METHODOLOGIES FOR IMPLEMENT VIDEO DECODERS OVER MULTIPROCESSOR PLATFORMS

METODOLOGÍAS PARA IMPLEMENTAR DESCODIFICADORES DE VÍDEO SOBRE PLATAFORMAS MULTIPROCESADOR

TRABAJO FIN DE MÁSTER

Autor: Daniel Gómez-Lobo García
Ingeniero Técnico de Telecomunicación

Tutor: Fernando Pescador del Oso
Doctor Ingeniero de Telecomunicación

Julio de 2013
<table>
<thead>
<tr>
<th><strong>Título</strong></th>
<th>METHODOLOGIES FOR IMPLEMENT VIDEO DECODERS OVER MULTIPROCESSOR PLATFORMS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Autor</strong></td>
<td>Daniel Gómez - Lobo García          VºBº</td>
</tr>
<tr>
<td><strong>Tutor</strong></td>
<td>Fernando Pescador del Oso</td>
</tr>
<tr>
<td><strong>Ponente</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Tribunal**

| **Presidente** | Luis Ignacio Ortiz Berenguer                       |
| **Secretario** | Eduardo Juárez Martínez                            |
| **Vocal**      | Mariano Ruiz González                              |

**Fecha de lectura** 18-07-2013

El Secretario:
ACKNOWLEDGEMENTS

I would like to express my gratitude to the people that helped me and encouraged me to carry out this Master’s Thesis work. I have learned a lot thank to them.

I am thankful to Eduardo Juárez Martínez, who proposed me the opportunity of working in his attractive research group, the GDEM (Grupo de Diseño Electrónico y Microelectrónico).

Thanks to Mickaël Raulet for his help and advices in the development of this project. Thanks to Matías Garrido González for his collaboration and advices in the development of the research paper submitted to ESTIMedia 2013.

Thanks to Jesús Pablo Caño Velasco and Rafael Pons Castellanos to their company in the GDEM laboratory and the mutual support that we have given us.

I am thankful to Miguel Chavarriás Lapastora, who has been always sharing his knowledge and has worked closely, leading the research. Thank him for his collaboration, help and teaching.

I am thankful to Fernando Pescador del Oso, who has been not only an excellent teacher in my first university course, but also, an excellent mentor, advisor and leader during the development of this project. Thank him for being a hard-worker looking always for the best for his students. Thank him for his spirit, encouragement, cooperation, advices, patience and teaching.

Finally, I would like to express also my gratitude to those other people who are contributing daily to my happiness and who are always looking for my time and the best for me. Thank to my family (specially, to my parents) and my best friendships.
### Index

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>i</td>
</tr>
<tr>
<td>Index of figures</td>
<td>vi</td>
</tr>
<tr>
<td>Index of tables</td>
<td>x</td>
</tr>
<tr>
<td>Index of lists of changes</td>
<td>xii</td>
</tr>
<tr>
<td>Acronyms</td>
<td>xiv</td>
</tr>
<tr>
<td>Resumen</td>
<td>xvi</td>
</tr>
<tr>
<td>Summary</td>
<td>xvii</td>
</tr>
<tr>
<td><strong>1</strong> Introduction and objectives</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Introduction</td>
<td>2</td>
</tr>
<tr>
<td>1.2. Context of the work</td>
<td>3</td>
</tr>
<tr>
<td>1.3. Objectives</td>
<td>4</td>
</tr>
<tr>
<td>1.4. Structure of this document</td>
<td>5</td>
</tr>
<tr>
<td><strong>2</strong> Reconfigurable Video Coding</td>
<td>6</td>
</tr>
<tr>
<td>2.1. The origin of RVC</td>
<td>7</td>
</tr>
<tr>
<td>2.2. The approach of RVC</td>
<td>8</td>
</tr>
<tr>
<td>2.3. Specification of a decoder based on RVC</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1. BSD</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2. FND</td>
<td>12</td>
</tr>
<tr>
<td>2.3.3. FU</td>
<td>14</td>
</tr>
<tr>
<td>2.4. CAL</td>
<td>15</td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>3</td>
<td>High Efficiency Video Coding</td>
</tr>
<tr>
<td>3.1</td>
<td>Video coding standards evolution</td>
</tr>
<tr>
<td>3.2</td>
<td>Features of HEVC</td>
</tr>
<tr>
<td>3.3</td>
<td>HEVC in RVC</td>
</tr>
<tr>
<td>4</td>
<td>Multicore platforms and the dataflow model of RVC</td>
</tr>
<tr>
<td>4.1</td>
<td>GPP systems</td>
</tr>
<tr>
<td>4.1.1</td>
<td>PC microprocessor</td>
</tr>
<tr>
<td>4.1.2</td>
<td>ARM device</td>
</tr>
<tr>
<td>4.2</td>
<td>DSP systems: the DM6437 EVM</td>
</tr>
<tr>
<td>4.3</td>
<td>Design and running of RVC applications over multicore architectures</td>
</tr>
<tr>
<td>4.4</td>
<td>Reason of running an RVC decoder on a single DSP core</td>
</tr>
<tr>
<td>5</td>
<td>Software development tools and methodology</td>
</tr>
<tr>
<td>5.1</td>
<td>Development tools</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Eclipse</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Open RVC-CAL Compiler</td>
</tr>
<tr>
<td>5.1.3</td>
<td>CMake</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Code Composer Studio</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Code::Blocks, Visual Studio or Eclipse CDT</td>
</tr>
<tr>
<td>5.1.6</td>
<td>Other tools</td>
</tr>
<tr>
<td>5.2</td>
<td>Working methodology</td>
</tr>
<tr>
<td>6</td>
<td>Procedure for software migration</td>
</tr>
</tbody>
</table>
6.1. Overview 51
6.2. Inspection of the starting point 52
6.3. Preliminary settings of parameters 53
   6.3.1. Adjustment of image size 53
   6.3.2. Other parameters 55
6.4. Generation of the C backend 55
6.5. Create and configure the DSP project 56
6.6. Adapt the thread API 59
6.7. Enhance the main function and set input parameters 61
6.8. Initialization of software parts 63
   6.8.1. Initialize threads before launch the actor’s scheduler 64
   6.8.2. Initialize actor communication FIFOs 65
   6.8.3. Initialize string variables for input/output and mapping files 67
   6.8.4. Initialize other variables 68
6.9. SDL references suppression 69
6.10. Remove multicore management code 71
6.11. Implement unavailable functions 72
   6.11.1. Replacement of the fdopen function 73
   6.11.2. Function strdup 73
   6.11.3. Library stat and function fileno 75
   6.11.4. Functions to measure performance 79
   6.11.5. YUV files 83
6.12. Others

6.13. Need of automating the migration

7 Experiments and results

7.1. Test sequences

7.2. Test automation

7.3. Results obtained for HEVC-RVC

7.4. Comparison with other HEVC decoders

8 Conclusions and future work

8.1. Conclusions

  8.1.1. About RVC

  8.1.2. About multicore DSP platforms

  8.1.3. About the developed methodology

8.2. Future works

9 References

10 Annex I: CD content

11 Annex II: Structure of the C backend

12 Annex III: Installation guide and tutorial

12.1. Installation guide

12.2. Tutorial

  12.2.1. Step one - Get applications

  12.2.2. Step two - Import the applications to Eclipse and build
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.2.3</td>
<td>Step three - Work with Orcc projects</td>
<td>124</td>
</tr>
<tr>
<td>12.2.4</td>
<td>Step four - Generate a backend</td>
<td>126</td>
</tr>
<tr>
<td>12.2.5</td>
<td>Step five - Use CMake to establish a project for the IDE</td>
<td>130</td>
</tr>
<tr>
<td>12.2.6</td>
<td>Step six - Use the preferred IDE</td>
<td>132</td>
</tr>
</tbody>
</table>
Index of figures

Figure 1. Conceptual diagram of codec applications based in RVC. .................. 8
Figure 2. RVC framework [12]. ........................................................................ 9
Figure 3. Example of application of the RVC framework [12]. ..................... 10
Figure 4. Specification of the ADM [3].......................................................... 11
Figure 5. Graphical view of a network with four functional blocks. .............. 13
Figure 6. XML-based textual representation of the network shown in Figure 5. 13
Figure 7. Graphical view of the network of the HevcDecoder component. ...... 14
Figure 8. Idealized representation of an actor entity. ...................................... 15
Figure 9. Chronogram with relevant video coding standards. ....................... 21
Figure 10. Typical block diagram of a HEVC encoder [21]. ......................... 22
Figure 11. Typical block diagram of a HEVC decoder [23]. ......................... 22
Figure 12. Architecture of the OMAP44x family of processors. ..................... 30
Figure 13. PandaBoard. .................................................................................. 31
Figure 14. DM6437 Evaluation Module block diagram [31]. ....................... 33
Figure 15. Memory map of the DM6437 EVM............................................... 34
Figure 16. Example of the Master/Slave model (left) and the Data Flow model (right) [29]. ........................................................................................................ 35
Figure 17. Example of an RVC Dataflow Process Network............................ 35
Figure 18. Round-robin (left) and Data-driven/Data-Demand (right) scheduling policies [3]........................................................................................................ 37
Figure 38. Main folder structure of a C backend. ................................. 114

Figure 39. SVN Workbench to download the decoder applications for Orcc. 120

Figure 40. With “NightlyExample” selected, initiate the download of the Orcc applications................................................................. 121

Figure 41. Messages shown during the download process.................... 121

Figure 42. Content of the “trunk” folder............................................ 122

Figure 43. Selection of a workspace................................................. 123

Figure 44. Imported projects, with highlighting of the ones required. .... 124

Figure 45. Block diagram of Top_mpegh_part2_main.xdf.................. 125

Figure 46. FU network of the block “HevcDecoder”. .......................... 125

Figure 47. RVC-CAL code of the block “Source”. ............................. 126

Figure 48. Compilation settings tab. .................................................. 127

Figure 49. Compilation options tab.................................................... 128

Figure 50. Mapping tab..................................................................... 129

Figure 51. Common tab..................................................................... 129

Figure 52. Messages displayed in the console during an Orcc compilation.. 130

Figure 53. Initial aspect of CMake...................................................... 131

Figure 54. Configuration of the building environment for the decoder with CMake................................................................. 132

Figure 55. Execution of “make” and start of the generation process. .... 133

Figure 56. End of the generation process; the decoder solution has been built. .................................................................................. 133
Figure 57. Execution of the decoder, the number of fps decoded is shown in the console.

Figure 58. Execution of the decoder, the video sequence decoded is displayed.
Index of tables

Table 1. Picture partitioning concepts in HEVC. .................................................. 23
Table 2. Top_mpegh_part2_main.xdf. (Top network of the decoder). .............. 25
Table 3. HEVCDecorderInter.xdf. (The decoder itself). ............................... 26
Table 4. GenerateInterInformation.xdf. ............................................................ 26
Table 5. xIT.xdf............................................................................................... 26
Table 6. IT4x4.xdf, IT8x8.xdf, IT16x16.xdf, IT32x32.xdf (for the last three, substitute 4x4 by 8x8, 16x16 and 32x32 respectively). ................................. 26
Table 7. Dbf.xdf.............................................................................................. 27
Table 8. MD5Net.xdf........................................................................................ 27
Table 9. Features of the PC employed. ........................................................... 29
Table 10. Last versions of the Eclipse platform.............................................. 42
Table 11. Calculation of the memory required for different buffer image sizes. 55
Table 12. Settings for each CCS project. (* Depends on the hardware employed). ................................................................. 58
Table 13. Dependencies among CCS projects and TI libraries....................... 58
Table 14. Short description of function “memset”. ............................................ 65
Table 15. Short description of function “strdup”. ............................................ 73
Table 16. Short description of functions “fstat” and “fileno”. ......................... 77
Table 17. Sequences of the testbench and their main features....................... 90
Table 18. Test settings in the OS script .......................................................... 92
Table 19. Mapping of FUs to common decoder functional blocks. .................. 94
Table 20. HEVC-RVC: Performance analysis and profiling results. ...............95

Table 21. Comparison of several HEVC implementations in one processor core. ...................................................................................................................................98

Table 22. General characterization of different computation technologies. .... 102

Table 23. CD content (folders of the root directory). ....................................112

Table 24. Files inside the “libs/orcc” folder.................................................114

Table 25. Files inside the “libs/roxml” folder..............................................115

Table 26. Decoder source files inside the “src” folder. ...............................115

Table 27. Example of XCF file for a single-core implementation..................116

Table 28. Software development environment. (*tool not recommended). .....118
Index of lists of changes

List 1. Change in semaphore.h (PTE).................................................................61
List 2. Change in Top_mpegh_part2_main.c (RVC)........................................61
List 3. Change in function “main” of Top_mpegh_part2_main.c (RVC). .........62
List 4. Change Top_mpegh_part2_main.c (RVC) to support automated tests. 63
List 5. Changes in function launcher() of Top_mpegh_part2_main.c (RVC) ....64
List 6. Declaration of the macro DECLARE_FIFO (orcc_fifo.h).......................65
List 7. Changes in function launcher() of Top_mpegh_part2_main.c (RVC).....65
List 8. Changes in orcc_util.c (ORCC)...........................................................67
List 9. Changes in HevcDecoder_generateInfo_GenerateRefList.c (RVC).....68
List 10. Change in Top_mpegh_part2_main.c (RVC)......................................69
List 11. Changes in fpsPrint.c (ORCC)..........................................................69
List 12. Change in display.c (ORCC)............................................................71
List 13. Changes in Top_mpegh_part2_main.c (RVC)...................................71
List 14. Change in roxml.c (ROXML)............................................................72
List 15. Changes in roxml.c (ROXML)...........................................................73
List 16. Changes in roxml.c (ROXML)..........................................................74
List 17. Changes in function “init_orcc” of orcc_util.c (ORCC)..................74
List 18. Changes in accessFile.c, compare.c, compareyuv.c, y source.c (ORCC)...........................................................................................................76
List 19. Changes in function “source_sizeOfFile” of source.c (ORCC).......77
List 20. Changes in function “sizeOfFile” of accessFile.c (ORCC).............78
List 21. Changes in function “compareYUV_init” of compareyuv.c (ORCC). ..... 78
List 22. Changes in function “source_sizeOfFileFd” of source.c (ORCC). ..... 79
List 23. Code of medidas.h ................................................................. 79
List 24. Code of medidas.c ............................................................... 80
List 25. Changes in orcc_scheduler (ORCC) ...................................... 80
List 26. Changes in Top_mpegh_part2_main.c (RVC) ......................... 81
List 27. Changes in display.c (RVC) .................................................. 83
List 28. Changes in fpsPrint.c (ORCC) .............................................. 85
List 29. Change in function “compareKey” of MD5_compute.c (RVC) .... 86
List 30. Change in roxml-internal.h (ROXML) ..................................... 86
Acronyms

ADM: Abstract Decoder Model
API: Application Programming Interface
BSD: Bitstream Syntax Description
CAL: CAL Actor Language
CCS: Code Composer Studio
CD: Compact-Disc
DD: Decoder Description
DSS: Debug Scripting Server
DSP: Digital Signal Processing
FPGA: Field-Programmable Gate Array
FND: Functional unit Network Description
FU: Functional Unit
GPP: General Purpose Processor
HEVC: High Efficiency Video Coding
IEC: International Electrotechnical Commission
ISO: International Standardization Organization
ITU: International Telecommunication Union
JCT-VC: Joint Collaborative Team on Video Coding (equivalent to JVT)
JVT: Joint Video Team (equivalent to JCT-VC)
MPEG: Moving Pictures Experts Group
MT: Master Thesis
MVC: Multiview Video Coding
OS: Operating System
PC: Personal Computer
RVC: Reconfigurable Video Coding
TI: Texas Instruments
VCEG: Video Coding Experts Group
VTL: Video Tool Library
XML: eXtensible Markup Language
Resumen

Los estándares de codificación de vídeo desarrollados más recientemente, como HEVC (High Efficiency Video Coding, aprobado en enero de 2013), requieren para su implementación el uso de dispositivos capaces de soportar una elevada carga computacional. Teniendo en cuenta que actualmente no es suficiente con utilizar un único Procesador Digital de Señal (DSP), han aparecido recientemente dispositivos multinúcleo en el mercado. Sin embargo, debido a su novedad, la metodología de trabajo que permite elaborar soluciones para tales configuraciones se encuentra en un estado muy inicial, ya que actualmente la mayor parte del trabajo debe realizarse manualmente. En consecuencia, el objetivo marcado consiste en encontrar metodologías que faciliten este proceso.

El estudio se ha centrado en extender una metodología, en desarrollo, para la generación de soluciones para PC y sistemas empotrados. Durante dicho estudio se han empleado los estándares RVC (Reconfigurable Video Coding) y HEVC, así como DSPs de la compañía Texas Instruments. En su desarrollo se ha tratado de atender a todos los factores que influyen tanto en el desarrollo como en la puesta en marcha de estas nuevas implementaciones de descodificadores de vídeo; abarcando desde las herramientas a utilizar hasta aspectos del particionado de los algoritmos, sin que por ello se produzca una reducción en el rendimiento de las aplicaciones.

Los resultados de este estudio son una descripción de la metodología empleada, la caracterización del proceso de migración de software, y medidas de rendimiento para el estándar HEVC en una implementación basada en RVC.
Summary

The latest video coding standards developed, like HEVC (High Efficiency Video Coding, approved in January 2013), require for their implementation the use of devices able to support a high computational load. Considering that currently it is not enough the usage of one unique Digital Signal Processor (DSP), multicore devices have appeared recently in the market. However, due to its novelty, the working methodology that allows produce solutions for these configurations is in a very initial state, since currently the most part of the work needs to be performed manually. In consequence, the objective set consists on finding methodologies that ease this process.

The study has been focused on extend a methodology, under development, for the generation of solutions for PCs and embedded systems. During this study, the standards RVC (Reconfigurable Video Coding) and HEVC have been employed, as well as DSPs of the Texas Instruments company. In its development, it has been tried to address all the factors that influence both the development and deployment of these new implementations of video decoders, ranging from tools up to aspects of the partitioning of algorithms, without this can cause a drop in application performance.

The results of this study are the description of the employed methodology, the characterization of the software migration process and performance measurements for the HEVC standard in an RVC-based implementation.
1 Introduction and objectives
1.1. Introduction

Video coding and decoding are very demanding applications that require technology supports with a great computing capacity. Currently, the different technological alternatives used to implement this kind of applications can be classified in three groups: the based on General Purpose Processors (GPP), the based on specific architectures and the based on Digital Signal Processors (DSP).

When deciding what technological alternative is the most suitable for these applications, it must be taken into account the constant evolution of video coding standards and the successive modifications carried out over them, because they entail that development time of coders and decoders should be each time shorter. In this respect, the advantage of GPPs and DSPs with respect to specific hardware architectures is the easiness to modify the standard to implement, since it only implies the replacement of the software that runs on the processor. Nevertheless, the price and energy consumption are higher with respect to those systems specifically designed to implement a standard.

Due to the versatility offered by GPPs and DSPs on the software to run, the manufacturers of multimedia-capable devices have opted for these technologies; producing multimedia devices that are mainly based on a GPP and a DSP in order to optimize and adapt the execution of multiple tasks.

In general these solutions run the video decoders on the DSP. However, there are currently a couple of factors that frustrate the implementation of the most advanced decoders running in real time on these devices. The former is the complexity of the latest video coding standards, like H.264/MVC and HEVC, which demand higher computing capacity than their predecessors. The latter is that the spatial resolution of images is increasing, which implies also a higher demand of computing capacity.

To address the implementation of these new standards and withstand high screen resolutions, it is necessary to employ more than one processor to distribute among them the computational load. In this sense, devices that integrate several DSPs have appeared in the market.

Implementing solutions on these devices is currently a task that involves a work which is laborious, inflexible and costly in terms of development time. The main causes are that the procedure is not sufficiently automated, the distribution of the
computational load is not trivial, and changes in standards and distribution settings are more or less usual.

At the present time there are already some research groups working on methodologies to automatically optimize this task [1]. Despite that, these studies have been focused mostly on implementations based on GPPs and FPGAs.

The Group of Electronic and Microelectronic Design (GDEM), of the Technical University of Madrid, has set the objective to contribute to these methodologies with respect to the implementation of the standards on multiple DSPs. It is supported by its experience [2] [3] to achieve this goal.

This Master’s Thesis (MT henceforth) is aligned with the mentioned goal of the GDEM, trying to provide an advance within its research activities. Given their innovative character, this work will be done following two standards called Reconfigurable Video Coding (RVC) [4] [5] and High Efficiency Video Coding (HEVC) [6]; the latter is the most recent video coding standard since it was approved in January 2013.

1.2. Context of the work

The research is placed within a context that implies considerable difficulties, not only technical but also due to aspects like team coordination, scheduling, communication, tracking of the progress, or exploration of new alternatives. As instances, the difficulties identified at the beginning of this work and confirmed later, were:

- The international and dynamic context of the research, where contributing groups (from INSA Rennes [7] and EPFL [8]) constantly work on the improvement of certain working tools, as well as on the development of some required solutions using such tools. This circumstance boosts the occurrence of failures, non operative versions, continuous updates, lack of documentation, etc.

- The compatibility between tools. It is necessary to explore whether there are effective and efficient procedures that allow the interconnection of the tools employed, so that the results of one can be applied as input to another. In the absence of such an option, appropriate solutions should be looked for.
1. Introduction and objectives.

- The complexity added by the lack of documentation face-to-understand properly how to handle some tools or how interpret their results. Even when the documentation is proper, it is often too bulky for certain purposes and must be analyzed carefully.

- The requirements imposed by the platform which houses the DSP when running algorithms on the same. In a further step, the requirements imposed by the multi-DSP platform to distribute and execute algorithms on them.

This has forced to define a challenge to overcome and to work by objectives that are determined as soon as a new advance of the research project is reached in a reasonable amount of time.

1.3. Objectives

The main objective consists on proposing a methodology that allows execute video decoders created according to the RVC standard using DSPs of the company Texas Instruments (TI) [9].

To achieve this aim keeping in mind the possible consequences of the issues of the context described in the previous section, several smaller objectives were set. The following list summarizes these partial objectives:

- Study the video standards, focusing on HEVC and RVC.

- Study, test and propose working tools.

- Study the platform based on the multicore DSP which is employed.

- Evaluate the methodologies developed.

- Evaluate the performance of a standard using this environment.

- Generate documentation that serve as help for future works.
1. Introduction and objectives.

1.4. Structure of this document

The approach followed in the writing of this document consists on highlighting the most relevant aspects of the work done in order to provide a meaningful understanding of what has been done.

The document has been organized in nine chapters and three annexes. Summarizing, a theoretical introduction is done from Chapter 1 to Chapter 4, and the most experimental work is explained from Chapter 5 to Chapter 8. In the following lines, the topics dealt by each chapter are outlined.

Chapter 1 is the current chapter; it offers an overview of the purposes of the MT, the context where it is developed and this document as memory of the MT. Chapter 2 explains what is RVC and how is standardized. Chapter 3 summarizes the motivation for the HEVC standard, it features and its current state within an RVC implementation. Chapter 4 encompasses the different multicore platforms employed, highlighting the features of these test devices. Chapter 4 introduces also how an RVC application can be mapped in a processor system with any number of cores.

Chapter 5 introduces the tools employed. The most important information covered by this chapter is the workflow (or working methodology) developed. Based on this working methodology, Chapter 6 presents the needed steps to migrate the decoder’s software from an only-GPP environment to an only-DSP environment. Chapter 7 describes the test performed to an HEVC decoder based on RVC, and the results attained. Chapter 8 describes the conclusions obtained and the future work lines that may be followed.

After the chapters the references are reported. Finally, there are three annexes. The first lists the contents of the CD that is delivered with this document. The second gives additional information that supports some explanations along this MT document. The last provides an installation guide and a quick tutorial to start working with the methodologies introduced.
2 Reconfigurable Video Coding
2. Reconfigurable Video Coding.

2.1. The origin of RVC

Since its foundation in 1988, the Moving Pictures Experts Group (MPEG) [10] has led the development of standards and technologies that deal with the compression, codification and transmission of digital audio, video and multimedia data. During its 28-years history, the MPEG has reinvented its working framework several times with the purpose of effectively adapt to the each time higher complexity of technology and broader demands of the market.

One of these reinventions was proposed within MPEG in 2004 with the aim at providing a more suitable framework for video codec specifications and developments. The result was a new standard called Reconfigurable Video Coding (RVC), which defines video codecs at the level of library components instead of monolithic algorithms (it was the previous methodology adopted by MPEG).

RVC standard is designated as ISO/IEC MPEG RVC and it is supported by the standards ISO/IEC FDIS 23001-4 (or MPEG-B part 4) [4] and ISO/IEC FDIS 23002-4 (or MPEG-C part 4) [5].

This innovative approach establishes an adaptive framework that overcomes many limitations of monolithic specifications like [11]:

- Lack of flexibility, because they do not allow the combination of coding algorithms from different standards enabling to achieve specific design or performance trade-offs and thus fill, case by case, the requirements of specific applications. Moreover, it may result in non-efficient implementations because a decoder conformant to a standard has to support all the specification independently of the application scenario or the set of tools required.

- Drawbacks consequence of specifications performed by means of textual descriptions or reference software written in C, C++ or VHDL programming languages, especially when the complexity of the video coding standard is high. Such means employ non-optimized non-modular software packages, hide features of the standards (for instance, the original knowledge about the intrinsic properties of the algorithms, which are useful when deploying and optimizing them on multi-core
platforms), and force system designers and developers to rewrite these software packages and then adapt them to their working methodologies.

2.2. The approach of RVC

Explained in a nutshell, RVC establishes the framework to define video codecs at the level of library components. The codecs are defined in this way in order to meet the following requirements regarding algorithms [11]: scalable parallelism, modularity and reuse, concurrency and scheduling, abstraction and portability, and encapsulation and adaptivity. These considerations provide a better scenario for designers and developers; for example it allows to: exploit commonalities between standards (materialized through common components), reconfigure the decoder by adjusting the loaded components, extract easily the dataflow for deployment purposes on multicore platforms or implement a component of the library with platform independence and abstracting from the inner details of other components.

Due to the flexibility of the decoder, the decoding platform needs to know which components are necessary and how to use them in order to implement a concrete decoder model. That is accomplished by supplying the Decoder Description (DD) to the decoding platform. As Figure 1 represents, the encoder (which can represent for example specific equipment generating a video stream in real-time or a file storing the video data) sends the DD together with the encoded video data or bitstream.

![Figure 1. Conceptual diagram of codec applications based in RVC.](image)

The DD is composed of two important types of information:

- The Bitstream Syntax Description (BSD): it describes the structure of the bitstream and it is written in RVC-BSD language (BSDL).

- The Functional unit Network Description (FND): it describes the required components, the values of their parameters during the instantiation process and the connections among them. A component usually
receives the name Functional Unit (FU) and is considered a video coding tool. The FND is written in FU Network Language (FNL).

Once the DD is known, it is possible to elaborate the RVC-compliant decoder. Figure 2 shows how it is done. In a first stage, according to the information provided in the DD: the required FUs are retrieved from component libraries, the bitstream parts are associated to the parameters of some FUs and the network of FUs is created, resulting in a model called Abstract Decoder Model (ADM).

In Figure 2, it is shown that an ADM is totally described using both the FNL and RVC-CAL languages; owing to the network of FUs is written in FNL and the software architecture of FUs is implemented with a special language called RVC-CAL, which will be explained later. Because there is not any consideration in the description of the ADM about the architecture of the target platform, all the previous process is done with independence of the target platform; hence, abstraction is preserved. It is worth noting also that this framework allows the reconfiguration of the decoder, forming a new ADM when the DD changes.

Figure 2. RVC framework [12].
In a second stage, when the solution must be deployed on a target device, in general, the demanded components need to be “translated” from RVC-CAL to the suitable language (C, Java or VHDL, for example). Finally, by means of the ADM and the correctly implemented components, the decoder is built and employed.

As can be noticed from Figure 2, the parts of the RVC standard define only two things within this working framework. Firstly, MPEG-B part 4 describes the overall framework and establishes the languages employed along it. Secondly, MPEG-C part 4 establishes the normative component library, which is often named Video Tool Library (VTL) and is focused on existing MPEG standards. Nevertheless, there is also the chance of using non-normative libraries; since it is recommendable have a mechanism for the creation and use of alternative video coding tools.

To end this section an example of application of the RVC framework is shown in Figure 3. The figure represents how the Scalable Video Coding (SVC) specification, an annex of the H.264 video coding standard, can be extended with RVC, providing useful degrees of freedom.

Figure 3. Example of application of the RVC framework [12].

SVC describes how encode/decode a video stream with different levels of complexity, giving a solution for the problems posed by the features of modern video systems (streaming, conferencing, surveillance, broadcast and storage using the whole heterogeneity of devices). As can be seen in Figure 3, the solution is based on a high-quality encoded video bitstream composed by a base layer and several enhancement layers (additional data to enhance the quality and the spatial and temporal resolutions).
The decoding devices discard those enhancement layers that they do not process. Where RVC takes place is precisely in the reconfiguration of the decoders of these devices, allowing the dynamic adaption to the state of the system. For instance, when the battery of a mobile phone is low, this device can decide to decode only the base bitstream, which is less complex and requires a basic decoder that can be rearranged in the processing system.

### 2.3. Specification of a decoder based on RVC

As can be seen in Figure 4, the essential elements of the RVC framework consist on the DD and the VTL. Explain how these elements are defined technically is the purpose of this section.

The DD is defined with the BSD and the FND, and the VTL is defined with its FUs. The BSD is employed to generate the appropriate syntax parser to decode the corresponding input encoded data. The FND serves to instantiate the network of FUs contained in the VTL. The final result is the specification of the ADM, which is a behavioral model of the decoder and as it will be justified later, it is described as a dataflow diagram.

![Figure 4. Specification of the ADM [3].](image-url)
2.3.1. BSD

BSDL is the language used to define the BSD, which represents all the possible structures of the incoming bitstream. BSDL is a language derived from XML and hence the BSD is presented with an XML document. The BSD is therefore an instance of a BSDL schema (for further details, consult [13]). In the RVC framework, BSDL has been chosen because:

- It is stable and already defined as an international standard. Moreover, the BSDL is a human- and machine-readable format.
- The XML-based syntax interacts well with the XML-based representation of the RVC decoder configurations. In addition, the XML-based syntax integrates well with the XML infrastructure of existing tools.
- By means of standard tools, like for example XSLT, the parser of a decoder can be easily generated from the BSDL schema given in the DD.
- The XML formalism allows organizing the BSD in a hierarchical structure with different levels of granularity, and therefore it is fully customizable for the application requirements.

2.3.2. FND

FNL is the language used to define the FND, which describes the network configuration of the video codec. FNL is a dialect of XML and hence the FND is presented with an XML document. This language has been selected for the FND because it is a flexible and widespread way to create common information formats in a structured manner.

In this MT, the Graphiti editor (described later) has been employed in order to edit networks of FUs, because the XDF (Xml Dataflow Format) files managed by Graphiti describe them in a form equivalent to the FND. A simple illustrative example is given with the next two figures. Figure 5 shows a network composed by four blocks that represent (from left to right and from top to bottom): the reading of the incoming video bitstream, an HEVC decoder, the unit that displays the decoded bitstream, and the generator of the MD5 hash value. The arrows represent the data flow between these
components. Figure 6 is the representation of this network in a human-readable format by means of a XDF file.

Figure 5. Graphical view of a network with four functional blocks.

Figure 6. XML-based textual representation of the network shown in Figure 5.

The blocks or FUs of Figure 5 represent algorithms encapsulated into independent entities. Hence, this figure is an example of how the different functional parts of a decoder can be implemented in any software or hardware language “considering them as black box components”.

13
Moreover, a network can be implemented in a hierarchical way; this means a certain FU can encompass a network of FUs. This is the case of the HevcDecoder block, whose inner network is presented in Figure 7.

The network of Figure 7 is a good example to explain briefly the main information that usually appears in a XDF file: instances, connections, inputs and outputs. An instance is a request of a certain component class. A connection is a dataflow association between the output port of a FU and the input ports of other FUs. An input is the input port of the network, which is connected to one or more FUs. An output is the output port of the network, which can only be connected to the output of one FU. Therefore, the XDF file associated with Figure 7 describes ten instances, many connections, one input and five outputs.

Figure 7. Graphical view of the network of the HevcDecoder component.

2.3.3. FU

FUs are defined with textual specifications and sometimes with the corresponding code implementation written in RVC-CAL. A textual specification of a FU gives the FU name, a short functionality description, the original standard and profile it comes from, and the properties of its input and output data. Examples of how textual
specifications are implemented in code using the RVC-CAL language are presented in
next section.

2.4. CAL

RVC-CAL is simply a subset [13] of the CAL or CAL Actor Language [14] [15], a
high-level programming language to write dataflow actors. RVC-CAL is employed to
specify the internal functionality of the FUs as actor entities. As can be seen in Figure
8, an actor has usually an internal state, parameters, actions, input ports and output
ports. This section pretends to give a basic introduction to CAL, so didactic examples
are exposed. To delve into CAL, references [13] [14] [15] are highly advised.

In a first approach to the RVC-CAL usage, in the declaration of the actor their
input/output ports and instantiation parameters must be declared as:

```
actor ActorName [<Type parameters>] (<Actor parameters>)
InputPortsList ==> OutputPortsList
```

An example is shown in next piece of code:

```
actor EXAMPLE_ACTOR() In1, In2, In3 ==> Out
(…)
end
```

The processing of a FU is described in the body of its actor, accordingly to a set
of atomic tasks called actions. An action expresses a relation between the state of an
actor and input tokens, and the successor state of the actor and output tokens. The
execution of an action is called firing an action and during that, the actor consumes input tokens, produces output tokens and updates state variables. This happens for all actors and the procedure is the same for any firing cycle.

Because the actors can have one or more actions, non-deterministic execution behaviours can happen if the execution conditions are not controlled for all possible situations. The following code exemplifies this situation in which both actions can be fired at the same time, resulting that the selection of the action to be fired is ambiguous.

```
"<ActionLabel>: action <InputPortLabel>[tokens] ==><OutputPortLabel>[tokens] "

SelectInput1: action In1:[tokenAtInput1] ==> [output] end
SelectInput2: action In2:[tokenAtInput2] ==> [output] end
```

To avoid this ambiguity, the execution of an action will be performed not only when enough input data is available but also if a guard condition is present and it is true. The guard condition can refer to an input token value, the state of the actor or both. An example is shown in the next code, which is a selector of actions.

```
actor Select() In1, In2, In3 ==> Out:
  action1: action In1:[tokenAtInput1] ==>[output]
      guard selector end
  action2: action In2:[tokenAtInput2] ==>[output]
      guard not selector end
end
```

In this code, depending on the internal state of the actor (materialized by the “selector” variable), the next action to perform is determined. An action guarded by state conditions can be fired/triggered once these conditions are met and the action has enough token on its input ports. These aspects form a part of the normative CAL programming language, where it is specified that the regulation of action firings must be managed by a Finite States Machine (FSM), input tokens, guards and priority assignments.

An action can contain a description that defines the series of statements to perform. Among other statements, internal action/actor variables can be read and written, and functions and procedures can be called. These functions and procedures are identified using labels and can be declared within the actor or imported from other packages.

The next example shows several actions of an actor, called “skip”, “start” and “done”. These actions manipulate some variables already declared. Actions “skip” and “start” wait for tokens on the same port, “In3”. On the one hand, the action “skip” is fired
if and only if the first token available at “In3” has strictly a negative value. On the other hand, action “start” is fired for any value equal or greater than 0. Action “start” modifies later the variable “add_buf”. The “done” action is fired only when the value of the variable “count” is 64.

```plaintext
skip: action In3:[i] ==> //Wait for data i in In3 and i<0
guard i < 0 end

start: action In3:[i] ==> // Wait for data i in In3 and i>=0
guard i >= 0 do
    add_buf := i;
end

done: action ==> //Wait for count equal to 64
guard count = 64
(...)```

The following didactic example presents the FSM of an actor and priority declarations. The FSM works as a scheduler, defining the initial state and a number of transitions between states, each tagged with one or more action labels (names of the actions). Priority declarations specify also series of firing rules for the actions. In the example, the initial state is denominated “rest” and when the action “skip” is executed the state remains the same, but instead, when action “start” is executed the state changes to the “read” state. To solve the ambiguity of this FSM, a priority rule is set; the “skip” action has higher priority than the “start” action.

```plaintext
schedule fsm rest :
    rest (skip) --> rest ;
    rest (start) --> read ;
(...)
end

priority
    skip > start ;
end```

The last detail that is introduced here about CAL consists on other statements that an RVC-CAL file may contain before its actor declaration. One can find the package it pertains, imported utilities and comments describing the functionality of the actor. When working with decoders, the package information is employed not only to gather and arrange video tools, but also to indicate the standard in which the actor is placed. An example is shown in the following piece of code:
package devel.org.sc29.wg11.mpegh.part2.inter;

/**
 * Actor Framedelay.
 */

import std.util.Math.*;
import common.CommonConstantHEVC.*;
import devel.org.sc29.wg11.mpegh.part2.common.CommonConstantHEVC.*;

actor Framedelay()
    uint(size=16) PictSize, uint(size=8) Block_in =>
    uint(size=8) Block_out:
    (.....)
3 High Efficiency Video Coding
3.1. Video coding standards evolution

In order to know why the standard High Efficiency Video Coding (HEVC) has appeared, a brief summary of the evolution of the principal video coding standards has been considered appropriate. To acquire detailed information about this history the reader may consult [16] [17] [18] and [19].

There are three global standardization bodies in the field of video coding: the International Standardization Organization (ISO), the International Electrotechnical Commission (IEC) and the ITU (International Telecommunication Union). These bodies have specific groups of experts that dealt with multimedia coding; being the most relevant the MPEG of both ISO and IEC, and the Video Coding Experts Group (VCEG) of the ITU. During the last decade these two groups have joined efforts by forming the Joint Collaborative Team on Video Coding (JCT-VC), also called simply Joint Video Team (JVT). The last video coding standards are result from the JVT and as a consequence, each standard of the JVT have different names depending on the organization referred; in the case of HEVC: ISO/IEC 23008-2 MPEG-H Part 2 and ITU-T H.265.

Figure 9 is a chronogram of video coding standards. This chronogram indicates approximately their position in time. After a study of the standards that appear in the chronogram, the ideas that should be extracted are the following:

- There are many standards and some of them are very similar (all of them use the hybrid-coding scheme).
- Video compression standards usually emerge to adapt to the current market demands, to face the competency, and to overtake the features of the existing ones.
- The time between the rise of an idea for a new video compression standard and its implementation is much shorter nowadays.

HEVC emerged as an attempt to reach a new standard which delivers equal or better visual quality than H.264/MPEG-4 AVC high profile, but with only half of the bit rate required. This attempt was motivated by the limited capacity of telecommunication networks regarding the growth of video material with high definition (HD) and the existence of advances in some technologies (e.g. parallel processing). The objective
3. High Efficiency Video Coding.

has been accomplished at the cost of an increase of two to ten times in computational complexity [20].

Figure 9. Chronogram with relevant video coding standards.

### 3.2. Features of HEVC

HEVC is an asymmetric codec that follows the typical block-based hybrid-coding scheme of video codecs. Asymmetric codec means that the encoder is more complex than the decoder and only the bitstream between these communication endpoints is specified (the decoder has to be compliant with the bitstream in order to decode it well). Hybrid-codecs are those codecs that compress the video data by reducing different types of redundancy that are present in any sequence of images. Since explain in detail the standard is not fruitful for the MT purposes and there is enough information in reference literature [21] [22], here it is offered only an overall synopsis of HEVC.

The most relevant features that difference the video coding layer of HEVC from predecessor standards can be outlined with the support of the block diagrams of a compliant encoder and decoder. In Figure 10, the reader can see the main functional blocks of the encoder and the main data that are employed with the aim of shape the coded bitstream (decoder modeling elements are shaded in light gray). In Figure 11, the same for a simplified decoder is shown.
3. High Efficiency Video Coding.

Figure 10. Typical block diagram of a HEVC encoder [21].

Figure 11. Typical block diagram of a HEVC decoder [23].
Before focusing on the blocks of the decoder, it is worth noting that to provide higher high-quality compression and to ease parallel processing, HEVC does the picture partitioning getting away from standards with the classic macroblock approach. The picture partitioning begins with the splitting in Coding Tree Units (CTUs), whose size is selected by the encoder and each consists on a luma Coding Tree Block (CTB), several chroma CTBs depending on the sampling schema used and syntax elements. The rest of main concepts related with picture partitioning are summarized in Table 1. Notice how the recursive partitioning employs different sized information blocks in order to feed the different blocks of the codec (e.g. PUs for prediction, TUs for transform, scaling and quantization processes). Notice also, the decision whether to code a picture area using intra or inter-prediction is made at the CU level. Related with the picture partitioning, HEVC uses different techniques that enable a better parallel processing (like Wavefront Parallel Processing, WPP) and correct the drawbacks of this partitioning (such as DF and SAO filters).

Table 1. Picture partitioning concepts in HEVC.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Information about the concept.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTU</td>
<td>• As large 64x64, square</td>
</tr>
<tr>
<td>(Coding Tree Unit)</td>
<td>• It is recursively quarter-size split to form CUs</td>
</tr>
<tr>
<td>CU</td>
<td>• Basic unit for intra- and intercoding.</td>
</tr>
<tr>
<td>(Coding unit)</td>
<td>• As large as CTU, as small as 8x8</td>
</tr>
<tr>
<td></td>
<td>• It is split only one time to form PUs.</td>
</tr>
<tr>
<td>PU</td>
<td>• Basic unit for prediction.</td>
</tr>
<tr>
<td>(Prediction unit)</td>
<td>• As larger as CU, as small as 4x4</td>
</tr>
<tr>
<td></td>
<td>• Types: symmetric (intra- and interpdcition) and asymmetric (interprediction).</td>
</tr>
<tr>
<td>TU</td>
<td>• Basic unit for transform and quantization processes.</td>
</tr>
<tr>
<td>(Transform unit)</td>
<td>• Size depends on PU size and ranges from 32x32 to 4x4.</td>
</tr>
<tr>
<td>Slices</td>
<td>• To support parallel encoding/decoding and error resilience.</td>
</tr>
<tr>
<td></td>
<td>• Each slice can be subdivided into smaller slices (entropy slices)</td>
</tr>
<tr>
<td>Tiles</td>
<td>• To support parallel encoding/decoding and error resilience.</td>
</tr>
<tr>
<td></td>
<td>• In each tile, CTUs are processed in a raster scan order.</td>
</tr>
<tr>
<td></td>
<td>• In each picture, tiles are processed in a raster scan order.</td>
</tr>
</tbody>
</table>

With this in mind and because this MT deals with HEVC decoders, the functional blocks of Figure 11 are explained further but briefly next:

- Entropy decoder: decodes the syntax elements within the incoming video stream, most of them using context adaptive binary arithmetic coding (CABAC). The structure of the video stream is based in network
abstraction layer (NAL) units, which are logical data packets that carry and identify the associated payload data.

- Inverse quantization and transform (IQ/IT): after the residual data is obtained from the entropy decoder, it is dequantized and inverse transformed with transforms similar to the inverse DCT. These operations are carried out at a TU level.

- Picture reconstruction: for each decoded CTU, the reconstructed CTU is obtained by adding the residual data plus the intra/inter-prediction.

- Intra-prediction: for intra decoded CUs, a prediction is calculated based on the boundary pixels belonging to previously decoded neighboring CUs. It is the same type of intra-prediction used in H.264 but with more modes (35 in total: 33 directional, plus planar and flat).

- Inter-prediction: for inter decoded CUs, a prediction is calculated based on the previously decoded pictures that are stored in the reference frames buffer. Inter prediction is carried out at a PU level. Like in H.264, unidirectional, bidirectional and weighted prediction may be used. Respect to H.264, it introduces an advanced method for motion vector prediction, provides a merge mode for motion vectors, improves the skip mode, uses also quarter-sample precision for motion compensation and employs larger filters for the interpolation of fractional-sample positions.

- Deblocking filter (DF) and Sampling Adaptive Offset (SAO) filters: their commitment consists on improving the reconstructed pictures, avoiding problems caused by the decoding process (e.g. artifacts due to the partitioning). The DF is similar to the one of H.264, but has been improved to reduce the complexity and support parallel processing. After the DF, the SAO classifies the reconstructed pel into different categories and adds an offset to each pel based on its category.

### 3.3. HEVC in RVC

There is an open tool, called Open RVC-CAL Compiler (Orcc) [24], which enables the development of RVC-based decoders. The applications developed
together with this tool are publicly available and the video decoders used in this MT come from this set of applications. Currently, the Orcc project comprises RVC-based implementations of several video coding standards:

- MPEG-2: Main Profile (MP).
- MPEG-4 part 2: Simple Profile (SP), Advanced Simple Profile (ASP).
- MPEG-4 part 10: Constrained Baseline Profile (CBP), Progressive High Profile (PHP), Scalable Video Coding.
- HEVC: Main profile, Main Still Picture profile.

During this work, the focus was put on HEVC-Main profile, because few times others were used (only for introductory purposes: AVC-CBP, AVC-PHP and HEVC-Main Still Picture profile). When in this document HEVC is mentioned, it refers to HEVC Main profile.

To end this chapter, a short overview of how HEVC was implemented in RVC at date June, 13th 2013, is given. The date is indicated because HEVC is currently under development and some details can differ in the future due to unexpected modifications.

The implementation of HEVC at this date follows a hierarchical network of actors. Thank to the number of actors is reasonable (38 actors), the hierarchical network has been represented here by means of the tables from Table 2 to Table 6. Notice that when a FU has not any network inside, its implementation in RVC-CAL is stored in a file with extension “.cal”. When there is a network inside, this sub-network is stored in a file with extension “.xdf”. Notice also that the same “.cal” file can be used to instantiate many independent components (it is the case of the components of Table 6). Finally, notice that Table 2 corresponds to Figure 5 and Table 3 corresponds to Figure 7.

Table 2. Top_mpegh_part2_main.xdf. (Top network of the decoder).

<table>
<thead>
<tr>
<th>Component name</th>
<th>Associated file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>SourceTest.cal</td>
</tr>
<tr>
<td>HevcDecoder</td>
<td>HEVCDecoderInter.xdf</td>
</tr>
<tr>
<td>Display</td>
<td>DisplayYUVWithCrop.cal</td>
</tr>
<tr>
<td>MD5</td>
<td>MD5Net.xdf</td>
</tr>
</tbody>
</table>
Table 3. HEVCDecoderInter.xdf. (The decoder itself).

<table>
<thead>
<tr>
<th>Component name</th>
<th>Associated file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algo_Parser</td>
<td>Algo_Parser.cal</td>
</tr>
<tr>
<td>xIT</td>
<td>xIT.xdf</td>
</tr>
<tr>
<td>DecodingPictureBuffer</td>
<td>DecodingPictureBuffer.cal</td>
</tr>
<tr>
<td>generateInterInfo</td>
<td>GenerateInterInformation.xdf</td>
</tr>
<tr>
<td>InterPrediction</td>
<td>InterPrediction.cal</td>
</tr>
<tr>
<td>IntraPrediction</td>
<td>IntraPrediction.cal</td>
</tr>
<tr>
<td>SelectCU</td>
<td>SelectCU.cal</td>
</tr>
<tr>
<td>QpGen</td>
<td>QpGen.cal</td>
</tr>
<tr>
<td>SAO</td>
<td>SaoFilter.cal</td>
</tr>
<tr>
<td>DBFilter</td>
<td>Dbf.xdf</td>
</tr>
</tbody>
</table>

Table 4. GenerateInterInformation.xdf.

<table>
<thead>
<tr>
<th>Component name</th>
<th>Associated file</th>
</tr>
</thead>
<tbody>
<tr>
<td>MvComponentPred</td>
<td>MvComponentPred.cal</td>
</tr>
<tr>
<td>GenerateRefList</td>
<td>GenerateRefList.cal</td>
</tr>
</tbody>
</table>

Table 5. xIT.xdf.

<table>
<thead>
<tr>
<th>Component name</th>
<th>Associated file</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT_Splitter</td>
<td>IT_Splitter.cal</td>
</tr>
<tr>
<td>IT_Merger</td>
<td>IT_Merger.cal</td>
</tr>
<tr>
<td>Block_Merger</td>
<td>Block_Merge.cal</td>
</tr>
<tr>
<td>invDST4x4_1st</td>
<td>invDST4x4_1st.cal</td>
</tr>
<tr>
<td>invDST4x4_2nd</td>
<td>invDST4x4_2nd.cal</td>
</tr>
<tr>
<td>IT4x4</td>
<td>IT4x4.xdf</td>
</tr>
<tr>
<td>IT8x8</td>
<td>IT8x8.xdf</td>
</tr>
<tr>
<td>IT16x16</td>
<td>IT16x16.xdf</td>
</tr>
<tr>
<td>IT32x32</td>
<td>IT32x32.xdf</td>
</tr>
</tbody>
</table>

Table 6. IT4x4.xdf, IT8x8.xdf, IT16x16.xdf, IT32x32.xdf (for the last three, substitute 4x4 by 8x8, 16x16 and 32x32 respectively).

<table>
<thead>
<tr>
<th>Component name</th>
<th>Associated file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transpose4x4_0</td>
<td>Transpose4x4.cal</td>
</tr>
<tr>
<td>IT4x4_1d_0</td>
<td>IT4x4_1d.cal</td>
</tr>
<tr>
<td>Transpose4x4_1</td>
<td>Transpose4x4.cal</td>
</tr>
<tr>
<td>IT4x4_1d_1</td>
<td>IT4x4_1d.cal</td>
</tr>
</tbody>
</table>
Table 7. Dbf.xdf.

<table>
<thead>
<tr>
<th>Component name</th>
<th>Associated file</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenerateBs</td>
<td>GenerateBs.cal</td>
</tr>
<tr>
<td>DeblockFilt</td>
<td>DeblockingFilter.cal</td>
</tr>
</tbody>
</table>

Table 8. MD5Net.xdf

<table>
<thead>
<tr>
<th>Component name</th>
<th>Associated file</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5SplitInfo</td>
<td>MD5SplitInfo.cal</td>
</tr>
<tr>
<td>padding</td>
<td>padding.cal</td>
</tr>
<tr>
<td>shifter</td>
<td>MD5Shifter.cal</td>
</tr>
<tr>
<td>compute</td>
<td>MD5Compute.cal</td>
</tr>
</tbody>
</table>
4 Multicore platforms and the dataflow model of RVC
4. Multicore platforms and the dataflow model of RVC.

4.1. GPP systems

A general-purpose processor (GPP) is a computation system designed for a great variety of computation tasks. Exactly for that reason GPPs are widely employed in personal computers, workstations and other devices (such as latest mobile phones, videogame consoles and automotive infotainment systems). In this MT, the GPP systems that have been considered are a personal computer (PC) and a device based on the ARM family of processors.

4.1.1. PC microprocessor

The main GPP platform employed during the development of this work is a PC. This computer is a commercial model with the features described by Table 9. As will be explained in next paragraphs, for the tasks done throughout this MT, it is not recommendable to use models with worse features (other less powerful models were tested and very poor performance was obtained; making impossible the work sometimes).

Table 9. Features of the PC employed.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel® Core™ 2 Quad Q8200 @2.33GHzx4</td>
</tr>
<tr>
<td>RAM memory</td>
<td>3.9 GiB</td>
</tr>
<tr>
<td>Hard drive capacity</td>
<td>488.0 GB</td>
</tr>
<tr>
<td>Operating System (OS)</td>
<td>Ubuntu 12.04.1 LTS (32 bit)</td>
</tr>
</tbody>
</table>

The number of cores of the processor is very relevant because it influences directly the number of existing options for the distribution and parallelizing of the algorithms. For instance, this model has 4 cores and therefore up to 4 FUs might be executed concurrently.

A lot of RAM memory is important because some tools of the working environment (mostly the Eclipse-based) are very demanding in this sense. Indeed, that is the main reason because computers with worse features do not work well. Moreover, having enough RAM memory is necessary to have good performance at video decoding (particularly with HD large images).

The other features are not as relevant as the previous ones. The hard drive capacity should ensure there is enough capacity for the working material (for example,
raw video sequences or YUV files take up large sizes). Regarding the OS, the reason because Ubuntu was selected consists on other embedded systems (such as the described in next section) operates with Linux-based OS and then, having a Linux-based on the computer eases make tests and comparisons with these systems.

The PC has been a key element in the learning of the theoretical concepts of this MT. It is the platform where the software of the video decoders has been tested before start working with it. It is also the platform where distributions of FUs among several cores have been experimented. Another example is that this platform has been used sometimes during the development of DSP-based implementations, because the debugging information provided can be useful to debug issues in the development.

4.1.2. ARM device

The PandaBoard [27] is an embedded system based on the OMAP4460 [28]. The OMAP4460 consists on a multicore device; which integrates two ARM® Cortex™-A9 MPCore™ processors operating at 1.2 GHz each, a graphics accelerator, an Image Signal Processor (ISP), and RAM memory of 1 GB (see Figure 12). It is able to run an OS like “Ubuntu Release 11.0 Oneiric, Kernel Linux 3.0.0-1207-omap4, GNOME 3.2.1”. Thank to these features, the PandaBoard is an integrated board with all the technical resources needed for set up a wide range of applications.

![Figure 12. Architecture of the OMAP44x family of processors.](image-url)
As can be seen in Figure 13, the PandaBoard provides a good environment to develop multimedia embedded systems. It comprises a complete set of peripherals that are widely extended in the most modern multimedia devices: peripherals for connectivity (Ethernet, WLAN, Bluetooth, USB 2.0, etc.), audio inputs and outputs, interface for cameras, video outputs (HDMI, DVI), etc.

Figure 13. PandaBoard.

The PandaBoard was chosen in [3] to perform multicore experimentation and scheduling using several processor cores. In [3], it was proven that the parallelizing of RVC-based video decoders among several cores can provide better results than a single core implementation. Like the PC, it is a platform where distributions of FUs among several cores have been experimented. However, throughout this MT, this device is slightly employed with the purpose of establish comparisons with the results obtained with the DM6437 EVM, the DSP system employed. The PandaBoard is interesting because the performance results that it is able to achieve are similar to the attained with the most powerful DSPs; therefore, it provides a reference for the evaluation of the results obtained with these DSP systems.
4.2. DSP systems: the DM6437 EVM

Digital Signal Processors (DSPs) are microprocessors specifically designed to handle digital signal processing tasks. Hence, they are optimized for certain tasks respect to GPPs. Depending on the application they can provide lower-cost solutions with better performance, lower latency, non-specialized cooling, smaller batteries and less space. As a consequence, they are widely extended in portable embedded systems that perform intensive signal processing tasks, like mobile phones and television set-top-boxes.

Nevertheless, in some applications such as video coding, the requirements of computing power are each time higher and there is a need of more powerful optimized systems. Because the speed of the base technology is difficult to increase [29], complex multicore DSP processors have been developed in the market during latest years [30]. The complexity of these devices brings a new challenge to application developers: new methodologies are required. This is the challenge for the research community in which this MT is involved.

In order to start exploring better working methodologies for these DSP platforms, logically a DSP-based platform must be employed. Because the development of new methodologies is complex, it is preferred to do the work in simpler steps or milestones; hence, following this approach, this MT has focused the research in how to obtain a working methodology for only one core of a DSP. Later, this methodology will be extended to multicore implementations. Therefore, for this purpose, the DM6437 EVM [31] has been selected, because the unique core of its DSP is the same as the existing ones in the TMS320C6472 [32], a multicore DSP from TI with 6 cores.

The DM6437 EVM is a standalone development platform that enables users to evaluate and develop applications for the TI DaVinci™ processor family. Moreover, it provides a rich environment for video application development; although not all the available features have been employed (the focus is placed on the DSP processor rather than for example in its video, audio or Ethernet interfaces).

A block diagram of the DM6437 EVM is shown in Figure 14. For the scope of this work, it is used only as an evaluation module where RVC-based HEVC decoders and the methodology developed are evaluated for a DSP suitable for video coding, the DM6437 processor from TI. Hence, it will be highlighted here only the most relevant
features of this platform, which comprises processing, storing and debugging capabilities; for more information refer to [31].

Figure 14. DM6437 Evaluation Module block diagram [31].

Firstly, the processing is done with the TMS320DM6437 processor [33], usually called DM6437. The DM6437 is a fixed-point DSP of the family C64x+™ [34], with a core based on the VelociTi™ VLIW architectures. Within this board, it operates up to 600 MHz (4800 MIPS) and interfaces to on-board peripherals through integrated device interfaces and an 8-bit wide EMIF bus. It has application-specific hardware logic, on-chip memory and additional on-chip peripherals. Regarding on-chip memory, it has 64KB of ROM for booting and 240KB of RAM. The RAM is arranged using a two-level cache-based architecture. The Level 1 program memory/cache (L1P) consists of a 32KB memory space that can be configured as mapped memory or direct mapped cache, and the Level 1 data (L1D) consists of an 80KB memory space (a maximum of 32KB can be configured as cache). The Level 2 memory/cache (L2) consists of a 128KB memory space that is shared between program and data space, and it can be configured as mapped memory, cache, or combinations of the two.

Secondly, the DM6437 EVM has 128MB of DDR2 DRAM, 16MB of non-volatile Flash memory, 64MB of NAND Flash and 2MB of SRAM. Figure 15 shows how it is mapped the address space of the DM6437. The DDR2 memory is connected with the DM6437 by means of a dedicated EMIF bus and it is mapped at the end of the memory map. The other memories are mapped into the space with name CS2.
Thirdly, the board has an embedded JTAG emulator interface which enables to analyze and debug interactively the applications running on the board. As a remark about the emulator that has been used, it is the Blackhawk™ USB560m JTAG Emulator [35].

![Memory map of the DM6437 EVM](image)

**Figure 15. Memory map of the DM6437 EVM.**

### 4.3. Design and running of RVC applications over multicore architectures

The first main task in mapping any kind of application to a multicore DSP consists on identifying the tasks to be parallelized and select the best processing model for the application. Figure 16 exemplifies with nine identified tasks (from A to G) the two dominant models that are usually employed:

- **Master/Slave**: one core controls the work assignments on all cores.

- **Data Flow**: work flows through cores (processing stages) as in a pipeline.

In the Data Flow model, execution and control tasks are distributed among cores. Each core performs a defined task and then the resulting data is passed to another core for further processing; hence, scheduling is triggered upon data availability. RVC applications fit very well in this model; since RVC implies networks of actors and each actor is executed as soon as it has all the required data on its inputs.
4. Multicore platforms and the dataflow model of RVC.

The networks of actors of RVC fall into the category of Dataflow Process Networks (DPN). The DPN for RVC delimits a specific environment where series of deterministic processes, described as series of actors, communicate asynchronously by passing messages through unidirectional point-to-point FIFO channels.

An example diagram of a DPN with five actors and seven FIFO channels is represented in Figure 17. The actors are represented with circles and enumerated as A1, A2, A3, and so on. The FIFOs are represented with tubes and inside them the messages, also called tokens, are represented with rectangles. The tokens are discrete data packets transmitted through the FIFOs and generated or consumed by actors. The arrows indicate the flow of tokens, making possible recognize which actors generate and consume each stream of data. For instance, in Figure 17 the actor A3 consumes tokens that are produced by A1, A2 and A4; and it produces tokens that are consumed by A5.

Figure 16. Example of the Master/Slave model (left) and the Data Flow model (right) [29].

Figure 17. Example of an RVC Dataflow Process Network.
To be executed in a processor with any number of cores, scheduling policies are required for this DPN model. Consider first that only one core is being used. Two main levels of scheduling are required, one for determine which actor must be executed and other for determine which action has to perform the actor selected to be executed. These two levels of scheduling are denominated respectively, actor scheduler and action scheduler. They will be explained next before introducing the extension to multicore architectures.

Firstly, the action scheduler is a scheduler completely internal to an actor. It does the scheduling of actions taking into account their execution conditions. These conditions comprise inner guard conditions like state variables of the actor, the availability in the input ports of the actor for the required tokens of the actions, and a list of priorities that arrange the execution of actions in the case of two or more actions are prepared to be fired. Therefore, the action firings represent a mapping of input tokens to output tokens applied repeatedly and sequentially in one or more data streams.

It is worth noting that if there are not enough input tokens for an action, this action does not demand execution time until all its conditions will be fulfilled; and if this condition is never met, the output tokens of this action will never be produced. This effect can propagate throughout the network, so the actor scheduler must implement countermeasures that avoid the “starving” of actors.

Secondly, the actor scheduler is a scheduler that in function of the state of the network, decides which actor takes the processor and which not. It abstracts completely from the management of actions and performs the control of the network using two policies, Round-Robin and Data-driven/Data-Demand (see Figure 18):

- Round-Robin scheduling strategy establishes the execution of actors accordingly to their firing conditions. Current actor execution will continue until the firing conditions of other actor will be satisfied.

- Data-driven/Data-Demand strategy manages the execution of actors based on the state of the FIFOs and the demands of tokens by actors. The principles of this strategy are basically two. On the one side, if some actor requires input data, predecessor actor will be called to execution. On the other side, if a FIFO is complete, the consumer actor of these tokens will be called to execution.
4. Multicore platforms and the dataflow model of RVC.

The way in which the actor scheduler and the actor calls are implemented entails an important design decision. For example, taking into account that working with only one processor core, there is no chance for parallelization, consider two design decisions. The first alternative is based on the assignment of separated execution threads for the scheduler and each actor, allowing their execution in a concurrent way, based on a time-sharing strategy that is controlled by the OS scheduler accordingly to its scheduling policy and the assigned priorities (the actor scheduler must have higher priority than the actors). The second alternative is based on a process where the actor scheduler consults iteratively the actors and the depending on the status reported by them, it decides which actor is launched.

The second alternative is the employed by the Orcc tool that will be introduced in next chapter. In this implementation, the scheduler and all the actors of the application are associated to the same execution thread, so there is not any concurrence among actors. Considering an example with Figure 17, the actor scheduler will ask first all actors (A1, A2, etc.) about their current status. Then, with the information provided, it will select for instance A1, which acquires control over the processor and after finish or get blocked, returns the control to the actor scheduler, which repeats the process. This means that actors A1 and A5 cannot run concurrently.

In the case of working with several cores, parallelization (and therefore concurrence) is possible. The set of actors that implements all the functionality of the application needs to be split in order to distribute the actors among cores. Some ways in which this mapping distribution can be performed may retrieve important synergies for the application efficiency.

The implementation of an application distributed in different cores is performed following the design decision of one thread per core (each thread comprises all the actors assigned to its core). The behaviour of a thread remains similar to the recently explained for a single core implementation, but however, the core threads need to
implement additional mechanisms in order to support the distributed implementation (the core threads interact interchanging information).

Because the Data-driven/Data-Demand policy is employed, it is imposed the need to interchange scheduling information between the actor schedulers of different cores. This is due to the possibility in execution time that a predecessor or successor actor will be listed on a different core from the one in which the current actor consumes/generates data.

To implement these information interchanges, scheduling-FIFOs shared between cores are the mechanism chosen. It is important not to confound this flow of scheduling information with the data-FIFO information flow among actors; the Figure 19 represents the differences. This lock-free inter-core communication is carried out without synchronization elements like semaphores. Instead, shared memory is employed on a “single producer - single consumer” pattern, in which read/write pointers manages information flow.

![Figure 19. Example diagram of the different communication flows in a dual core architecture [3].](image)

In this sense, different network topologies for the communication of inter-core scheduling data have been proposed. The most relevant are the ring and mesh topologies (see Figure 20). In ring topology the number of scheduling-FIFOs is equal to the number of cores and direct communication between some schedulers might not be possible. In mesh topology a bidirectional communication channel is used between each couple of schedulers, allowing direct communication between all core schedulers.
In this topology an exponential number of scheduling-FIFOs is required accordingly to the number of cores under use. Consequently, the mesh topology is more flexible than the ring topology, but it is more memory demanding.

![Possible topologies of scheduling flow communications ring (left) and mesh (right) [3].](image)

### 4.4. Reason of running an RVC decoder on a single DSP core

The final aim of the research in which this MT is involved, consists on the evaluation of the concepts explained in the previous section, by means of multicore DSPs. The concepts presented are very relevant because they could be used in multicore DSP architectures to provide applications with better performance, something that is very demanded in video coding applications.

For instance, the features of the multicore platform and the design decisions taken, determine several important aspects like the level of parallelization possible, the simplicity of the software architecture, the amount of required resources, the inter-core bandwidth needed, the overhead introduced by the schedulers and the management of threads, the likelihood of incidents such as bottlenecks, the level of granularity achievable, etc. The developers of any kind of application based on a dataflow model among multiple cores should look after these aspects.

In the field of video coding, RVC allows start working over multicore platforms and exploiting the parallelization opportunities. However, the development of RVC decoders using multicore DSPs is currently a challenge, because there are not automated methodologies that help to generate the desired application more directly.
Although the multicore options cannot be experimented using the DM6437 (because it is not a multicore DSP), the fact that its core is the same as the existing ones in some multicore DSPs (like the TMS320C6472), makes it suitable for starting the development of the automated methodologies desired. The main reason is that develop a methodology for one core is much simpler than developing one that deals with several cores at the same time. Moreover, a methodology based on the common element of similar architectures but with different number of cores, may be easy to reuse and extend. For example, it is expected that the task of perform the add-ons for a specific multicore implementation (like the placement of inter-core FIFOs for scheduling and data) will be more automatic, if a suitable methodology is already known for one core and all cores have the same software design.

Therefore, given that this MT is an initial work within this research line, a methodology that allows running an RVC decoder on a single DSP core has been developed. The core is the provided by the TMS320DM6437 processor.
5 Software development tools and methodology
Several tools have been employed during this work. This chapter aims to introduce them and to explain which working methodology should be followed for obtain implementations of decoders that run on specific devices.

5.1. Development tools

5.1.1. Eclipse

Eclipse [36] is an integrated development environment (IDE). It is free, open source and cross-platform (operates over multiple OS such as Windows or Linux-based distributions). It is designed to be an all-purpose IDE.

Eclipse comprises a base workspace which can be extended and customized by adding plug-ins. These plug-ins are very useful because they enrich the development environment by providing the required functionalities with a seamless integration. For example, tools like Orcc and Code Composer Studio (described in next sections) work as a plug-ins on Eclipse. Furthermore, it is possible working at the same time with several Eclipse IDEs with different plug-ins, something that is very useful when working for instance with the previous plug-ins.

There are many versions of the software core of Eclipse, as can be viewed on Table 10. In this work, the release called Indigo has been chosen due to its well-known stability and compatibility with the other tools employed.

<table>
<thead>
<tr>
<th>Codename</th>
<th>Platform version</th>
<th>Date</th>
<th>Codename</th>
<th>Platform version</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galileo</td>
<td>3.5</td>
<td>June 24, 2009</td>
<td>Juno</td>
<td>4.2</td>
<td>June 27, 2012</td>
</tr>
<tr>
<td>Helios</td>
<td>3.6</td>
<td>June 23, 2010</td>
<td>Kepler</td>
<td>4.3</td>
<td>June 26, 2013 (planned)</td>
</tr>
<tr>
<td>Indigo</td>
<td>3.7</td>
<td>June 22, 2011</td>
<td>Luna</td>
<td>4.4</td>
<td>June 25, 2014 (planned)</td>
</tr>
</tbody>
</table>

5.1.2. Open RVC-CAL Compiler

The Open RVC-CAL Compiler [24] (Orcc hereafter) is a collection of tools that provide developers with a compiler infrastructure which allows generate solutions for several languages from RVC-CAL actors and XDF networks. In other words, it is the tool designed for the creation, edition, transformation, analysis and debug of ADMs. This infrastructure works as a plug-in for Eclipse and it is formed by many tools.
The Graphiti [25] tool is a network editor, which serves to define the connections between FUs and manages this information using XDF files. The XText [26] tool is a text editor, which is used to implement actors using RVC-CAL language.

The developers of decoders interact mainly with these two tools, but there are other components that perform the rest of the generation process. For instance, there are simulators, scheduling analyzers, and software and hardware code generators. The current simulators are based on the OpenDF and Ptolemy II projects [11]. Regarding to the code generators some examples are Cal2C and Cal2HDL; briefly, Cal2C serves to transform RVC-CAL code into C code.

For the purposes of this work, Figure 21 summarizes what is interesting to know about the usual operation of Orcc, with a video decoder as the application under development.

Firstly, the decoder is designed defining the algorithm of each involved actor using RVC-CAL and building the XDF network that connects these actors. This design establishes the “front-end” of the decoder; this concept means the complete representation of the decoder in Orcc.

Secondly, the front-end is converted immediately into an Intermediate Representation (IR). An IR is the language of an abstract machine designed to aid in the analysis of computer programs. The IR allows the compiler system of Orcc to be targeted by different source languages and support the generation of solutions for different target architectures. This last aspect is used to generate decoder solutions as a set of source files written in a predefined language; this set of files receives the name “back-end”. Along this work, C back-ends are employed.
5. Software development tools and methodology.

5.1.3. CMake

CMake [37] is a family of tools designed to build, test and package software in an automatic way. CMake generates native makefiles and workspaces that can be used in a great variety of compiler environments, like the Unix makefiles that work with GCC (GNU Compiler Collection) or some integrated development environments such as Visual Studio, Code::Blocks or Eclipse CDT.

To do that, CMake receives as input a set of configuration files, which have CMakeLists.txt as name and are plain text because this format ensures platform independency. These configuration files follow a syntax consisting of comments, commands and white spaces. The commands operate with those source files and file system directories that are specified. The aspect of a CMakeLists.txt file is given in Figure 22.

CMake is free, open source, cross-platform and easier to use than other comparable tools like the GNU build system (Autotools). It may be employed by means of three alternatives: the command line interface (CLI) of the operating system, a more suitable graphical user interface (GUI), or as an Eclipse plug-in. Throughout this work, CMake 2.8.10 was used by means of the GUI option.

![Figure 22. Example of the CMakeLists.txt at the top of the backend.](image-url)
5. Software development tools and methodology.

5.1.4. **Code Composer Studio**

Code Composer Studio [38] (hereafter CCS) is an IDE for Texas Instruments embedded processor families. It comprises a suite of tools that includes compilers for each device, source code editor, project build environment, debugger, profiler, simulators and real-time operating systems; just to mention a few. In this work, it is used for develop and debug applications for DSP targets.

The version recommended for work is the 5.4, because previous versions contains bugs or are too old. Since version 4, CCS works over an Eclipse environment. Both family of versions 4 and 5 allow work with DSP/BIOS and SYS/BIOS, the specific operating systems developed by TI to operate on its manufactured devices. DSP/BIOS is older than SYS/BIOS and as will be justified in Chapter 6, it is the used in this MT.

5.1.5. **Code::Blocks, Visual Studio or Eclipse CDT**

Code::Blocks [39] is a free, open source and cross-platform IDE. Like Eclipse, it is extensible with plug-ins and supports multiple compilers; but instead, it is mainly focused on C and C++ projects. In this work, this IDE is employed to modify and build C projects for GPP targets (the personal computer).

Existing alternatives to this IDE are Visual Studio [40] and Eclipse CDT [41]. None of them are perfect candidates because of the appearing of a couple of problems when using them. On the one hand, there is the suitability for different operating systems (Windows and Ubuntu based). This concern makes Visual Studio inadequate, which does not work properly in Ubuntu. On the other hand, there is the required performance and compatibility with other tools employed during this work. Eclipse could be overcharged when working with big workspaces or when loading many heavy plug-ins. Moreover, the proofs over the Eclipse CDT4 version, the unique that CMake is able to manage, show some unsolved errors that difficult severely the work (for example, the behavior of the indexer is erroneous, inexisten errors are reported, etc.).

5.1.6. **Other tools**

Other tools were used during the development of this MT. Some of them are employed along the developed methodology for certain less relevant tasks. Among these tools there are for instance the tools shipped with the operating system (text editors, the shell, compilers, tools that permit compare content of folders and files, etc.), software for control and management of versions (PySVN Workbench [42]), etc.
5.2. Working methodology

The working methodology is the sequence of tasks that is needed to do in order to achieve a specific purpose. In this MT, the purpose consists on obtaining GPP and DSP implementations of decoders. Actually, the objective is to extend an existing methodology for GPPs, with the aim of make it suitable for multicore DSPs.

For clarity reasons, the explanation of the working methodology has been split in two; although first stages are similar. Firstly, the one for GPP targets is presented and summarized by Figure 23. Secondly, the one for DSP targets is presented as from the previous methodology and it is summarized by Figure 24. Furthermore, the “Annex III: Installation guide and tutorial” presents training resources with the aim of help the reader to start working with this methodology.

The different stages of the working methodology are the turquoise and yellow boxes of Figure 23 and Figure 24. They are called stages because imply the use of a development tool. Each one of these stages is associated to a certain part of the methodology. These parts have been called “edition levels”, because the nature of the files edited is totally different. The developed methodology has therefore two edition levels.

On the first edition level, there is the Orcc infrastructure. Here, the designed decoder based on the RVC specification is implemented: the structure of the decoder is edited with Graphiti and the functionality of the actors is edited with a CAL text editor such as XText. Orcc takes the information created with these two tools and generates the backend according to the settings programmed by the user. As shown in the figures, C backends have been employed during this work.

In Orcc, there is the option of generating an XCF file, which is an XML-based file that indicates how the different actors must be distributed among the cores of a processor. Currently, this XCF file can be used only in GPP targets, but it is expected that in the future it will be used to generate automatic mappings on multicore DSP targets. In “Annex II: Structure of the C backend” there is an example of XCF file in which all actors are mapped to the same core.

In the work here developed, because the decoding applications are already developed, few edition tasks are performed at the first level of edition. Firstly, these decoding applications [43] must be imported into the Eclipse’s workspace in order to
work with them using the Orcc infrastructure. Secondly, when the edition is finished, the Orcc compilation is configured and then Orcc generates the C backend and the XCF file for the application specified. Up to here, the methodology is the same for GPPs and DSPs.

<table>
<thead>
<tr>
<th>First level of edition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoder designs based on</td>
</tr>
<tr>
<td>the PVC specification</td>
</tr>
<tr>
<td>are implemented using</td>
</tr>
<tr>
<td>these plugins for Eclipse.</td>
</tr>
<tr>
<td><strong>Graphite</strong></td>
</tr>
<tr>
<td><strong>CAL Editor</strong></td>
</tr>
<tr>
<td><strong>ORCC</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partial result.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The ORCC compilation</td>
</tr>
<tr>
<td>provides a backend in the</td>
</tr>
<tr>
<td>chosen language and a</td>
</tr>
<tr>
<td>static distribution of</td>
</tr>
<tr>
<td>actors among the processor</td>
</tr>
<tr>
<td>cores.</td>
</tr>
<tr>
<td><strong>Backend</strong></td>
</tr>
<tr>
<td><strong>xcf</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Second level of edition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The solution is configured</td>
</tr>
<tr>
<td>to an IDE (or alternative</td>
</tr>
<tr>
<td>environment) capable to</td>
</tr>
<tr>
<td>perform edition and</td>
</tr>
<tr>
<td>building tasks.</td>
</tr>
<tr>
<td>At this level, the solution</td>
</tr>
<tr>
<td>is customized editing the</td>
</tr>
<tr>
<td>decoder source files and</td>
</tr>
<tr>
<td>the distribution file.</td>
</tr>
<tr>
<td>The compiler employed</td>
</tr>
<tr>
<td>must be suitable for the</td>
</tr>
<tr>
<td>target platform.</td>
</tr>
<tr>
<td><strong>CMake</strong></td>
</tr>
<tr>
<td><strong>Project</strong></td>
</tr>
<tr>
<td><strong>IDE or Editor+Builder</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Final result.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The result of the building</td>
</tr>
<tr>
<td>process is the executable</td>
</tr>
<tr>
<td>that is able to decode</td>
</tr>
<tr>
<td>video sequences on the</td>
</tr>
<tr>
<td>multicore target processor</td>
</tr>
<tr>
<td>according to the</td>
</tr>
<tr>
<td>distribution file.</td>
</tr>
<tr>
<td><strong>Executable</strong></td>
</tr>
<tr>
<td><strong>xcf</strong></td>
</tr>
</tbody>
</table>

Figure 23. Working methodology for GPP targets.
5. Software development tools and methodology.

On the second edition level, as can be seen in Figure 23 and Figure 24, specific tools must be used depending on the target architecture and the tasks required.

In the case of a GPP, the backend has to be processed by CMake to build the proper environment for the tool that builds definitely the application. The commitment of CMake is therefore create a version of the backend (project) that can be compiled and built by the suitable tool, which can be Code::Blocks, Visual Studio or GCC depending on the supporting OS. These IDEs and common text editors are also employed to make changes in the source code of the backend, adapting it to the specific needs of
the moment. The XCF file can be edited as well with a text editor. Finally, a GPP runs the final application using optionally the XCF file (see Figure 25).

![Diagram of running video decoder in a GPP target]

**Figure 25. Running of a video decoder in a GPP target.**

In the case of a DSP, Code Composer Studio is placed too in the second level of edition; it is necessary because it is the tool that permits compile, build, debug, etc. applications for the DSP. Because CMake does not generate projects for CCS, the source files of the application need to be imported into the projects of the CCS workspace. This action can be performed from the files of the backend generated by Orcc or their equivalent after use CMake (see respectively points 1 and 2 in Figure 24). Then, CCS allows edit the source files fitting them for the DSP target. Other IDEs may be also employed for some tasks (see point 3); for example, to observe in a PC the result of change the value of specific flags or analyze the execution of the decoder generated by Orcc.

Currently, the XCF file is not employed when building decoders for DSP (see point 4). However, it is expected to take advantage of it when developing the final methodology for DSP-based multicore targets. Finally, with CCS it is generated a “*.out” file that comprises the DSP application and configuration files for the DSP target, allowing to run the solution in this kind of target (Figure 26).

![Diagram of running video decoder in a DSP target]

**Figure 26. Running of a video decoder in a DSP target.**
6 Procedure for software migration
6. Procedure for software migration.

6.1. Overview

In order to run the same decoder solutions that were created using the working methodology for GPP targets, in a certain DSP platform like the DM6437 EVM, it results necessary to perform a software migration procedure. The intrinsic differences between these two kinds of platforms impose this need of carry out the proper changes. For example, when migrating to the DSP platform, the most relevant modifications are usually due to its limited capabilities (for instance, memory and/or speed constraints) and the features of the software environment it provides (specific operating system, libraries that are equivalent but customized, restrictions of the cross compiler, additional configuration files, etc.).

The analysis of the different steps of this software migration procedure might serve as base work to build a methodology for DSP targets more automated. With this purpose, along this chapter the steps that have been done in order to run on the DM6437 EVM board a decoder obtained for PC are described. Given that these steps should be performed in a very ordered way, Figure 27 gives an overview of the process and proposes a working guide.

![Figure 27. Overview of the working methodology to migrate the decoder.](image)
6.2. Inspection of the starting point

Before starting the software migration procedure, it is highly recommendable inspect the initial state of the process. There can be many reasons for that, but the most relevant are:

- Discover the dependencies between files and projects within Orcc in order to know where changes must be made at the first edition level. As exemplified in next section, the size of certain buffers can be determined using information from several RVC-CAL files.

- The location and organization of the files in the C backend is not trivial and must be known. There are different parts involved in the C backend and have a good understanding of them is advisable. For example, it is important to know what libraries are required and why. Other instances consists on knowing the dependencies across these libraries and source files, the C file associations with the files written in RVC-CAL, the functionality of each source code file, etc.

- There can be modifications in the starting point due to updates along the process that allows obtain C backends. For example, both the RVC-CAL source files and Orcc might be modified, resulting in distinct C backend versions. This might produce the need of redefining the working sequence presented in Figure 27. Obviously, merge files of different backend versions is not recommended.

In “Annex II: Structure of the C backend”, the result of a typical case for this step is given: the file structure of the backend with respect to the files necessary to build the decoder (as remark, notice that the date of the backend is specified). Within the backend, there are approximately 71 files that must be managed during the migration process. Hence, the need of an automated methodology is very justified, because these files have each a large amount of lines of code. For this migration procedure, the analysis of the backend provides useful information (like for example, that the main file is located inside the folder “src” and it is Top_mpegh_part2_main.c).
6.3. Preliminary settings of parameters

At the first level of edition, some settings need to be performed due to either features of the target platform or aspects of the debugging process. These settings should be performed at this level because the RVC-CAL design determines a large amount of variables of the C backend and therefore, the action of configure most parameters of the decoder results easier and safer in this place than through the C backend files. Once edited at this level, Orcc is in charge of carrying out the rest of changes automatically.

6.3.1. Adjustment of image size

In this case, this setting is mainly due to the memory limitations of the DM6437 EVM. The DM6437 EVM consists of a 128 MB DDR2 memory which is unable to allocate the amount of memory imposed by the default parameter values. The DDR2 memory is the biggest memory on the board; hence, it is essential to reduce the demanded memory in order to run the decoder on this DSP system.

After a searching for huge memory demands inside the code, it was discovered that the maximum picture size is the aspect that consumes the highest percentage of the demanded memory. As bigger the picture size, the bigger the buffers of pictures too. This demand can be reduced because the target applications of this work are not focused on high resolution images.

This problem with the memory is caused by the big size of the variable “pictureBuffer[7][3][4352][2304]” (located at HevcDecoder_DecodingPictureBuffer.c). This size cannot be allocated, because it demands:

\[ 7 \times 3 \times 4352 \times 2304 \approx 210 \text{ MB} \] (bit depth considered as 8 bits)

This memory demand is too high with respect to the capacity of the DM6437 EVM. Therefore, a way to obtain a smaller picture buffer is required and the new size must be lower than 128 MB.

As mentioned before, the way to modify this issue should be searched first in the first level of edition. Then, a search for tags ‘pictureBuffer’ and ‘4352’ fired relevant results: ‘4352’ is referenced in the file DecodingPictureBuffer.ir (an Intermediate Representation) and ‘pictureBuffer’ in DecodingPictureBuffer.cal; both files are associated. In fact, DecodingPictureBuffer.cal is the place where it is ordered how the
size of the picture buffer must be computed, and DecodingPictureBuffer.ir is the place where the sizes has been already computed and reflected prior to generate the C backend. DecodingPictureBuffer.cal has as “qualified name” (path to the file in the Orcc workspace):

```plaintext
.../RVC/src/devel/org/sc29/wg11/mpegh/part2/inter/DecodingPictureBuffer.cal
```

DecodingPictureBuffer.cal indicates that the size of the pictureBuffer variable is computed by an operation that uses three constants: PICT_WIDTH, PICT_HEIGHT and BORDER_SIZE. In this same file, BORDER_SIZE is declared with value 128. However, the other two constants are located in the file CommonConstantHEVC.cal (see Figure 28), which has as “qualified name”:

```plaintext
.../RVC/src/devel/org/sc29/wg11/mpegh/part2/common/CommonConstantHEVC.cal
```

![Figure 28. Constants PICT_WIDTH and PICT_HEIGHT in CommonConstantHEVC.cal.](image)

CommonConstantHEVC.cal contains general constants employed to generate the HEVC-RVC solutions. For PICT_WIDTH and PICT_HEIGHT, the default values are 4096 and 2048 respectively (both, integer powers of 2). Applying the same operation to
lower values (but using also integer powers of 2), the results of Table 11 are obtained. After a trade-off decision, 1024x512 was selected as the maximum resolution allowed. Then, the final task consists on changing the values of these constants with the new ones.

Table 11. Calculation of the memory required for different buffer image sizes.

<table>
<thead>
<tr>
<th>PICT_WIDTH</th>
<th>PICT_HEIGHT</th>
<th>Memory demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>2048</td>
<td>~ 210 MB</td>
</tr>
<tr>
<td>2048</td>
<td>1024</td>
<td>~ 62 MB</td>
</tr>
<tr>
<td>1024</td>
<td>512</td>
<td>~ 20 MB</td>
</tr>
<tr>
<td>512</td>
<td>256</td>
<td>~ 8 MB</td>
</tr>
</tbody>
</table>

6.3.2. Other parameters

Change the default value of other parameters can be useful in specific implementations. For instance, for debugging purposes the file MvComponentPred.cal has flags that can be enabled to debug the implementation of the inter-prediction. As remark, sometimes, it is possible enable/disable directly some debugging flags in the C backend; for example DEBUG_BITSTREAM of HevcDecoder_Algo_Parser.c.

6.4. Generation of the C backend

After the prior settings at the first level of edition, the C backend can be generated using Orcc. A couple of comments are required about this process. In the configuration window for a new Orcc compilation (see "Annex III: Installation guide and tutorial"), firstly, a FIFO size of at least 8192 must be set because otherwise, solutions do not work. Secondly, a single-core decoder solution must be generated. The reason for this last comment consists on the fact that the migration of an application with one thread might be easier than the one of a multithreading application, since the implementation of threads usually depend on the OS. As was mentioned earlier, Orcc solutions work assigning a thread to each core and a list of actors to each thread. Moreover, this very first DSP implementation works over one core, so it has not sense to generate a multithreading application that later will be converted in a single-thread application.

The result of this step must be a C backend similar to the presented in "Annex II: Structure of the C backend".
6.5. Create and configure the DSP project

CCS is the tool that permits build and debug applications for DSPs of Texas Instruments. Although it is based on the Eclipse framework since version 4, it is not possible to generate directly a new configured workspace using CMake. Then, it obliges to manually create a new workspace and set up the involved projects.

Once the workspace is created, it must be established the following CCS projects (see Figure 29):

- **RVC**: it is the main project because it must contain at least the specific source code files of the decoder, located at the “src” folder of the backend, and also, the configuration files for the target platform. The final executable, a “*.out” file, will be generated within this project.

- **ORCC**: it comprises the library of Orcc, which has the files that implement the management of actors, schedulers and FIFOs, among others. This library is located at the “libs/orcc” folder of the backend.

- **ROXML**: it comprises the “libroxml” library [44], which is a multi-platform library written in C for XML parsing targeting embedded systems. It is located at the “libs/roxml” folder.

- **PTE**: it is dedicated to another library that will be introduced in next section. It provides an API for threads.

Notice there is no need to include other files existing on the backend. For example, the SDL (Simple DirectMedia Layer, [45]) library is discarded because it is not suitable for the target platform.

The dialog window that appears when creating each project (see Figure 30) is filled following the indications of Table 12. This procedure configures the working environment. Notice that the columns “Family&Variant” and “Connection” depend on the specific target device and the connection employed; in this case a DM6437 EVM and a Blackhawk emulator, respectively. If required, these fields can be easily changed later, for example in order to use a simulator instead of an emulator. After creating the projects, the previous libraries and source files are added to the correspondent project.
7. Experiments and results.

Figure 29. CCS projects for the software migration procedure.

Figure 30. CCS window to create a new CCS project.
Table 12. Settings for each CCS project. (* Depends on the hardware employed).

<table>
<thead>
<tr>
<th>CCS project name</th>
<th>Output type</th>
<th>Family &amp; Variant *</th>
<th>Connection *</th>
<th>Project template</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVC</td>
<td>Executable</td>
<td>C6000 / DaVinci</td>
<td>&quot;Blackhawk emulator&quot;</td>
<td>Empty project</td>
</tr>
<tr>
<td>ORCC</td>
<td>Static</td>
<td>DM643x / EVDM6437</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>ROXML</td>
<td>Static</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTE</td>
<td>Static</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The next step consists on linking these projects among them and with the required Texas Instruments’s libraries. The libraries of TI are stored inside different folders below the installation path of TI software, which usually comprises a folder called “ti”. The linking process is done following Table 13, which indicates for example that the PTE project must include in its search path the route:

```
../ti/ccsv5/tools/compiler/c6000_7.4.2./include
```

Table 13. Dependencies among CCS projects and TI libraries.

<table>
<thead>
<tr>
<th>Static library or path</th>
<th>CCS projects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RVC</td>
</tr>
<tr>
<td>ORCC</td>
<td>X</td>
</tr>
<tr>
<td>ROXML</td>
<td>X</td>
</tr>
<tr>
<td>PTE</td>
<td>X</td>
</tr>
<tr>
<td>../ti/ccsv5/tools/compiler/c6000_7.4.2./include</td>
<td>X</td>
</tr>
<tr>
<td>../ti/bios_5_42_01_09/packages/ti/bios/include</td>
<td>X</td>
</tr>
<tr>
<td>../ti/bios_5_42_01_09/packages/ti/rtdx/include/c6000</td>
<td>X</td>
</tr>
</tbody>
</table>

The following task is to configure the “*.tcf” (Textual Configuration Script) file. This file configures the Real-Time OS that runs on the DSP environment and permits create the task that implements the decoder application (the task function field should have the value “_main_RVC” in order to associate with the main function of the decoder).

As shown in the block diagram of Figure 31, this task is employed in order to establish the test bench of the decoder. The “input file processing” process reads the input stream from a file, the “decoding process” is the decoder itself, and the “output file processing” process stores the decoded pictures on a file.
6.6. Adapt the thread API

In order to take advantage of the benefits of software parallelization, Orcc generates decoder solutions as multithreading applications that rely on the support OS. However, this implementation of the multithreading capability causes dependency from the OS and it is impossible execute directly decoders created for a specific OS on some others. For example, an “Ubuntu-decoder” does not work on Windows or on DSP/BIOS, but it should work on other Linux-based distributions, like FreeBSD.

The employed TI’s DSPs operate using either DSP/BIOS or SYS/BIOS, and their compilers require code written in the C or C++ languages. Due to Orcc does not generate solutions for these operating systems, the C files of the backend have to be adapted manually to one of them.

Fortunately, the C backend is constructed in a manner that does not generate big troubles. It is based on a generic API for multithreading, which is transformed later at compile time in a specific OS API. The specific OS is selected by means of directives to the compiler preprocessor, and the generic API is defined using common macro functions that have the same meaning as the ones of the specific OS (see Figure 32). All this automatic setup process is performed in the following four header files: orcc_thread.h, roxml.h, roxml-types.h, and roxml_win32_native.h.

Thank to the generic API, the adaption can be carried out simply associating the macro functions of the generic API to the suitable ones of a specific API for the DSP OS. Without matter of its use in the code, the macro functions will fulfill the DSP OS requirements. The problem consists then in how the functions of the specific API should be implemented. Again, fortunately, there is already available a suitable specific API.
This API is implemented by the PTE (PThreads-Embedded, [46]) library. PTE is an open source implementation of the POSIX API (used in Win32 [47] and Linux [48] distributions) for multithreaded applications, also known as “pthreads”; but instead, it is designed for embedded operating systems that do not provide natively a pthreads API. For the interests of this MT, it supports only DSP/BIOS (not SYS/BIOS).

PTE follows the same nomenclature and structure of the pthreads API (for instance, it declares functions in the same way and with the same usage). This library allows then the abstraction from the inner interaction details with DSP/BIOS. Consequently, adapting the thread API is practically direct. If this possibility had not existed, the method of adaption would have consisted necessarily on the substitution of sections of code related with thread management or alternatively, on the elaboration of the specific API from the scratch.

The first task to perform is therefore import the PTE library into the correspondent CCS project defined before. The second task is adding the line indicated in List 1 (call in “semaphore.h”, from PTE project, for the inclusion of the “pte_types.h” header file). The final task of this section consists on including the header “orcc_thread.h” in Top_mpegh_part2_main.c (from RVC project).

From now on, the changes carried out in each fragment of code are highlighted by means of the comment “//CHANGE!”, whose background must be colored in yellow (the yellow background means that the change corresponds to the current section, while the white background means that the change is not associated with the topic dealt in the current section). There are examples in List 1 and List 2.
List 1. Change in semaphore.h (PTE).

```c
#include "pte_types.h" //CHANGE!
```

List 2. Change in Top_mpegh_part2_main.c (RVC).

```c
#include "orcc_scheduler.h"
#include "orcc_util.h"
#include "orcc_thread.h" //CHANGE!
```

### 6.7. Enhance the main function and set input parameters

The decoder implementation for the GPP systems has a top function which is the one that instantiates the actors of the decoder. It is advisable enhance the readability of this main function in order to have a better representation of the different tasks it do; for example the declaration, initialization and instantiation of actors, or the instantiation of the scheduler of the decoder. If possible, it is recommendable to group each kind of the previous functionalities within a function; otherwise, descriptive comments are preferred. Moreover, the new structure will serve as a better starting point when implementing changes, debugging or profiling.

The other thing necessary to perform consists on creating a practical method for the introduction of input parameters into the decoder, since CCS does not provide a practical way. The parameters used throughout the tests performed are currently the bitstream to decode and the YUV file that the decoder must produce. As an example, consider that these parameters are introduced with a command that follows the next structure (the “i” option indicates the bitstream to decode and the “w” option the YUV file that must be produced):

```
<Decoder application> -i <Coded bitstream (sequence)> -w <Output YUV file>
```
List 3 shows the total change that has been carried out in function “main” of Top_mpegh_part2.c with this purpose. It is worth noting that depending on if DECANA is defined, the method for parameter introduction is automated or not. The description of the automation method is described in Chapter 7, where the experiments and their results are presented. The usage of the automated method requires the additional change shown in List 4 and the inclusion of the decana.h and decana.c files in the RVC project (these files implement a swap file explained later). The non-automated method consists basically on writing manually the parameters inside the “main” function; therefore, this method results practical only when carrying out the migration changes, not when a set of sequences test the implemented decoder.

List 3. Change in function “main” of Top_mpegh_part2_main.c (RVC).

```c
// Main
int main_RVC() {

#ifdef DECANA //Automated method
  decana_init(&decana, "C:/DecAnaPath/automate/SWAPFILE");
  printf("Decoding: %s\n", decana.bitstream);

  int argc=5;
  char* argv[5]; //argv[argc]

  char arg0[3]="-i";
  char *arg1;
  arg1=decana.bitstream;

  char arg2[3]="-w";
  char *arg3;
  arg3=decana.decoded;

  argv[1] = &arg0[0];
  argv[2] = arg1;
  argv[3] = &arg2[0];
  argv[4] = arg3;
#else //Non-automated method
  int argc=5;
  char* argv[5];
  char arg0[3]="-i";
  char arg1[150]="C:/MT/Sequences_HM_10/ld_main/
                 PartyScene_832x480_50_qp32.bin";
  char arg2[3]="-w";
  char arg3[150]="C:/MT/Decoded_HM_10/ld_main/
                 PartyScene_832x480_50_qp32.yuv";

  argv[1] = &arg0[0];
  argv[2] = &arg1[0];
  argv[3] = &arg2[0];
  argv[4] = &arg3[0];
#endif

  init_orcc(argc, argv);
  launcher();
}
```
7. Experiments and results.

```c
printf("End of decoding process !\n");
return compareErrors;
}
```

---

List 4. Change Top_mpegh_part2_main.c (RVC) to support automated tests.

```
[. . .]
#define MAX_THREAD_NB 10
#define SIZE 8192
#define CONST ULONG 4294967295
// #define PRINT_FIRINGS

//CHANGE! Adaption in order to perform automated tests
#define DECANA
#ifdef DECANA
    #include "decana.h"
    /* Global variables */
    TDecana decana;
#endif
[. . .]
```

---

6.8. Initialization of software parts

The next step consists on specifying explicitly the initialization of some sections of the code. In general these sections refer to a large number of variables, but also refer to some design utilities like threads and FIFO queues.

The initialization of the variables and the FIFO queues is required due to the compiler does not perform their compilation without having a predefined default value. These variables and queues can be initialized to any default value because the correct values are really set up at runtime and they are not used previously. As examples, some variables will be configured after read the stream of coded video, and the FIFO channels will be written and read following the Data Flow model explained in Chapter 4.

For the threads, the PTE API specifies that before use threads, they need to be created and initialized previously.
6.8.1. Initialize threads before launch the actor’s scheduler

The initialization of threads must be done after use the actor schedulers, because they are the software entities that manage the behavior of threads; remember that for each core there are an actor scheduler and a thread. The implementation carried out employs only a core and therefore there is only one actor scheduler, which is constructed within the function "launcher()" of the file Top_mpegh_part2_main.c. The changes required are highlighted in List 5

List 5. Changes in function launcher() of Top_mpegh_part2_main.c (RVC).

```c
// Initializer and launcher
static void launcher() {
    //cpu_set_t cpuset;   //CHANGE!
    thread_struct threads[MAX_THREAD_NB];
    thread_id_struct threads_id[MAX_THREAD_NB];
    // . . .
    struct mapping_s *mapping = map_actors(actors, sizeof(actors) /
               sizeof(actors[0]));
    struct scheduler_s *schedulers = (struct scheduler_s *)
        malloc(mapping->number_of_threads * sizeof(struct
               scheduler_s));
    struct waiting_s *waiting_schedulables = (struct waiting_s *)
        malloc(mapping->number_of_threads * sizeof(struct
               waiting_s));
    for(i=0; i < mapping->number_of_threads; ++i){
        sched_init(&schedulers[i], i, mapping->partitions_size[i],
                   mapping->partitions_of_actors[i], &waiting_schedulables[i],
                   &waiting_schedulables[(i+1) % mapping->number_of_threads],
                   mapping->number_of_threads, NULL);
    }
    //clear_cpu_set(cpuset);   //CHANGE!
    pthread_init();        //CHANGE!  Initialization of threads!
    // . . .
    for(i=0; i < mapping->number_of_threads ; i++){
        thread_create(threads[i], scheduler, schedulers[i],
                     threads_id[i]);
        //set_thread_affinity(cpuset, mapping->
        >Threads_affinities[i],threads[i]);   //CHANGE!
    }
    // . . .
}
```
6.8.2. Initialize actor communication FIFOs

The initialization of FIFOs can be performed also in the function “launcher()” of Top_mpegh_part2_main.c. The changes required are highlighted in List 7. Notice that in List 7, the initialization of the FIFO arrays is performed by means of the function “memset” (library string.h of the TI compiler; see the brief description provided in Table 14). Their values are established to zero.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>void *memset(void * _mem, int _ch, size_t _n);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library (header file)</td>
<td>string.h</td>
</tr>
<tr>
<td>Description</td>
<td>Sets the first “_n” bytes of the block of memory pointed by “_mem” to the specified value in “_ch”.</td>
</tr>
</tbody>
</table>

Observe also there are 72 FIFOs and the structures that must be initialized have as names “array_<number of FIFO>” and “read_inds_<number of FIFO>”. Both structures are declared using the macro “DECLARE_FIFO” defined orcc_fifo.h (see the declaration of this macro in List 6).

List 6. Declaration of the macro DECLARE_FIFO (orcc_fifo.h).

```c
// declare FIFO with a size equal to (size)
#define DECLARE_FIFO(type, size, count, readersnb)
static type array_##count[(size)];
static unsigned int read_inds_##count[readersnb] = {0};
static struct FIFO_S(type) fifo_##count = { (size), array_##count, readersnb, read_inds_##count, 0 };
```

List 7. Changes in function launcher() of Top_mpegh_part2_main.c (RVC).

```c
///////////FFFFFFF
// FIFO allocation
DECLARE_FIFO(u8, 8192, 0, 1)
DECLARE_FIFO(i16, 8192, 1, 3)
DECLARE_FIFO(i32, 8192, 2, 1)
```
DECLARE_FIFO(u8, 8192, 69, 1)
DECLARE_FIFO(u32, 8192, 70, 1)
DECLARE_FIFO(u64, 8192, 71, 1)

/////////////////////////////////////////////////
// FIFO pointer assignments
struct fifo_u8_s *Source_O = &fifo_0;
struct fifo_u8_s *HevcDecoder_Algo_Parser_byte = &fifo_0;
struct fifo_i16_s *HevcDecoder_Algo_Parser_RefPoc = &fifo_1;
struct fifo_i16_s *HevcDecoder_DecodingPictureBuffer_RpsPoc = &fifo_1;
struct fifo_i16_s *HevcDecoder_generateInfo_GenerateRefList_RefPoc = &fifo_1;
struct fifo_i16_s *HevcDecoder_generateInfo_MvComponentPred_RpsPoc = &fifo_1;
struct fifo_i32_s *HevcDecoder_Algo_Parser_DBFDisable = &fifo_2;
struct fifo_i32_s *HevcDecoder_DBFilter_DeblockFilt_DBFDisable = &fifo_2;

struct fifo_u32_s *MD5_MD5SplitInfo_REF_MD5 = &fifo_70;
struct fifo_u32_s *MD5_compute_REF = &fifo_70;
struct fifo_u64_s *MD5_MD5SplitInfo_Length = &fifo_71;
struct fifo_u64_s *MD5_padding_LENGTH = &fifo_71;

/////////////////////////////////////////////////
// Initializer and launcher

static void launcher() {
    int i;

    //cpu_set_t cpuset;   //CHANGE!
    thread_struct threads[MAX_THREAD_NB];
    thread_id_struct threads_id[MAX_THREAD_NB];

    //resher() --> mem
    memset(array_0, 0, 8192);
    memset(array_1, 0, 8192*2);
    memset(array_2, 0, 8192*4);

    //SIMILAR
    memset(array_70, 0, 8192*4);
    memset(array_71, 0, 8192*8);
    memset(read_inds_0, 0, 1*4);
    memset(read_inds_1, 0, 3*4);
    memset(read_inds_2, 0, 1*4);
7. Experiments and results.

```c
memset(read_inds_3, 0, 4*4);

// SIMILAR
memset(read_inds_70, 0, 1*4);
memset(read_inds_71, 0, 1*4);

/////////////////////////////////<---<---
// CHANGE!

As final comment, observe that the sizes that the function "memset" initializes, corresponds to the “DECLARE_FIFO” calls; for example, with the second call to this macro; “DECLARE_FIFO(i16, 8192, 1, 3)”; the following two statements are required:

memset(array_1, 0, 8192*2);
memset(read_inds_1, 0, 3*4);

As can be seen, the fact that “array_1” uses the type “i16” in its 8192 elements, makes that “memset” must initialize 8192*2 bytes for “array_1”, since an element of type “i16” takes up 2 bytes. In the case of “read_inds_1”, the type employed is “unsigned int”, which takes up 4 bytes; therefore, “memset” must initialize 3*4 bytes for “read_inds_1”, since it has 3 elements.

6.8.3. Initialize string variables for input/output and mapping files

There are several strings for the management of files, whose pointers must be initialized to “NULL” values, because otherwise valid memory positions will be addressed. In general, these files are related with the arguments of the decoder application and hence, the correspondent strings will be configured depending on them. The changes are located at orcc_util.c and highlighted in List 8.


```c
extern char *optarg;
extern int getopt(int nargc, char * const *nargv, const char *ostr);

// Directory for input files.
char *input_directory = NULL;
```
6.8.4. Initialize other variables

There are many other variables that must be initialized. There are located in the following source files:

- HevcDecoder_Algo_Parser.c
- HevcDecoder_DecodingPictureBuffer.c
- HevcDecoder_generateInfo_GenerateRefList.c
- HevcDecoder_generateInfo_MvComponentPred.c

Zero has been chosen as the default value for these variables. A simple example is shown in List 9; the other files are not shown because they have too many variables to initialize and the task is the same. As can be seen, several variables correspond to state variables of actors (each previous source file represents an actor). Fortunately, the array variables are not too large and can be initialized manually as well. It is advised to look for an automatic procedure, because the task of initialize this large amount of variables is tedious and error prone.

6.9. SDL references suppression

The SDL library, that the C backend uses, is designed to write portable multimedia applications that run in personal computers. The backend employs it to measure the frames decoded per second and display the decoded video on a screen. This library is not suitable for the target platform because it relies on supported OSs different to DSP/BIOS. Therefore, the references to this library are annulled.

The main changes carried out are shown in List 10, List 11 and List 12. The first change consists on the exclusion of the SDL header file. The exclusion of this library forces to annul the calls to the function “SDL_GetTicks” in fpsPrint.c, the file where takes place the computation of the frames decoded per second. It is worth noting that the functionality of some functions of this file has been then annulled; as a clarification, this is not a problem because these functions are not employed and alternative procedures have been carried out (they are presented in a posterior section called “Functions to measure performance”).

List 10. Change in Top_mpegh_part2_main.c (RVC).

```c
#include "SDL.h" //osx required //CHANGE!
```

List 11. Changes in fpsPrint.c (ORCC).

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
static unsigned int startTime;
```
static unsigned int relativeStartTime;
static int lastNumPic;
static int numPicturesDecoded;

static unsigned int partialStartTime;  //CHANGE! before it was Uint32
static unsigned int partialEndTime;  //CHANGE! before it was Uint32
static int partialNumPicturesStart;
static int partialNumPicturesEnd;

static int show_fps = 1;

static void print_fps_avg(void) {
    unsigned int endTime = SDL_GetTicks();  //CHANGE!
    printf("%i images in %f seconds: %f FPS\n", 
           numPicturesDecoded, (float) (endTime - startTime)/ 1000.0f, 
           1000.0f * (float) numPicturesDecoded / (float) (endTime – 
           startTime));
}

void fpsPrintInit() {
    startTime = SDL_GetTicks();  //CHANGE!
    numPicturesDecoded = 0;
    lastNumPic = 0;
    if(show_fps) {
        atexit(print_fps_avg);
    }

    relativeStartTime = startTime;
    // For genetic algorithm
    partialStartTime = startTime;
    partialNumPicturesStart = 0;
}

void fpsPrintNewPicDecoded(void) {
    unsigned int endTime;
    numPicturesDecoded++;
    //endTime = SDL_GetTicks();  //CHANGE!
    if (show_fps && (endTime - relativeStartTime) / 1000.0f >= 5) {
        /* printf("%f images/sec
", 
               1000.0f * (float) (numPicturesDecoded - lastNumPic) 
               / (float) (endTime - relativeStartTime));*/

        relativeStartTime = endTime;
        lastNumPic = numPicturesDecoded;
    }
}

void backupPartialStartInfo() {
    //partialStartInfo = SDL_GetTicks();  //CHANGE!
    partialNumPicturesStart = partialNumPicturesEnd;
}

void backupPartialEndInfo() {
    //partialEndTime = SDL_GetTicks();  //CHANGE!
    partialNumPicturesEnd = numPicturesDecoded;
}

[...]
7. Experiments and results.

List 12. Change in display.c (ORCC).

```
//#include "SDL.h"    //CHANGE!
#include "orcc_util.h"
```

The file display.c of the project ORCC has many more references to the SDL library, precisely inside the functions it implements. However, these references do not require to be annulled, because they are never called due to the fact that those calling functions are the ones annulled. This means that for instance, if a function “displayYUV_init” references the SDL library and all the calls to this function are annulled, it is not necessary to annul explicitly the reference. The suppression of this function calls is presented in a posterior section called “YUV files”.

6.10. Remove multicore management code

Although the solution generated was a single-thread implementation, it keeps some code associated to the management of several CPU cores. This code is even OS dependent. The task to do is simply remove this code following the indications of List 13 and List 14.


```
static void launcher() {
    int i;
    //cpu_set_t cpuset;    //CHANGE!
    thread_struct threads[MAX_THREAD_NB];
    thread_id_struct threads_Id[MAX_THREAD_NB];
```

```
for(i=0; i < mapping->number_of_threads; ++i){
    sched_init(&schedulers[i], i, mapping->partitions_size[i],
        mapping->partitions_of_actors[i], &waiting_schedulables[i],
        &waiting_schedulables[(i+1) % mapping->number_of_threads],
        mapping->number_of_threads, NULL);
}
```
7. Experiments and results.

//clear_cpu_set(cpuset);   //CHANGE!

pthread_init();  //CHANGE! inicializacion threads importante!

for(i=0 ; i < mapping->number_of_threads ; i++){
    thread_create(threads[i], scheduler, schedulers[i],
    threads_id[i]);
    //set_thread_affinity(cpuset, mapping->threads_affinities[i], threads[i]);   //CHANGE!
}

for(i=0 ; i < mapping->number_of_threads ; i++){
    thread_join(threads[i]);
}

[. . .]

List 14. Change in roxml.c (ROXML).

void ROXML_API roxml_release(void * data)
{
    memory_cell_t *ptr = &head_cell;
    memory_cell_t *to_delete = NULL;

    if(data == RELEASE_LAST) {
        /**while((ptr->prev != NULL)&&(ptr->prev->id !=
                pthread_self()))
        {
            ptr = ptr->prev;
        }   //CHANGE!
        printf("\n Change: while removed! \n");   //CHANGE!
    }

    [. . .]

6.11. Implement unavailable functions

In this step, several changes are necessary due to the fact that some functions
and structures are not present in the libraries of TI for the DSP environment, while
others are required because several functionalities are not present in the backend’s
code. The former case is the case of the functions fdopen, strdup, fstat, fileno, etc. The
latter is the case of the code required to measure the performance and generate the
YUV files.
6.11.1. Replacement of the fdopen function

The fdopen function, which is employed for file openings and is present in common stdio libraries, is not available in the DSP environment, so a solution must be found. Because it is a simply file opening, it can be replaced by the function “fopen”, which is available in the stdio library of TI (see List 15).

List 15. Changes in roxml.c (ROXML).

```c
node_t * ROXML_API roxml_load_fd(int fd)
{
    FILE * file = NULL;
    node_t *current_node = NULL;
    if(fd < 0) {
        return NULL;
    }
    //file = fdopen(fd, "r"); //CHANGE!
    file = fopen((const char*)fd,"r"); //CHANGE!
    if(file == NULL) {
        return NULL;
    }
    current_node = roxml_create_node(0, file, ROXML_ELM_NODE | ROXML_FILE);
    current_node = roxml_parent_node(NULL, current_node, 0);
    return roxml_load(current_node, file, NULL);
}
```

6.11.2. Function strdup

The function “strdup” (see Table 15), declared in usual string.h libraries, is unavailable in the specific version of string.h provided by TI. The changes carried out, necessary to fix this issue, are shown in List 16 and List 17.

Table 15. Short description of function “strdup”.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>char *strdup(const char *s1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library (header file)</td>
<td>string.h</td>
</tr>
<tr>
<td>Description</td>
<td>The strdup() function shall return a pointer to a new string, which is a duplicate of the string pointed to by s1. A null pointer is returned if the new string cannot be created.</td>
</tr>
</tbody>
</table>
7. Experiments and results.

**List 16. Changes in roxml.c (ROXML).**

```c
node_t ** ROXML_API roxml_xpath(node_t *n, char * path, int *nb_ans) {
    int index = 0;
    int count = 0;
    xpath_node_t * xpath = NULL;
    node_t ** node_set = NULL;
    node_t * root = n;
    char * full_path_to_find;
    char * path_to_find;
    if(n == NULL) {
        if(nb_ans) { *nb_ans = 0; }
    return NULL;
    }
    root = roxml_get_root(n);
    //full_path_to_find = strdup(path); //CHANGE!
    strcpy(full_path_to_find,path); //CHANGE!
    printf("\n Ojo CHANGE quitado strdup!\n"); //CHANGE!
    path_to_find = full_path_to_find;
    index = roxml_parse_xpath(path_to_find, &xpath, 0);
    [. . .]
    return node_set;
}
```

**List 17. Changes in function “init_orcc” of orcc_util.c (ORCC).**

```c
void init_orcc(int argc, char *argv[]) {
    // every command line option must be followed by ':' if it takes an
    // argument, and '::' if this argument is optional
    const char *ostr = "i:no:d:m:f:w:g:l:";
    int c;
    const char * fichero2;  
    fichero2=(char*)malloc(255);
    char * fichero3;
    fichero3=(char*)malloc(255);
    strcpy(fichero2(optarg));
    strcpy(fichero3,optarg);
    while ((c = getopt(argc, argv, ostr)) != -1) { 
        switch (c) {
        case '?': // BADCH
```
7. Experiments and results.

6.11.3. Library stat and function fileno

The "sys/stat.h" library of Unix-based OS has not any equivalent in the DSP development environment and consequently, an alternative must be developed. This library allows obtain information of files. The adaption has been done in the way described below.
Firstly, as shown in List 18, the inclusion of the “sys/stat.h” library is cancelled in all the files that reference it. Then, the “stat” structure of this library is explicitly copied into these files, because this is the structure that stores the information of a file. Observe that in the structure “stat”, several pre-compiler directives have been added (the ones dealing with the “__ARMEB__” flag); they are not needed for the migration of the application, but have been included because they are used later when building the same decoder for an ARM GPP and comparing both platforms (the procedure for running this decoder in an ARM device like the PandaBoard is described in [3]).

List 18. Changes in accessFile.c, compare.c, compareyuv.c, y source.c (ORCC).

```c
//#include <sys/stat.h>  //CHANGE!

struct stat {  //CHANGE!
  #if defined(__ARMEB__)
    unsigned short st_dev;
    unsigned short __pad1;
  #else
    unsigned long  st_dev;
  #endif
  unsigned long  st_ino;
  unsigned short st_mode;
  unsigned short st_nlink;
  unsigned short st_uid;
  unsigned short st_gid;
  #if defined(__ARMEB__)
    unsigned short st_rdev;
    unsigned short __pad2;
  #else
    unsigned long  st_rdev;
  #endif
  #endif
  unsigned long  st_size;
  unsigned long  st_blksize;
  unsigned long  st_blocks;
  unsigned long  st_atime;
  unsigned long  st_atime_nsec;
  unsigned long  st_mtime;
  unsigned long  st_mtime_nsec;
  unsigned long  st_ctime;
  unsigned long  st_ctime_nsec;
  unsigned long  __unused4;
  unsigned long  __unused5;
};
```


Secondly, the calls to the function “fstat” of “sys/stat.h” must be adapted. There is also a function of the common library “stdio.h”, “fileno”, that is not inside the implementation of “stdio.h” provided by TI; therefore it must be adapted as well. A brief description of these functions is provided in Table 16.

Table 16. Short description of functions “fstat” and “fileno”.

<table>
<thead>
<tr>
<th>Function</th>
<th>Prototype</th>
<th>Library (header file)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fstat</td>
<td>int fstat(int fildes, struct stat *buf);</td>
<td>sys/stat.h</td>
<td>The fstat() function obtains information about an open file associated with the file descriptor “fildes”, and writes it to the area pointed to by “buf”.</td>
</tr>
<tr>
<td>fileno</td>
<td>int fileno(FILE *stream);</td>
<td>stdio.h</td>
<td>The fileno() function returns the integer file descriptor associated with the stream pointed to by stream.</td>
</tr>
</tbody>
</table>

As it is expressed in [49], the field “st_size” of the structure “stat” represents the number of bytes in a file and at the same time, the position of the end-of-file mark for the file. Hence, “st_size” gives the file size in bytes. The functions “fstat” and “fileno” are used to retrieve the value of this “st_size” field. This behavior can be replicated using the functions “fseek” and “ftell” available in the “stdio.h” library (see List 19).


```c
unsigned int source_sizeOfFile() {
    struct stat st;
    //fstat(fileno(file), &st); //CHANGE! “fileno for size of file”
    //return st.st_size;

    //---------------------------------- //CHANGE!
    int tamano=0,sz=0;
    int prev=ftell(file);
    fseek(file, 0L, SEEK_END);
    sz=ftell(file);
    fseek(file,prev,SEEK_SET); //go back to where we were

    return sz;
}
```
However, not always this behavior must be replicated; sometimes it is possible to discard these sections of code because the functions that contain them are never called. It could look like a bad solution, because they could be substituted as in List 19; however, this serves to acquire knowledge of the behavior of the application when performing the migration changes (if these functions are called, some errors will appear). See List 20, List 21, List 19 and List 22 to know how these sections of code have been solved.


```c
int sizeOfFile(long desc)
{
    FILE* fd = (FILE*) desc;
    struct stat st;
    // fstat(fileno(fd), &st); //CHANGE!
    // return st.st_size; //CHANGE!
    return ftell((fd)); //CHANGE!
}
```


```c
void compareYUV_init()
{
    struct stat st;
    // Fix me!! Dirty but it's the only way for the moment.
    if (yuv_file == NULL) {
        useCompare = 0;
        return;
    }
    useCompare = 1;
    ptrFile = fopen(yuv_file, "rb");
    if (ptrFile == NULL) {
        fprintf(stderr, "Cannot open yuv_file concatenated input file '%s' for reading\n", yuv_file);
        exit(-1);
    }
    // fstat(fileno(ptrFile), &st); //CHANGE! “fileno for YUV file”
    // fileSize = st.st_size; //CHANGE!
    compareErrors = 0;
}
```
6.11.4. Functions to measure performance

Among other things, the C backend uses the SDL library to implement a measurement mechanism to indicate the number of frames decoded per second. Nevertheless, the usage of the SDL library is not valid for the DSP environment and as mentioned in a previous section, its usage is removed. Hence, an alternative mechanism must be employed.

The alternative chosen is based on the employment of a timer. Thank to the modules offered by TI within DSP/BIOS, the current time in CPU clock cycles can be obtained using the function “CLK_gethtime” of the module CLK (System Clock Manager). With additional computation, it is possible to measure the averaged time that the different software sections take. To measure the time that each FU takes, the measures can be easily taken in the actor scheduler and stored temporally in the data structure of each actor. The next paragraphs report the changes necessary in order to create this software infrastructure for taking measures.

Firstly, to control the measurement acquisition process, the files medidas.h and medidas.c have been created and added to the RVC project. Their usage consists basically on indicating when a test has finished (for example, just after decode 100 frames). Their implementation is shown in List 23 and List 24. As can be seen, their implementation is not very nice currently, but for experimentation and test execution is enough.

List 23. Code of medidas.h.

```c
#ifndef MEDIDAS_H
#define MEDIDAS_H

int fin_medidas(int fin_m);

#endif
```

7. Experiments and results.
7. Experiments and results.


```c
#include "medidas.h"

int salida= 0;

int fin_medidas(int fin_m){
    if(fin_m == 1)
        salida=1;
    else if(fin_m == 2)
        salida=0;
    return salida;
}
```

Secondly, the data structure of each actor (“actor_s”) must be redefined. List 25 shows that two additional fields, “ciclos” and “desborde”, have been added to “actor_s”. These fields are employed to store the count of the time spent in each actor (or FU).


```c
struct actor_s {
    char *name;
    int group; /** id of his group. */
    void (*init_func)();
    void (*reinit_func)();
    void (*sched_func)(struct schedinfo_s *);
    int num_inputs; /** number of input ports */
    int num_outputs; /** number of output ports */
    int in_list; /** set to 1 when the actor is in the schedulable list. Used by add_schedulable to do the membership test in O(1). */
    int in_waiting; /** idem with the waiting list. */
    struct scheduler_s *sched; /** scheduler which execute this actor. */
    int mapping;/** id of the processor core mapped to this actor.*/

    unsigned long ciclos;
    int desborde;
};
```
Finally, in Top_mpegh_part2_main.c (RVC project) it is necessary to perform the changes highlighted in List 26. Firstly, the library headers medidas.h, std.h and clk.h must be included. “clk.h” is the header of the CLK module, which requires the prior inclusion of “std.h” in the application code (“std.h” defines types supported by TI’s DSPs). Secondly, several variables and statements are declared throughout the file with the aim of control the measurement process for all actors (examples of these variables are “cuentas_reinicio”, “reini_actores”, “contados”, “cuenta”, “ini”, “fin”, “diferencia”, etc.). Thirdly, the initialization of the structures “actor_s” requires be adapted to the addition of the fields “ciclos” and “desborde”. Fourthly, there are the calls to the function “CLK_gethtime”. Finally, it is added the fragments of code that control the measurement process and report the measurements obtained.


```
#include "medidas.h" //CHANGE!
#include "C:\ti\bios_5_41_11_38\packages\ti\bios\include\std.h"
//CHANGE!
#include "C:\ti\bios_5_41_11_38\packages\ti\bios\include\clk.h"
//CHANGE!

int cuentas_reinicio = 1; //CHANGE!
int reini_actores = 0; //CHANGE!

////////////////////////////////////////////////////////////////////////////////////////
//CHANGE! Change in the initialization of actors due to the addition of a couple of fields at the end, with the purpose of take measures of their execution!! --> --> -->

// Declaration of the actors array
struct actor_s Source = {"Source", 0, Source_initialize, NULL, Source_scheduler, 0, 0, 0, 0, NULL, 0, 0, 0};
struct actor_s display = {"display", 0, display_initialize, NULL, display_scheduler, 0, 0, 0, 0, NULL, 0, 0, 0};

//CHANGE! [DO THE SAME WITH ALL THE FOLLOWING actor_s initializations]

////////////////////////////////////////////////////////////////////////////////////////
// Actor scheduler
void *scheduler(void *data) {
    struct scheduler_s *sched = (struct scheduler_s *) data;
    struct actor_s *my actor;
```
struct schedinfo_s si;
int j, numero_actores = 38, contados = 0, cuenta = 0; //CHANGE!
unsigned long ini = 0, fin = 0, diferencia = 0; //CHANGE!
char *aux_name;

sched_init_actors(sched, &si);

while (1) {

COMMAND_CODE_CHANGE:
if (reini_actores == 1){
  my_actor->ciclos = 0;
  my_actor->desborde = 0;
  contados++;
  if (contados == numero_actores){
    reini_actores = 0;
    contados = 0;
  }
}
COMMAND_CODE_CHANGE:
ini = CLK_gethtime(); //CHANGE!

my_actor = sched_get_next(sched);
if (my_actor != NULL){
  si.num_firings = 0;
  my_actor->sched_func(&si);
  fin = CLK_gethtime(); //CHANGE!

COMMAND_CODE_CHANGE:
if (fin > ini)
  diferencia = fin - ini;
else
  diferencia = fin + (CONST_ULONG - ini);

COMMAND_CODE_CHANGE:
if (my_actor->ciclos + diferencia) < my_actor->ciclos)
  my_actor->desborde++;

COMMAND_CODE_CHANGE:
my_actor->ciclos = my_actor->ciclos + diferencia;

COMMAND_CODE_CHANGE:
#endif

COMMAND_CODE_CHANGE:
if (fin_medidas(0))
  printf("%s;%lu;desborde;%d\n", my_actor->name, my_actor->ciclos, my_actor->desborde);
  cuenta ++;
if (cuenta == numero_actores)
  //reinicio();
  cuenta = 0;
}
6. Experiments and results.

6.11.5. YUV files

Due to the suppression of the SDL library (mentioned in the section “SDL references suppression”), the functions that write the YUV files (with the content of the decoded sequences) must also be annulled. Therefore, an alternative implementation of the writing of YUV files must be carried out. The necessary changes are highlighted in List 27. These changes are based on the edition of the functions that serve to represent on a screen the video decoded.

List 27. Changes in display.c (RVC).

```c
static void displayPicture() {
    u8 tmp_displayYUV_getFlags;
    u8 local_DISP_ENABLE;
    u16 local_cropPicWthLuma;
    u16 local_cropPicHghtLuma;
    i32 local_nbFrameDecoded;

    static char primera_apertura=1; //CHANGE!
    FILE* f; //CHANGE!
    u16 i; //CHANGE!
    static unsigned int imagen=0; //CHANGE!

    fpsPrintNewPicDecoded();

    //tmp_displayYUV_getFlags = displayYUV_getFlags(); //CHANGE!
    tmp_displayYUV_getFlags=1 ; //CHANGE!

    printf("Picture %d . Function removed in display
            picture\n",imagen++); //CHANGE!
    //imagen++; //CHANGE!

    local_DISP_ENABLE = DISP_ENABLE;
    if (imagen == 100)
    {
        printf("\n100 pictures\n");
        fin_medidas(1); //CHANGE!
        imagen = 0; //CHANGE!
    }
```
7. Experiments and results.

local_DISP_ENABLE = DISP_ENABLE;
if ((tmp_displayYUV_getFlags & local_DISP_ENABLE) != 0) {
    local_cropPicWthLuma = cropPicWthLuma;
    local_cropPicHghtLuma = cropPicHghtLuma;
    //displayYUV_displayPicture(pictureBufferY,
        pictureBufferU, pictureBufferV, local_cropPicWthLuma,
        local_cropPicHghtLuma); //CHANGE!
}

static i32 isSchedulable_displayClose() {
    i32 result;
    i32 tmp_source_isMaxLoopsReached;
    i32 local_nbFrameDecoded;
    i32 tmp_displayYUV_getNbFrames;

    tmp_source_isMaxLoopsReached = source_isMaxLoopsReached();
    local_nbFrameDecoded = nbFrameDecoded;
    tmp_displayYUV_getNbFrames = nbFrames; //CHANGE! nbFrames instead displayYUV_getNbFrames();
    result = tmp_source_isMaxLoopsReached || local_nbFrameDecoded == tmp_displayYUV_getNbFrames;
    return result;
}

// Initializes
static void untagged_0() {
    u8 tmp_displayYUV_getFlags;
    u8 local_DISP_ENABLE;

    //tmp_displayYUV_getFlags = displayYUV_getFlags(); //CHANGE!
    local_DISP_ENABLE = DISP_ENABLE;
    if ((tmp_displayYUV_getFlags & local_DISP_ENABLE) != 0) { //displayYUV_Init();//CHANGE!
        compareYUV_init();
        fpsPrintInit();
        nbFrameDecoded = 0;
    }
}
[. . .]
6.12. Others

There are other non-classified changes that are briefly mentioned in this section:

- As shown in List 28, there are types not supported by the DSP environment (not recognized by the compiler and not present in the libraries of TI). The solution has consisted on change the type “Uint32” to “unsigned int” (both types take up 4 bytes).

- Several informative “printf” statements are removed. These statements are located in the following files of the RVC project: HevcDecoder_Algo_parser.c, HevcDecoder_DecodingPictureBuffer.c, and HevcDecoder_SAO.c. They are removed because the information that they shown is not used to measure the performance.

- Temporal removal of the warning information displayed by the actor of MD5_compute.c when the MD5 hash value generated does not coincide with the reference value (see function “compareKey” in List 29).

- The fcntl library, which is employed for file control options, is not available in the DSP environment. When removing it (see List 30), errors do not appear. Then, it is supposed that this function comes with the ROXML library but it is never used.

List 28. Changes in fpsPrint.c (ORCC).

```c
[. . .]
static unsigned int startTime;
static unsigned int relativeStartTime;
static int lastNumPic;
static int numPicturesDecoded;
static unsigned int partialStartTime; //CHANGE! before it was Uint32
static unsigned int partialEndTime; //CHANGE! before it was Uint32
static int partialNumPicturesStart;
static int partialNumPicturesEnd;
[. . .]
```
7. Experiments and results.

List 29. Change in function “compareKey” of MD5_compute.c (RVC).

```c
static void compareKey() {
    i32 flag;
    i32 i;
    u32 tmp_refKey;
    u32 tmp_genKey;
    u32 tmp_genKey0;
    u32 tmp_genKey1;
    u32 tmp_genKey2;
    u32 tmp_genKey3;

    flag = 0;
    i = 0;
    while (i <= 3) {
        tmp_refKey = refKey[i];
        tmp_genKey = genKey[i];
        if (tmp_refKey != tmp_genKey) {
            flag = 1;
        }
        i = i + 1;
    }
    if (flag == 1) {
        ////////////////////////// --> -->  //CHANGE!
        /*printf("error md5 : ");
            tmp_genKey0 = genKey[0];
            hexPrint(tmp_genKey0);
            tmp_genKey1 = genKey[1];
            hexPrint(tmp_genKey1);
            tmp_genKey2 = genKey[2];
            hexPrint(tmp_genKey2);
            tmp_genKey3 = genKey[3];
            hexPrint(tmp_genKey3);
            printf("\n");*/
        ////////////////////////// <-- <--  //CHANGE!
    }
}
```

List 30. Change in roxml-internal.h (ROXML).

```c
#include <stdio.h>
#include <stdlib.h>
//#include <fcntl.h>  //CHANGE!
#include <string.h>
```
6.13. Need of automating the migration

The software migration procedure is a manual process in which a large amount of changes is required. Because the migration procedure must be repeated each time a new DSP-based implementation of a decoder is carried out, it is proposed the research of a solution that automates this process. This solution will shorten the development time and will reduce the likelihood of errors during additional migration steps in the case of updates in the decoders provided in [43].
7 Experiments and results
This chapter describes how have been performed the tests for the decoders, the mechanisms employed to automate these tests and the obtained results.

**7.1. Test sequences**

Having a good set of test sequences is necessary when experimenting with decoders and when extracting performance measurements, which must be realistic. In order to cover the diversity of conditions (e.g. different platforms and configurations), the set of test sequences has to comprise different kinds of codified videos.

The test sequences have been extracted from the database of the standard [B]. This database has a variety of video sequences compliant with different HM versions (HM means HEVC Test Model and is reference software of the HEVC standard). It offers also valuable information about these sequences. The set of possibilities covers among others:

- Types of sequence: intra only (AI), low-delay (LD), low-delay with I and P types of frame only, and random-access (RA).
- Image sizes: 416x240, 832x480, 1024x768, 1280x720, 1920x1080 and 2560x1600.
- Frames per second (fps): 30, 50 and 60.
- Quantization parameter (QP): 22, 27, 32 and 37.

Many possibilities of this set were used during general proofs for GPP-based implementations. However, during proofs for DSP-based implementations the set was more limited but enough for a test of concept of the decoder. Due to the memory constraints of the DM6437 EVM and the requirements of the target applications, only image sizes smaller than 1024x512 were employed. Regarding QP, the most realistic values for commercial applications are the centered around 30 (a QP of 37 produces very poor quality and high compression, a QP of 22 produces very low compressing and high quality).

The test, whose results are introduced in a later subsection, employs the test bench presented in Table 17. This test bench is based on HM10.0 sequences with configurations AI, LD and RA, each with the same four video sequences of Class-C (832x480 pixels). These sequences have a bit depth of 8 bits and as QPs, 27 and 32.
The basic reason that motivated the selection of these sequences was their presence in other existing publications; something that can be employed for comparing the performance results.

**7.2. Test automation**

The characterization of decoders has been carried out using an automatic procedure. If the test for each one of the previous sequences had been performed manually, the tester would have required spend a lot of time because it would have been necessary configure the environment and wait until the end of the test for each sequence. The automation procedure consists on a method that is based on scripts and was elaborated in other work performed in the GDEM. Here, this method is outlined and some comments about required adaptations are presented.
The basics of the automation method are represented in Figure 33 and outlined in the following paragraphs. The color dark purple indicates the start and end of an automated test, the blue each stage of the method, the green input files that feed these stages and the red the results of the test.

The method takes advantage of the scripting possibilities afforded by CCS. The Debug Scripting Server (DSS) [50] is the component of CCS that performs the debugging and can be launched outside the CCS environment. The method calls the DSS from a script, something that forces to implement in the “main” function of the decoder the extraction of the “test parameters” using a swap file. The code that extracts this information opens firstly the swap file and reads then some input parameters written in this file, such as the current test sequence, the location of the output “*.yuv” file, the number of frames to decode, etc. The swap file is therefore a file that is useful for moving the test parameters from the PC OS environment to the DSP environment. Although there are other automation methods within the CCS environment, they result impractical because additional complex configurations are required (for instance, to introduce through the decoder arguments the parameters, it is necessary define explicitly a memory reserve for arguments, but for example the size of the test sequences is not the same for all and usually it is unknown).

Figure 33. Basics of the automation method.
The test starts launching an OS script file (a batch file for Windows-based OS or a shell file for Linux-based OS). This file is launched from the OS command prompt. Firstly, this script establishes the test settings described in Table 18. Notice that paths to the files are required if they are not in the current working path. Secondly, with these settings the OS script calls a Python script, which creates the swap file and calls the DSS.

Table 18. Test settings in the OS script.

<table>
<thead>
<tr>
<th>Test settings</th>
<th>Description</th>
<th>File format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>Decoder application</td>
<td>*.out</td>
<td>\application\RVC.out</td>
</tr>
<tr>
<td>ccsconf</td>
<td>Target configuration</td>
<td>*.ccxml</td>
<td>\application\DM6437EVM.ccxml</td>
</tr>
<tr>
<td>dssconf</td>
<td>Script to configure the DSS</td>
<td>*.js</td>
<td>emulator.js</td>
</tr>
<tr>
<td>frames</td>
<td>Number of frames to decode</td>
<td>N/A</td>
<td>100</td>
</tr>
<tr>
<td>listfile</td>
<td>List with the test sequence names</td>
<td>*.txt</td>
<td>list.txt</td>
</tr>
</tbody>
</table>

Prior to starting the execution of the decoder, the DSS must be correctly configured. To do this, the DSS uses a Javascript (JS) file that establishes all the steps required to run the decoder (e.g. open a debug session, connect to the target device, load the program, run it, disconnect, etc.).

The proper execution of the decoder creates a couple of files for each sequence, one is the CSV file with the measured data and the other is the YUV file produced by the decoder. These files are placed in the folders programmed in the OS script. The way to produce the automatic measurements consists on integrating in the decoder code some special functions that using the target platform timer are able to count the time passed in clock cycles (the changes required were explained in Chapter 6).

It is worth mentioning that depending on the OS employed, it is employed different scripting languages (e.g. batch or shell), OS symbols (e.g. separator character in paths) and tool names (the DSS executable may have different name). Although the results presented here were generated using this method under a Windows framework, in this MT this framework has been migrated in order to start working under Ubuntu. Next, the most notorious changes for the migration are listed:
• Transform the batch scripting language into shell scripting language (e.g. "#" instead "rem" for comments, "$bin" instead "%bin%", strings between double quotes, different form to define variables, etc.).

• Usage of the slash ("/") instead the backslash ("\”).

• Path to the DSS executable and its name (it is in the following folder within the installation folder of CCS: “…\ti\ccsv5\ccs_base\scripting\bin”).

Summarizing, the things that testers should supervise and edit when making tests with this methodology are: the working directory structure within the file system and the existence of all the files required, the OS script (general settings and OS adaption), the sequence names list, the Python script (OS adaption), and the JS file (information to specify how to open a session with the target device, for instance the usage of an emulator connection).

### 7.3. Results obtained for HEVC-RVC

As reference, HEVC-RVC is the name given to the decoder obtained using the working methodology and migration process explained in previous chapters. The tests performed have been focused on the analysis of performance for this decoder and its profiling.

For each sequence of the test bench, the number of CPU cycles (x10⁹) and the fps decoded by a single core have been measured. Also, the distribution of execution time among the different parts of the decoder has been analyzed. It is worth mentioning that all these measurements are averaged over the first 100 frames of each sequence.

In order to present the results, the main FUs of the HEVC-RVC decoder have been mapped into common functional blocks as it is shown in Table 19. HEVC-RVC corresponds to the component “HevcDecoder” that was introduced in Chapter 3 (hence, FUs “Source”, “display” and “MD5” have been discarded in the results).

At the end of this section, the results of the measurements are presented in Table 20. To ease the comparison of decoding time distribution among different sequence configurations (AI, LD and RA) and distinct QPs, the chart diagrams of Figure 34 and the graph of Figure 35 are provided after Table 20. Figure 35 presents the differences in the percentages of the decoding time distribution of the functional blocks, with QP=32 relative to QP=27. Next paragraphs comment these results.
The performance analysis of the HEVC-RVC decoder demonstrates that real-time performance is not achievable with one DSP core running at 700 MHz, as the best averaged measure is around 1.5 fps (see Table 20). Nevertheless, it is worth mentioning that, in all cases, the software that implements the decoder has not been optimized for the core architecture. Thus, an optimization process could lead to the achievement of real-time performance.

The profiling reveals that with AI configuration, ED and ITQ are the most time consuming functional blocks as they spend, taken as a whole, around 52% of the decoding time. The loop filters, DF and SAO, take around 12% each one, while the IP takes 10%.

As can be seen in Figure 34, contrasting the profiling results obtained for LD and RA configurations, it is observed they are not so different, even though RA has more reference frames than LD. In both cases, the EP block spends more than 50% of the decoding time. The total percentage of decoding time consumed by the loop filters, DF and SAO, is about 19%. The ITQ block consumes about 11%.

Finally, the analysis shown in Figure 35 indicate that in general the QP has influence in the distribution of the computational load, with higher influence in configurations LD and RA than in AI. For example, the ED is the functional block that experiments more variation, because its load can be up to 30% higher for the sequences with QP=27 regarding those with QP=32 (case of LD and RA, with AI it can be up to 13%).
Table 20. HEVC-RVC: Performance analysis and profiling results.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>1 core performance</th>
<th>Load distribution among functional blocks (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>QP</td>
<td>CLKS ($\times 10^9$)</td>
</tr>
<tr>
<td>Conf.</td>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>BD</td>
<td>27</td>
<td>88.1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>67.5</td>
</tr>
<tr>
<td>BM</td>
<td>27</td>
<td>83.1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>71.4</td>
</tr>
<tr>
<td>PS</td>
<td>27</td>
<td>118.6</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>103.3</td>
</tr>
<tr>
<td>RH</td>
<td>27</td>
<td>76.1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>69.2</td>
</tr>
<tr>
<td>Average</td>
<td>27</td>
<td>91.5</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>77.9</td>
</tr>
<tr>
<td>BD</td>
<td>27</td>
<td>51.5</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>45.5</td>
</tr>
<tr>
<td>BM</td>
<td>27</td>
<td>51.2</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>45.7</td>
</tr>
<tr>
<td>PS</td>
<td>27</td>
<td>65.4</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>55.7</td>
</tr>
<tr>
<td>RH</td>
<td>27</td>
<td>66.6</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>56.6</td>
</tr>
<tr>
<td>Average</td>
<td>27</td>
<td>58.68</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>50.9</td>
</tr>
<tr>
<td>BD</td>
<td>27</td>
<td>51.3</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>45.8</td>
</tr>
<tr>
<td>BM</td>
<td>27</td>
<td>50.6</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>46.5</td>
</tr>
<tr>
<td>PS</td>
<td>27</td>
<td>61.3</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>54.4</td>
</tr>
<tr>
<td>RH</td>
<td>27</td>
<td>63.8</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>55.8</td>
</tr>
<tr>
<td>Average</td>
<td>27</td>
<td>56.8</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>50.6</td>
</tr>
</tbody>
</table>

7. Experiments and results.
7. Experiments and results.

Figure 34. HEVC-RVC: Decoding time distribution among functional blocks.

Figure 35. HEVC-RVC: Comparison of decoding time distribution with respect to QP.
7.4. Comparison with other HEVC decoders

The performance analysis and profiling of the HEVC-RVC decoder serve as a test of concept for the HEVC standard within the RVC framework. In order to evaluate the goodness of HEVC-RVC, the performance results have been compared with the reached in other HEVC implementations. The following three have been taken into account:

- ARM-based version of the HEVC-RVC decoder. The decoder is the same as HEVC-RVC but without the migration procedure carried out in this MT. It was generated with the working methodology for GPP targets presented in Chapter 5 and its performance results have been taken from [23]. It was tested in a system running at 1 GHz.

- DSP-based decoder implemented from the HM9.0 reference software [51]. It was tested in a system running at 596 MHz.

- DSP-based decoder implemented from the open source code OpenHEVC [52]. It was tested in a system running at 596 MHz.

All these HEVC decoders were performed in works of the GDEM, and it is important noting that they are single-core implementations that were not optimized for the system where they were tested. When comparing HEVC-RVC with these three implementations, it is important considerate the difference in the operating rate, since HEVC-RVC was tested in a DSP core running at 700 MHz.

The performance results of the four implementations to be compared have been arranged in Table 21. For each of the sequences of the test bench, the frames per second obtained with each implementation are shown. Additionally, the percentage difference between the performance data of the HEVC-RVC decoder and its version based on an ARM processor is given.

As can be seen, the results for the HEVC-RVC decoder are better than those obtained for the ARM-based implementation; but worse than those obtained in the other DSP-based implementations. The DSP-based implementation of HEVC-RVC outperforms the ARM-based by more than 50% (up to around 85% with configurations LD and RA). In the rest of cases, HEVC-RVC is less efficient. The results for the HM9.0 implementation double the obtained for HEVC-RVC. The obtained for the Open HEVC implementation are around 5 to 6 times higher.
Table 21. Comparison of several HEVC implementations in one processor core.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>HEVC Decoder Implementations for DSP (fps)</th>
<th>ARM Processor (HEVC-RVC version)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conf.</td>
<td>QP</td>
<td>HEVC-RVC</td>
</tr>
<tr>
<td>AI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BD</td>
<td>27</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.04</td>
</tr>
<tr>
<td>BM</td>
<td>27</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0.98</td>
</tr>
<tr>
<td>PS</td>
<td>27</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0.68</td>
</tr>
<tr>
<td>RH</td>
<td>27</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.01</td>
</tr>
<tr>
<td>Average</td>
<td>27</td>
<td>0.79</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0.93</td>
</tr>
<tr>
<td>LD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BD</td>
<td>27</td>
<td>1.36</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.54</td>
</tr>
<tr>
<td>BM</td>
<td>27</td>
<td>1.37</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.53</td>
</tr>
<tr>
<td>PS</td>
<td>27</td>
<td>1.07</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.26</td>
</tr>
<tr>
<td>RH</td>
<td>27</td>
<td>1.05</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.24</td>
</tr>
<tr>
<td>Average</td>
<td>27</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.39</td>
</tr>
<tr>
<td>RA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BD</td>
<td>27</td>
<td>1.36</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.53</td>
</tr>
<tr>
<td>BM</td>
<td>27</td>
<td>1.38</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.51</td>
</tr>
<tr>
<td>PS</td>
<td>27</td>
<td>1.14</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.29</td>
</tr>
<tr>
<td>RH</td>
<td>27</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.25</td>
</tr>
<tr>
<td>Average</td>
<td>27</td>
<td>1.25</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.40</td>
</tr>
</tbody>
</table>

Consequently, the performance results obtained for an implementation of HEVC-RVC using one DSP core are low (the best averaged measure is far from real-time performance, since it achieves around 1.5 fps). However, it is expected that the performance increases after taking several possible improvements. Firstly, the optimization of the decoder software with respect to the core architecture could lead to important progresses. Secondly, an optimization of the decoder implementation using Orcc could generate more efficient versions. Finally, it is worth noting that the RVC
framework allows start working over multicore platforms, which will bring important benefits when exploiting the parallelization opportunities. For instance, if a multicore DSP is employed, it is expected a performance improvement if the functional block for inter-prediction, which consumes up to around 50% of the decoding time, is placed in one core and the rest of blocks are placed in other core.
Conclusions and future work
8. Conclusions and future work.

8.1. Conclusions

With this MT work, it was intended to achieve a methodology that would enable to run video decoders created according to the RVC standard, on multicore DSPs. However, due to the large number of issues and challenges to overcome, there are still many things to do since only it has been possible to perform a first advance to the goal. This advance is the achievement of a methodology that enables run these video decoders on one core of a multicore DSP of the company Texas Instruments.

This methodology comprises a software migration process in which the video decoders, which are targeted to GPP-based platforms, are adapted to the features of DSP-based devices. This adaption is carried out manually and due to the large amount of changes required, it is proposed the research of an automation procedure in order to shorten the development time. The automation procedure is a priority because currently, the migration procedure must be repeated each time a new DSP-based decoder is required. It is expected that the work performed in this MT may serve as starting point for this research and the accomplishment of implementations using several cores.

As a consequence of the attainment of this methodology, an RVC-based implementation of an HEVC decoder has been evaluated using one core of a multicore DSP. The results show that real-time performance is not achieved, but improvements are expected when optimization processes and parallelized solutions will be carried out in the future. Up to the best of the GDEM's knowledge, this implementation is the first that tests using DSPs the HEVC standard according to the RVC standard; as a consequence, the obtained results have been submitted to ESTIMedia 2013 [53] [23] in a research paper.

Finally, additional comments about RVC, multicore DSPs and the developed methodology, are provided in the next subsections.

8.1.1. About RVC

The decision of the MPEG for transforming its working methodology for the development of video codecs was an intelligent decision. The employment of a library of FUs provides flexibility and eases the reuse and parallelization of code. Regarding signal processing applications, the dataflow model used fits better than other
programming models; moreover, the programming using RVC-CAL resembles the object oriented programming technique, which usually reduces the development time.

Nevertheless, despite that, this new methodology presents new challenges. For example, a device running applications under RVC must be enough fast to load and instantiate the FUs in the decoder network. There is also an increase with respect to monolithic decoders, in the extension of code to store and execute. Future works with RVC should pay attention to overcome these challenges.

On the other hand, future lines of research can be the extension of the RVC framework to other spheres. Due to their suitability for a dataflow paradigm, RVC can be applied to applications like: pattern recognition, image processing for healthcare, cryptography [54], musical effect processors, communication protocol infrastructure, etc. This fact has led to proposals for the creation of RMC (Reconfigurable Media Coding), a reconfigurable standard for any kind of multimedia application [55].

### 8.1.2. About multicore DSP platforms

All computation technologies (CPU, GPU, DSP, FPGA, ASIC, etc.) have their advantages and disadvantages respect to the other technologies when talking about features like power consumption, latency, type of workload, parallelism, etc (see Table 22).

<table>
<thead>
<tr>
<th>Computation technology</th>
<th>DSP</th>
<th>CPU</th>
<th>GPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Consumption</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Latency</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Parallelism</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workload</th>
<th>Sequential</th>
<th>Iterative</th>
<th>Data-Parallel</th>
<th>Memory-intensive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Multicore platforms are in general a promising approach to solve the speed limitations of computation technologies, because at end, they are a mechanism to increment parallelism and therefore speed. Due to the features presented in the previous table, DSP cores are a good choice for the design of embedded systems that process video; but unfortunately, methodologies that permit a very quick design for a multicore DSP architecture are still under development.
8. Conclusions and future work.

Currently, there is focus on new tools and programming paradigms that allow this; perhaps even, on the way multicore DSPs are programmed and architected. The challenge for applications using the dataflow model is basically partitioning the complex components across cores and the high data flow rate required through the system.

In the case of a video decoder, its components need to be split and mapped to multiple cores while keeping the processing pipeline flowing regularly. Because data movement between cores is regular and low latency hand-offs might be critical, it is required good memory bandwidth between cores.

Moreover, tremendously important is the issue of developing effective and easy-to-use code development and real-time debug tools, because the probability for bugs increases significantly when developing concurrent tasks that are being executed in separated computing pieces.

8.1.3. About the developed methodology

The developed methodology allowed achieve the partial objectives presented in Chapter 1; but however, as mentioned in this section, it is still far from being a very automated methodology for DSP targets, because a lot of additional manual work must be carried out in order to obtain an executable DSP version of the decoder.

The ideal methodology is the described in Figure 36. Following a similar working flow regarding the methodology for GPPs, it permits keep the two different levels of edition while optimizing the usage of CCS thank to the direct generation of CCS projects with CMake. Moreover, the CCS projects generated with CMake are already adapted to the DSP environment, avoiding the need of performing repetitively migration tasks.

The backends of this ideal working methodology not only must overcome the problems explained in Chapter 6, but also must be able to provide backends optimized for the use and exploitation of the XCF file in DSPs. As shown in Figure 37, this consists on the automatic configuration of the multicore DSP target by means of the FU distribution described in the XCF file.

Therefore, the future work must consist on taking advantage of the migration procedure described and trying to construct, for example, a specific code generator for Orcc able to generate directly the appropriate backend for TI DSPs, and a suitable CMake generator for CCS (CMake provides flexibility for extending its set of
Conclusions and future work.

Generators). It is worth noting that the most time consuming part within the working methodology carried out, is the related with the migration procedure performed with CCS.

![Diagram of the working methodology for DSP targets](image)

**First level of edition.**
Decoder designs based on the RVC specification are implemented using these plugins for Eclipse.

**Partial result.**
The ORCC compilation provides a backend in the chosen language and a static distribution of actors among the processor cores.

**Second level of edition.**
The solution is configured to the Code Composer Studio IDE, capable to perform edition, debugging and building tasks.
At this level, the solution is customized editing the decoder source files and the distribution file.
It is employed the cross-compiler employed for the target platform.

**Final result.**
The result of the building process is the executable that is able to decode video sequences on the multicore DSP target according to the XCF distribution file.

Figure 36. Ideal working methodology for DSP targets.
8. Conclusions and future work.

Since the automation method for testing of decoders (outlined in Chapter 7) involves a series of changes in multiple files, it is proposed also the development of a tool, easy to use, that generates automatically the infrastructure required. For example, the tool could have a graphical interface that allows the user the selection of the operating system, the desired parameters of the test, the sequences of the test bench, etc., and then, generates or configures automatically all the files and folders required.

Other possible future works consists on optimizing the performance of tools over Eclipse, since it is a heavy environment for the PC that produce long waiting times, and use the methodologies developed for other kinds of applications different, not only video codecs.

8.2. Future works

Although this MT work has achieved the objectives posed, there are still many things to do. It is an initial work that addresses only the first challenge: the production of a working methodology that enables to run RVC-based decoders on a DSP core. It is expected that the work performed in this MT may serve as starting point for future works. Among the tasks that remain outstanding, there are:

- Automate the migration procedure, shortening the time required and optimizing the modifications carried out throughout this process.

- Automate the generation of environments to carry out decoder tests.

- Create tools that document automatically the changes performed.

- Extend the methodology and the migration procedure in order to carry out multicore implementations. In a further step, search a way to distribute automatically the actors among cores in a multicore DSP system.
The references that have appeared throughout this document are the following:


8. Conclusions and future work.


[22] Fraunhofer Heinrich Hertz Institute, web site about HEVC: http://hevc.hhi.fraunhofer.de/


[27] PandaBoard platform web site: http://pandaboard.org/content/platform


[37] CMake web site: http://www.cmake.org/

[38] Code Composer Studio™ web site: http://www.ti.com/tool/ccstudio


[40] Visual Studio web site: www.microsoft.com/visualstudio

[41] Eclipse CDT web site: http://www.eclipse.org/cdt/

[42] PySVN Workbench web site: http://pysvn.tigris.org/

[43] Orcc applications repository at GitHub: https://github.com/orcc/orc-apps

[44] Library ROXML, web site: http://www.libroxml.net/

[45] Library Simple Direct Media Layer (SDL), web site: http://www.libsdl.org/


8. Conclusions and future work.


[50] Debug Scripting Server (DSS) web site: http://www.opengroup.org/


[53] ESTIMedia 2013 web site: http://www.estimedia.org/


[55] Reconfigurable Media Coding article: http://ride.chiariglione.org/RMC.php

[56] Oracle web site: www.oracle.com

[57] Discussion in forum about how increase the Java heap: http://stackoverflow.com/questions/8600972/increasing-heap-space-in-eclipse-java-lang-outofmemoryerror
10 Annex I: CD content
This annex has the aim to provide an overview of the contents of the CD that is appended to this Master Thesis work.

In the root directory there are 5 folders whose name and content is reflected in Table 23. These folders are composed also by other folders with the purpose of an easy location of files.

<table>
<thead>
<tr>
<th>Folder name</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Thesis documents</td>
<td>• Memory (this document)</td>
</tr>
<tr>
<td></td>
<td>• Abstract (in Spanish and English)</td>
</tr>
<tr>
<td>Article</td>
<td>The article submitted to the symposium “ESTIMedia 2013”.</td>
</tr>
<tr>
<td>Orcc applications</td>
<td>The Orcc applications at several dates.</td>
</tr>
<tr>
<td>DSP solution</td>
<td>The implementation of the HEVC-RVC decoder for the DM6437 EVM.</td>
</tr>
<tr>
<td>Test results</td>
<td>Results of the characterization of the decoder.</td>
</tr>
<tr>
<td>References and others</td>
<td>• Reference documents by chapter</td>
</tr>
<tr>
<td></td>
<td>• Images used along this memory.</td>
</tr>
</tbody>
</table>

To visualize each one of the files of the CD is necessary have applications ready for that. Some remarks are provided:

- The Orcc applications can be opened usually using text editors; however, the best suite tool is Eclipse Indigo with the Orcc infrastructure.

- Most files of the DSP solution can be opened with usual text editors; however, the most suitable tool is CCS 5 or later.

- Images are opened using visors of JPEG and PNG formats.

- The rest of files are documents with extension .doc, .docx and .pdf. The use of proper tools to open these formats is advised.
11 Annex II: Structure of the C backend
The main folder structure of a C backend is the presented in Figure 38. The "src" folder stores the source files of the decoder and the XCF file generated. The "libs" folder stores three folders that contain each one a library required by the decoder application: “orcc”, “roxml” and “windows” (this last one is not used throughout this work because it contains the SDL library, which is not required). The “build” folder is employed by CMake to establish the building environment for the decoder (e.g. Unix utilities or Visual Studio). The “bin” folder is the place where the executable decoder application is stored.

![Figure 38. Main folder structure of a C backend.](image)

The interesting files of the C backend solution obtained at date 13/06/2013 are presented in Table 24, Table 25 and Table 26. An example of XCF file for a single-core implementation is presented in Table 27.

<table>
<thead>
<tr>
<th>Header files (.h)</th>
<th>Source files (.c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>orcc_fifo</td>
<td>accessFile</td>
</tr>
<tr>
<td>orcc_fifo_socket</td>
<td>compare</td>
</tr>
<tr>
<td>orcc_generic_fifo</td>
<td>compareyuv</td>
</tr>
<tr>
<td>orcc_generic_fifo_socket</td>
<td>display</td>
</tr>
<tr>
<td>orcc_genetic</td>
<td>fpsPrint</td>
</tr>
<tr>
<td>orcc_scheduler</td>
<td>genetic</td>
</tr>
<tr>
<td>orcc_thread</td>
<td>getopt</td>
</tr>
<tr>
<td>orcc_types</td>
<td>multiWriter</td>
</tr>
<tr>
<td>orcc_util</td>
<td>orcc_util</td>
</tr>
<tr>
<td>socket</td>
<td>scheduler</td>
</tr>
<tr>
<td>-</td>
<td>source</td>
</tr>
<tr>
<td>-</td>
<td>thread</td>
</tr>
<tr>
<td>-</td>
<td>writer</td>
</tr>
</tbody>
</table>
Table 25. Files inside the “libs/roxml” folder.

<table>
<thead>
<tr>
<th>Header files (.h)</th>
<th>Source files (.c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>roxml</td>
<td>roxml</td>
</tr>
<tr>
<td>roxml_win32_native</td>
<td>roxml-internal</td>
</tr>
<tr>
<td>roxml-defines</td>
<td>roxml-parse-engine</td>
</tr>
<tr>
<td>roxml-internal</td>
<td>-</td>
</tr>
<tr>
<td>roxml-parse-engine</td>
<td>-</td>
</tr>
<tr>
<td>roxml-types</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 26. Decoder source files inside the “src” folder.

<table>
<thead>
<tr>
<th>Source files (.c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>display</td>
</tr>
<tr>
<td>HevcDecoder_Algo_Parser</td>
</tr>
<tr>
<td>HevcDecoder_DBFilt_DeblockFilt</td>
</tr>
<tr>
<td>HevcDecoder_DBFilt_GenerateBs</td>
</tr>
<tr>
<td>HevcDecoder_DecodingPictureBuffer</td>
</tr>
<tr>
<td>HevcDecoder_generateInfo_GenerateRefList</td>
</tr>
<tr>
<td>HevcDecoder_generateInfo_MvComponentPred</td>
</tr>
<tr>
<td>HevcDecoder_InterPrediction</td>
</tr>
<tr>
<td>HevcDecoder_IntraPrediction</td>
</tr>
<tr>
<td>HevcDecoder_QpGen</td>
</tr>
<tr>
<td>HevcDecoder_SA0</td>
</tr>
<tr>
<td>HevcDecoder_SelectCU</td>
</tr>
<tr>
<td>HevcDecoder_xIt_Block_Merger</td>
</tr>
<tr>
<td>HevcDecoder_xIt_invDST4x4_1st</td>
</tr>
<tr>
<td>HevcDecoder_xIt_invDST4x4_2nd</td>
</tr>
<tr>
<td>HevcDecoder_xIt_IT4x4_1d_0</td>
</tr>
<tr>
<td>HevcDecoder_xIt_IT4x4_1d_1</td>
</tr>
<tr>
<td>HevcDecoder_xIt_IT4x4_Transpose4x4_0</td>
</tr>
<tr>
<td>HevcDecoder_xIt_IT4x4_Transpose4x4_1</td>
</tr>
<tr>
<td>HevcDecoder_xIt_IT8x8_1d_0</td>
</tr>
</tbody>
</table>

display
HevcDecoder_xIt_IT8x8_1d_1
HevcDecoder_xIt_IT8x8_Transpose8x8_0
HevcDecoder_xIt_IT8x8_Transpose8x8_1
HevcDecoder_xIt_IT16x16_1d_0
HevcDecoder_xIt_IT16x16_1d_1
HevcDecoder_xIt_IT16x16_Transpose16x16_0
HevcDecoder_xIt_IT16x16_Transpose16x16_1
HevcDecoder_xIt_IT32x32_1d_0
HevcDecoder_xIt_IT32x32_1d_1
HevcDecoder_xIt_IT32x32_Transpose32x32_0
HevcDecoder_xIt_IT32x32_Transpose32x32_1
HevcDecoder_xIt_IT_Merger
HevcDecoder_xIt_IT_Splitter
MD5_compute
MD5_padding
MD5_MD5SplitInfo
Source
Top_mpegh_part2_main
Table 27. Example of XCF file for a single-core implementation.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Partitioning>
    <Partition id="1">
      <Instance id="Source"/>
      <Instance id="display"/>
      <Instance id="HevcDecoder_Algo_Parser"/>
      <Instance id="HevcDecoder_IntraPrediction"/>
      <Instance id="HevcDecoder_SelectCU"/>
      <Instance id="HevcDecoder_DecodingPictureBuffer"/>
      <Instance id="HevcDecoder_InterPrediction"/>
      <Instance id="HevcDecoder_SAO"/>
      <Instance id="HevcDecoder_QpGen"/>
      <Instance id="HevcDecoder_xIT_IT_Splitter"/>
      <Instance id="HevcDecoder_xIT_IT_Merger"/>
      <Instance id="HevcDecoder_xIT_invDST4x4_1st"/>
      <Instance id="HevcDecoder_xIT_invDST4x4_2nd"/>
      <Instance id="HevcDecoder_xIT_IT4x4_IT4x4_1d_0"/>
      <Instance id="HevcDecoder_xIT_IT4x4_Transpose4x4_0"/>
      <Instance id="HevcDecoder_xIT_IT4x4_IT4x4_1d_1"/>
      <Instance id="HevcDecoder_xIT_IT4x4_Transpose4x4_1"/>
      <Instance id="HevcDecoder_xIT8x8_IT8x8_1d_0"/>
      <Instance id="HevcDecoder_xIT8x8_Transpose8x8_0"/>
      <Instance id="HevcDecoder_xIT8x8_IT8x8_1d_1"/>
      <Instance id="HevcDecoder_xIT8x8_Transpose8x8_1"/>
      <Instance id="HevcDecoder_xIT8x8_IT16x16_1d_0"/>
      <Instance id="HevcDecoder_xIT8x8_IT16x16_IT8x8_1d_0"/>
      <Instance id="HevcDecoder_xIT8x8_IT16x16_IT8x8_1d_1"/>
      <Instance id="HevcDecoder_xIT8x8_IT16x16_Transpose16x16_1"/>
      <Instance id="HevcDecoder_xIT8x8_IT32x32_IT32x32_1d_0"/>
      <Instance id="HevcDecoder_xIT8x8_IT32x32_Transpose32x32_0"/>
      <Instance id="HevcDecoder_xIT8x8_IT32x32_IT32x32_1d_1"/>
      <Instance id="HevcDecoder_DBFilter_DeblockFilt"/>
      <Instance id="HevcDecoder_DBFilter_GenerateBs"/>
      <Instance id="HevcDecoder_generateInfo_GenerateRefList"/>
      <Instance id="HevcDecoder_generateInfo_MvComponentPred"/>
      <Instance id="MD5_padding"/>
      <Instance id="MD5_shifter"/>
      <Instance id="MD5_compute"/>
      <Instance id="MD5_MD5SplitInfo"/>
    </Partition>
  </Partitioning>
  <!-- Other useful informations related to any element of the instanciated model can be printed here -->
</Configuration>
```
12 Annex III: Installation guide and tutorial
12.1. Installation guide

This installation guide is not a complete installation guide, because the installation will depend on the operating system employed and the tools employed. Instead, the aim of this installation guide is to provide some hints about the installation in the Ubuntu OS. Another installation guide more complete can be found in [3].

During this MT, the software development environment used for carried out the working methodology presented in Chapter 6, is basically the indicated in Table 28 (notice that the Eclipse CDPT4 tool is not recommended.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Tools for Ubuntu</th>
<th>Tools for Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orcc</td>
<td>Eclipse Indigo with Orcc infrastructure plugins</td>
<td></td>
</tr>
<tr>
<td>CMake</td>
<td>CMake 2.8.10 for Ubuntu</td>
<td>CMake 2.8.10 for Windows</td>
</tr>
<tr>
<td>General IDE for C</td>
<td>Code::Blocks, Eclipse CDT4*</td>
<td>Code::Blocks, Visual Studio, Eclipse CDT4*</td>
</tr>
<tr>
<td>CCS</td>
<td></td>
<td>CCSv5.4</td>
</tr>
</tbody>
</table>

The hints to install this environment are the following:

1. Ensure that the Java environment is installed. If not, download from the Oracle website [56] and install:
   - The Java-JRE (Java Runtime Environment).
   - The Java-JDK (Java Development Kit).

2. Download from the Eclipse web site the Eclipse Indigo (v3.7 or later, but Indigo; with other versions of Eclipse, the environment does not work). Decompress the “tar.gz” file in “/usr/lib”.

3. Download from the Eclipse web site the Eclipse for RCP and RAP developers. Decompress the “tar.gz” file in “/usr/lib”.


4. Now, launch Eclipse and move to “Help → Install new software”. Select all the repositories and install the following tools in this order: Eclipse Platform, Eclipse Platform SDK, Eclipse SDK.

5. In the software manager of Eclipse (“Help → Install new software”), add the following repositories:

http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/
http://orcc.sourceforge.net/nightly

6. Install Xtext 2.3.0 (do not install other version of Xtext).

7. Install Graphiti and Orcc.

8. Increase the Java heap following the guide presented in [57]:

12.2. Tutorial

The next tutorial explains how to put in practice the working methodology developed. In general, the tutorial fits any target platform; if decisions dependent of the target are taken, they are notified.

12.2.1. Step one - Get applications

The decoder applications will be downloaded from Internet in their nightly version. The nightly version is the last release of the software project and comprises several development branches. There are other options, but this is the used throughout this MT.

1. Open SVN Workbench (PySVN) to download the nightly version. The reader can observe in Figure 39 the aspect of this SVN tool and the existence of other projects.
2. Create a folder where download the Orcc applications. Here, the name 'ORCC_applications_example' is given.

3. Press "Project → Add" and then select "Use new working copy directory" and click "Next". In the field 'Subversion URL' introduce the URL where the RVC applications are. This URL is:

   https://github.com/orcc/orc-apps

Continue and in the next window fill the field "Working copy Path" with the path to the folder 'ORCC_applications_example', previously created. Continue and introduce the "Project name"; it is advised to choose a meaningful name that indicates the date when the applications are downloaded. For this example "NightlyExample" is simply chosen. Finally, click on "Finish".

4. Now, place over the red text line "Use the Checkout command..." shown in Figure 40, click right and press "Checkout". Then, it is started the download process, which places the applications inside the folder selected (see Figure 41).
# Annex III: Installation guide and tutorial

**Figure 40.** With "NightlyExample" selected, initiate the download of the Orcc applications.

<table>
<thead>
<tr>
<th>State Name</th>
<th>Date</th>
<th>Rev</th>
<th>Author</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>branches</td>
<td>23-May-2013 18:43:22</td>
<td>2443</td>
<td>mariem.abid</td>
<td>dir</td>
</tr>
<tr>
<td>tags/</td>
<td>23-May-2013 18:43:22</td>
<td>2147</td>
<td>mariem.abid</td>
<td>dir</td>
</tr>
<tr>
<td>trunk/</td>
<td>23-May-2013 18:43:22</td>
<td>2429</td>
<td>mariem.abid</td>
<td>dir</td>
</tr>
</tbody>
</table>

**Figure 41.** Messages shown during the download process.
5. Inside this folder, there are three main folders: "trunk", "branches" and "tags". The ones interesting are "trunk" and "branches". The former is the main release of the applications and the latter are experimental versions; when the experimental versions are completely validated, usually are brought to the main release. For instance, the Figure 42 shows the contents of the "trunk" folder, where there are different projects developed for Orcc such as RVC.

![Figure 42. Content of the “trunk” folder.](image)

12.2.2. Step two - Import the applications to Eclipse and build

1. Open Eclipse and select a workspace (Figure 43). The name chosen here is "exampleWorkspace", but descriptive names are preferred. If it does not exist, it is automatically created.
2. This step is recommended to those people who work with computers without enough good capabilities for this environment (the dynamic memory is the resource most troublesome). Deactivate “Project → Build Automatically”. This gives the chance to build the environment progressively and on demand.

3. Now, the Orcc projects can be imported. For that, please use "File → Import...". A new dialog window appears, select "General → Existing Projects into Workspace" and continue. Now, in "Select root directory" write the path where the desired projects are stored. In this case we are going to select the "trunk" folder, because inside it there are many projects with dependencies between them.

4. In the list that appears, you should select those projects you require. For the first time doing this step or when expecting significant updates, it is advised to import all the projects and then use the tools that Eclipse provides to close and delete the projects that are not required for our target project. In this example, the projects required are the highlighted in orange color in Figure 44; this figure is the result for this tutorial step.

5. Now, it is time to build the projects. You can do either in many steps or in only one with option "Project → Build All" (shortcut key is "Ctrl+B"). This process may last time. If a message indicating an internal error appears with a message like "Java heap space", do not worry, continue and choose that you do not want to exit the workbench. Usually, this error has not severe consequences and it is possible continue working without future problems. The environment is currently ready to work with the Orcc applications and then generate the backends.

12.2.3. Step three - Work with Orcc projects

1. Locate the project and files that you would like to work with. In this example, we are going to work with the decoder within the RVC project that has as "Qualified name" the following text line:

   /RVC/src/devel/org/sc29/wg11/mpegh/part2/Top_mpegh_part2_main.xdf

   If you click on this XDF file, the top diagram of its FU network is shown (see Figure 45).
You can see there are three blocks: “Source”, “HevcDecoder” and “display”. If you double click on “HevcDecoder” you should obtain the internal network diagram for this FU (another XDF diagram, see Figure 46).
If you however double click on “Source” or “display” block, the RVC-CAL code associated to these FUs is displayed. This RVC-CAL code describes the actor that implements the associated FU (Figure 47 shows for example the used in the block “Source”).

When a FU contains more FUs inside it, its internal network is shown as an XDF network. If not, the RVC-VAL code of this actor is shown, because no more FUs are contained.

![Figure 47. RVC-CAL code of the block “Source”.](image)

### 12.2.4. Step four - Generate a backend

1. Click right on the decoder "Top_mpegh_part2_main.xdf" and then press “Run → Run configurations”. A new dialog window appears. Select "ORCC compilation". This window is now the place where the generation of the desired backend is configured and launched.
2. Within the "Compilation settings" tab (see Figure 48) configure:

- **Name**: a relevant name for this configuration. For example, "HEVC_RVC".
- **Project**: the project of the workspace where the decoder is. Here it is "RVC".
- **Select a backend**: the target language of the code of the backend. Depending on the selections, different additional options appear. For our purposes, "C" is selected.
- **Output folder**: the folder where the backend will be stored. It is recommended to create a new folder for each different backend, we have created a new folder called "TutorialExample".
- **Compile XDF network**: mark it. Fill the "XDF name" field by selecting the XDF network associated to the decoder, which here is:
  
  `devel.org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main`

![Compilation settings tab](image)

**Figure 48. Compilation settings tab.**
3. Within the "Compilation options" tab (see Figure 49), in "Default FIFO size" write a value equal or more than 8192 (at least, 13 as power of two). The generated decoders do not function with lower values!

![Compilation options tab]

**Figure 49. Compilation options tab.**

4. Within the "Mapping" tab (Figure 50):

You can see a table that associates the instances of the diagram blocks (on the left) to the correspondent component (on the right). The component represents the processor core where the associated actor must be executed. Because the work is currently done with only one core, all the rows of the component column will have then the same value (c0 to reference the first core). If a multicore distribution should be performed, the different cores of the processors are referenced as: c0, c1, c2, c3, and so on.

It is worth noting that if a FU has more FUs inside and it is associated to a certain core, all the internal FUs are associated to this core as well.
5. The "Common" tab (Figure 51) can be as default, so press on button "Apply" and finally "Run". The process generates on the console a message similar to the shown in Figure 52.
Now, if it is inspected what has been generated on the backend folder, it will be found four folders and two files: bin, build, libs, src, CMakeLists.txt and README.txt. They form the main structure of the C backend. The next steps will consist on how to obtain an executable from these files.

12.2.5. Step five - Use CMake to establish a project for the IDE

1. Open CMake, which displays a window like the shown in Figure 53. Fill the following fields in this way (one important remark: it is different if you would like to establish an Eclipse CDT framework, for this consult the README.txt file of the backend):

   - "Where is the source code": the folder that was chosen to generate the backend.
   - "Where to build the binaries": the folder "build" contained in the previous folder.
2. Press "Configure". In the new dialog window check "Use default native compilers" and select the generator for this project.

The options for the generator in Ubuntu are: Unix Makefiles, CodeBlocks - Unix Makefiles, EclipseCDT4 - Unix Makefiles, and KDevelop3 - Unix Makefiles. The first one uses the building tools provided by the operating system (makefiles, GCC compiler, etc.). The last three establish a framework for the correspondent IDEs.

The options for the generator in Windows are more; among others there is support for EclipseCDT4, CodeBlocks and several versions of Visual Studio.

Because we are working with Ubuntu and this is a general tutorial, the option "Unix Makefiles" is selected. Finally, press "Finish". A similar result to the one in Figure 54 is obtained.
3. On the table highlighted in red on Figure 54 ensure that the values of these variables are correct. With the options that have been selected in the tutorial, you usually should fill only the “CMAKE_BUILD_TYPE” variable: you can write “DEBUG” or “RELEASE” (“DEBUG” is selected for this tutorial).

4. Press again "Configure" and if it is all ok, press "Generate".

**12.2.6. Step six - Use the preferred IDE**

This step consists on using the preferred IDE. Because the previous step was explained using the "Unix Makefiles", the explanation continues for such an option. Anyway, this step can be done with the preferred IDE (after open/import the files of the project in this IDE). One large example of the use of other IDE was done in the explanation of the software migration procedure presented in Chapter 6, where CCS is employed. Finally, for "Unix Makefiles":

![Figure 54. Configuration of the building environment for the decoder with CMake.](image-url)
1. By means of the operating system terminal, execute the command "make" within the folder "TutorialExample/build". If all is right, this process starts the compilation and generates the decoder solution for this Linux-based operating system, which is stored in folder "TutorialExample/bin". The start and the end of the generation process have been shown respectively in Figure 55 and Figure 56.

Figure 55. Execution of “make” and start of the generation process.

Figure 56. End of the generation process; the decoder solution has been built.

2. Now, if the generation process was successful, the decoder solution can be executed. To do this, type in the command prompt the following command:

<path to the decoder application> -i <path to the sequence to decode>
Located in the folder where both the decoder solution and the bitstream to decode are, an example of the command in Ubuntu can be (see also Figure 57):

```
./Top_mpegh_part2_main -i 'ld_example_sequence '
```

If the decoder solution has not any problem messages like the shown in Figure 57 will appear. Furthermore, as shown in Figure 58, the video sequence decoded will be displayed on the screen.

Figure 57. Execution of the decoder, the number of fps decoded is shown in the console.

Figure 58. Execution of the decoder, the video sequence decoded is displayed.

Finally, it is worth mentioning that for experimenting with multicore distributions using the GPP systems indicated in Chapter 4 (PC and PandaBoard), an additional option must be used in the execution of the command. This option takes the XCF file in
order to known how must be performed the distribution of actors among the cores. The command in this case will be:

```bash
<path to the decoder application> -i <path to the sequence to decode> 
-m <path to the XCF file with the distribution of actors among the cores>
```

For more details, consult the tutorial presented in [3].