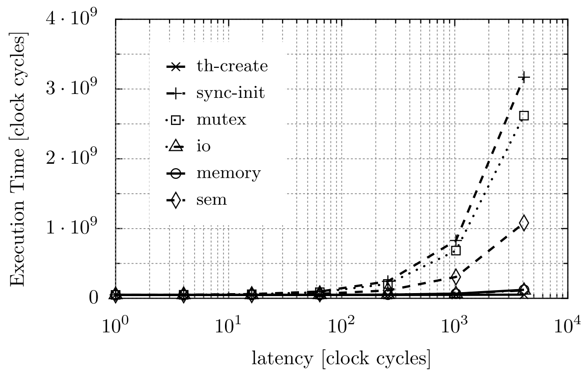


Fig. 6.22 Execution Time in front of different System Call latencies: 2 PE 2 threads

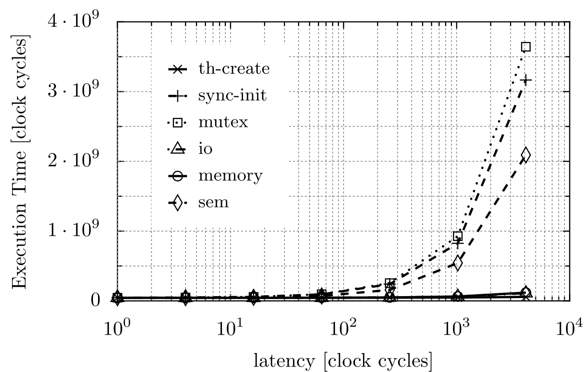


Fig. 6.23 Execution Time in front of different System Call latencies: 4 PE 4 threads

The trend is that, for increasing latency, synchronization primitives are the ones that affect execution time most, while I/O and memory management have negligible effect. Figures 6.22, 6.23, 6.24 and 6.25 show this effect for the `lu` benchmark: the

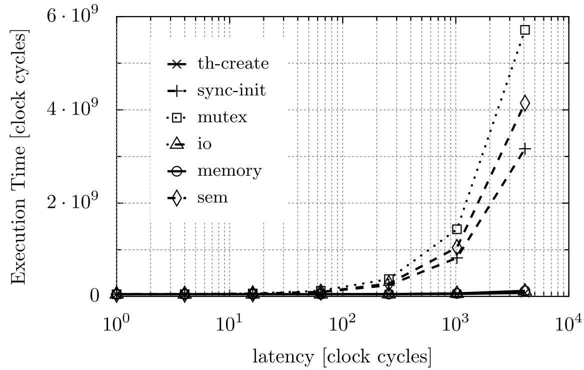


Fig. 6.24 Execution Time in front of different System Call latencies: 8 PE 8 threads

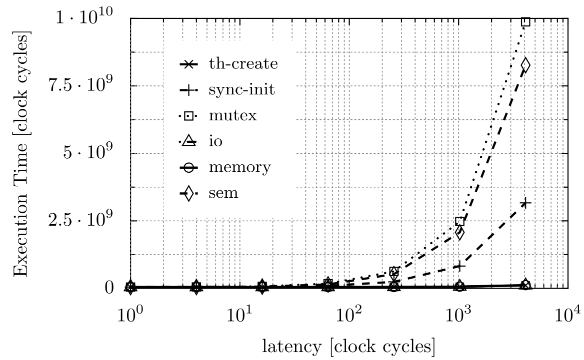


Fig. 6.25 Execution Time in front of different System Call latencies: 16 PE 16 threads

impact of synchronization grows noticeably and quickly exceeds initialization costs as the number of processors and threads is increased.

6.8 Conclusion

In this chapter, we have described a series of extensions that were researched and investigated within the hArtes project but that were not included in the final release of the tool chain. These extensions involved both runtime as well as design time features. They were not included for various reasons. Either the maturity of the techniques proposed was insufficient as was the case for the runtime and hardware support for concurrent execution or because the information generated by the tool could not yet be efficiently used by other components in the tool chain. This was mostly the case for the Quad tool and the high level profiling.

References

1. Arnout, G.: SystemC standard. In: Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific, pp. 573–577 (2000)
2. Beltrame, G., Fossati, L., Sciuto, D.: ReSP: a nonintrusive transaction-level reflective MPSoC simulation platform for design space exploration. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems archive, December, vol. 28(12) (2009)
3. Cescirio, W., Baghdadi, A., Gauthier, L., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., Jerraya, A.A.: Diaz-Nava, M.: Component-based design approach for multicore SoCs. In: 39th Proceedings of Design Automation Conference, 2002, pp. 789–794 (2002)
4. Chaves, R., Kuzmanov, G.K., Sousa, L.A., Vassiliadis, S.: Cost-efficient SHA hardware accelerators. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **16**(8), 999–1008 (2008). ISSN 1063-8210
5. de Mattos, J.C.B., Wong, S., Carro, L.: The Molen FemtoJava Engine. In: Proceedings of IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), USA, September, pp. 19–22 (2006)
6. Dorta, A.J.: The OpenMP source code repository. In: 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05), February, pp. 244–250 (2005)
7. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: a call graph execution profiler. SIGPLAN Not. **17**(6), 120–126 (1982)
8. Hasan, L., Al-Ars, Z.: Accurate profiling and acceleration evaluation of the Smith-Waterman algorithm using the MOLEN platform. In: Proceedings of IADIS International Conference on Applied Computing, Algarve, Portugal, April, pp. 188–194 (2008)
9. http://daedalus.liacs.nl/Site/More
10. http://ce.et.tudelft.nl/MOLEN/Prototype/
11. Kahn, G.: The semantics of a simple language for parallel programming. In: Proc. of the IFIP74 (1974)
12. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans. Comput. **C-22**, 786–793 (1973)
13. Kuzmanov, G.K., Gaydadjiev, G.N., Vassiliadis, S.: The Virtex II Pro MOLEN processor. In: Proceedings of International Workshop on Computer Systems: Architectures, Modelling, and Simulation (SAMOS), Samos, Greece, July. LNCS, vol. 3133, pp. 192–202 (2004)
14. Kuzmanov, G., Gaydadjiev, G.N., Vassiliadis, S.: The Molen media processor: design and evaluation. In: WASP'05
15. Lu, Y., Marconi, T., Gaydadjiev, G.N., Bertels, K.L.M.: A new model of placement quality measurement for online task placement. In: Proceeding of Prorisc Conference, Veldhoven, The Netherlands, November 2007
16. Lu, Y., Marconi, T., Gaydadjiev, G.N., Bertels, K.L.M.: An efficient algorithm for free resources management on the FPGA. In: Proceedings of Design, Automation and Test in Europe (DATE 08), Munich, Germany, March, pp. 1095–1098 (2008)
17. Lu, Y., Marconi, T., Gaydadjiev, G.N., Bertels, K.L.M., Meeuws, R.J.: A Self-adaptive on-line task placement algorithm for partially reconfigurable systems. In: Proceedings of the 22nd Annual International Parallel and Distributed Processing Symposium (IPDPS)—RAW2008, Miami, Florida, USA, April, p. 8 (2008)
18. Lu, Y., Marconi, T., Bertels, K.L.M., Gaydadjiev, G.N.: Online task scheduling for the FPGA-based partially reconfigurable systems. In: International Workshop on Applied Reconfigurable Computing (ARC), Karlsruhe, Germany, March, pp. 216–230 (2009)
19. Lu, Y., Marconi, T., Bertels, K.L.M., Gaydadjiev, G.N.: A communication aware online task scheduling algorithm for FPGA-based partially reconfigurable systems. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2010), Charlotte, North Carolina, USA, May 2010
20. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proc. of PLDI, pp. 190–200 (2005)

21. Marconi, T., Lu, Y., Bertels, K.L.M., Gaydadjiev, G.N.: Online hardware task scheduling and placement algorithm on partially reconfigurable devices. In: Proceedings of International Workshop on Applied Reconfigurable Computing (ARC), London, UK, March, pp. 306–311 (2008)
22. Marconi, T., Lu, Y., Bertels, K.L.M., Gaydadjiev, G.N.: Intelligent merging online task placement algorithm for partial reconfigurable systems. In: Proceedings of Design, Automation and Test in Europe (DATE), Munich, Germany, March, pp. 1346–1351 (2008)
23. Marconi, T., Lu, Y., Bertels, K.L.M., Gaydadjiev, G.N.: A novel fast online placement algorithm on 2D partially reconfigurable devices. In: Proceedings of the International Conference on Field-Programmable Technology (FPT), Sydney, Australia, December, pp. 296–299 (2009)
24. Marconi, T., Lu, Y., Bertels, K.L.M., Gaydadjiev, G.N.: 3D compaction: a novel blocking-aware algorithm for online hardware task scheduling and placement on 2D partially reconfigurable devices. In: Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC), Bangkok, Thailand, March 2010
25. Moscu Panainte, E., Bertels, K.L.M., Vassiliadis, S.: The PowerPC backend Molen compiler. In: 14th International Conference on Field-Programmable Logic and Applications (FPL), Antwerp, Belgium, September. Lecture Notes in Computer Science, vol. 3203, pp. 434–443. Springer, Berlin (2004)
26. Nawaz, Z., Dragomir, O.S., Marconi, T., Moscu Panainte, E., Bertels, K.L.M., Vassiliadis, S.: Recursive variable expansion: a loop transformation for reconfigurable systems. In: Proceedings of International Conference on Field-Programmable Technology 2007, December, pp. 301–304 (2007)
27. Nawaz, Z., Shabbir, M., Al-Ars, Z., Bertels, K.L.M.: Acceleration of smith-waterman using recursive variable expansion. In: 11th Euromicro Conference on Digital System Design (DSD-2008), September, pp. 915–922 (2008)
28. Nawaz, Z., Marconi, T., Stefanov, T.P., Bertels, K.L.M.: Flexible pipelining design for recursive variable expansion. In: Parallel and Distributed Processing Symposium, International, May 2009
29. Nawaz, Z., Stefanov, T.P., Bertels, K.L.M.: Efficient hardware generation for dynamic programming problems. In: ICFPT'09, December 2009
30. Nawaz, Z., Sumbul, H., Bertels, K.L.M.: Fast smith-waterman hardware implementation. In: Parallel and Distributed Processing Symposium, International, April 2010
31. Open SystemC Initiative (OSCI): OSCI TLM2 USER MANUAL, November 2007
32. Paulin, P., Pilkington, C., Bensoudane, E.: StepNP: a System-Level exploration platform for network processors. IEEE Des. Test **19**(6), 17–26 (2002)
33. Pimentel, A.D., et al.: A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans. Comput. **55**(2), 99–112 (2006)
34. Sabeghi, M., Bertels, K.: Toward a runtime system for reconfigurable computers: a virtualization approach. In: Design, Automation and Test in Europe (DATE09), April 2009
35. Sabeghi, M., Bertels, K.: Interfacing operating systems and polymorphic computing platforms based on the Molen programming paradigm. In: Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture, June 2010
36. Sabeghi, M., Sima, V., Bertels, K.: Compiler assisted runtime task scheduling on a reconfigurable computer. In: 19th International Conference on Field Programmable Logic and Applications (FPL09), August 2009
37. Sabeghi, M., Mushtaq, H., Bertels, K.: Runtime multitasking support on reconfigurable accelerators. In: First International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies, June, pp. 54–59 (2010)
38. Sigdel, K., et al.: rSesame-A generic system-level runtime simulation framework for reconfigurable architectures. In: Proc. of FPT09 (2009)
39. Sima, V.M., Bertels, K.: Runtime memory allocation in a heterogeneous reconfigurable platform. In: IEEE International Conference on ReConFigurable Computing and FPGA (2009)

40. Vassiliadis, S., Wong, S., Cotofana, S.: The MOLEN $\rho\mu$-coded processor. In: Proceedings of International Conference on Field-Programmable Logic and Applications (FPL). LNCS, vol. 2147, pp. 275–285. Springer, Berlin (2001)
41. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Moscu Panainte, E.: The MOLEN polymorphic processor. IEEE Trans. Comput. **53**, 1363–1375 (2004)
42. Vassiliadis, S., Wong, S., Gaydadjiev, G.N., Bertels, K.L.M., Kuzmanov, G.K., Moscu Panainte, E.: The Molen polymorphic processor. IEEE Trans. Comput. **53**(11), 1363–1375 (2004)
43. Xilinx, Inc: Early access partial reconfiguration user guide, Xilinx user guide UG208 (2006)
44. Xilinx, Inc: Virtex-4 FPGA Configuration User Guide, Xilinx user guide UG071 (2008)
45. Xilinx, Inc: Virtex-II Platform FPGA User Guide, Xilinx user guide UG002 (2007)
46. Yankova, Y., Bertels, K., Kuzmanov, G., Gaydadjiev, G., Lu, Y., Vassiliadis, S.: DWARV: DelftWorkBench automated reconfigurable VHDL generator. In: FPL'07
47. Yu, L., Abdi, S., Gajski, D.: Transaction level platform modeling in SystemC for Multi-Processor designs. Technical report (2007)

Chapter 7
Conclusion: Multi-core Processor Architectures Are Here to Stay

Koen Bertels

It is clear the that the Multi-core Era has already started some time ago. Multi-core processor architectures clearly solve a number of issues related to power dissipation, the frequency wall etc. However, the adoption of the such platforms poses a number of new challenges of which programmability is definitely one of the toughest ones to crack. Even more so because parallelising compilers and parallel programming languages were a hot research topic in the eighties/nineties and no dominant technology or solution has emanated from that effort. Evidently and maybe most importantly because each time the next generation processor, with an ever higher operating frequency and transistor density, provided the necessary performance upgrade needed. With the arrival of the multi-core architectures, we are witnessing a renaissance of the parallel computing and programming paradigm. Where many research initiatives explore the possibility of designing a new language that would provide the necessary constructs and hooks to efficiently exploit the available hardware resources, they usually represent a radical rupture with the past leading to incompatibility between the existing code base and the latest programming languages. And exactly this incompatibility may turn out to be the show-stopper for many of those emerging approaches. Especially in the Embedded Systems industry, applications tend to have life span covering decades during which period both software and hardware have to be maintained and upgraded when necessary. So, even though traditional views on all the past and future processor generations seems to suggest that the latest generation platforms are automatically adopted, the ES reality is definitely far away from this. All kinds of generations of processor architectures continue to live in parallel and only when hardware components are impossible to find, will one be required to go to a different, more recent platform. Figure 7.1 illustrates this particular phenomenon and smooth as possible. Once a particular path is chosen, it is

K. Bertels (✉)
Fac. Electrical Engineering, Mathematics & Computer Science, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: k.l.m.bertels@tudelft.nl

K. Bertels (ed.), *Hardware/Software Co-design for Heterogeneous Multi-core Platforms*,
DOI 10.1007/978-94-007-1406-9_7, © Springer Science+Business Media B.V. 2012

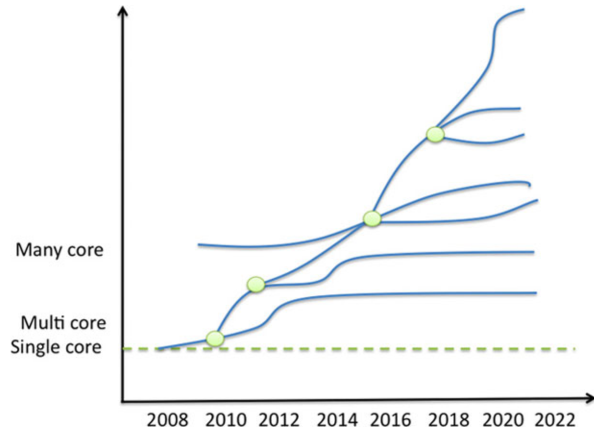


Fig. 7.1 Persistence of legacy platforms and applications

not going to be a trivial task to move from one branch to another without a major redesign and re-implementation effort.

In addition to the above, there is a second important trend in the semi-conductor industry, namely, the introduction of different kinds of computing cores, leading to heterogeneous multi-core processor architectures. This poses additional challenges as parallelising can no longer be done without taking the specific and low level characteristics of the hardware into account. The need to take low level hardware information into account seems to go against the need to provide a cross-platform programming approach.

In this book, we have described how the hArtes project has addressed the above two challenges. Starting point and corner stone of the hArtes approach was the adoption of the Molen Machine Architecture which provides a uniform machine organisation to the developer irrespective of the specific underlying hardware. This particular architecture also provides the so-called Molen programming paradigm offering the necessary programming hooks to the developer. We have implemented the Molen on a particular hardware platform comprising an ARM processor, the Atmel Diopsis DSP and the Xilinx Virtex 4 FPGA and have demonstrated that for several industrial,high performance applications that the hArtes technology allows to partition, transform and map an existing application in such a way that it can be efficiently executed on the hArtes platform.

In total around 150 scientific publications were presented at various international conferences and published in international journals. In addition to the scientific output, other projects have been initiated building on top of the results of the hartes project, such as the FP7 project REFLECT, and the Artemisia iFEST project. In addition, some partners have taken the initiative to start a company, BlueBee[1] that will bring a large part of the tools to the market.

Is the hArtes approach, based on the Molen machine, the final answer to all the issues to which the computing industry is faced? Clearly not. And hopefully not.

[1] www.bluebeetech.com.

The hArtes claim is that for multi-core platforms, providing a non-disruptive transition path to such platforms is of crucial importance. Recent announcements by companies such as STM, Xilinx but also Intel and Arm, give a clear indication that multi-core is the first step of industry into the realm of parallel computing. The hArtes approach is unlikely to be scalable to computing platforms having hundreds or even thousands of computing cores leaving a lot of opportunities of interesting and challenging research.