Development and Validation of NESSIE:
a multi-criteria performance estimation
Tool for SoC
Acknowledgments

I would like to thank my supervisors Prof. Frédéric Robert and Prof. François Horlin for providing me with this interesting thesis and for their guidance all along my thesis. I also thank them for setting up a collaboration between IMEC and the ULB.

I would like to thank Prof. Dragomir Milojevic for his involvement in my PhD, its technical support during the thesis via the collaboration of the NTUA.

I am grateful to Bruno Bougard, Praveen Raghavan, David Novo, Min Li and Tom Vander Aa, from the implementation department of IMEC, for integrating me in their team and for their availability in technical support.

I also want to thank Antonis Papanikolaou, Alexis Bartzas, Dimitri Soudris and Kostas Siozios, from the NTUA, for their availability and important collaboration during the thesis.

I am also grateful to Prof. Pierre Mathys for hosting me in the Bio, Electro and Mechanical Systems department at the ULB and for transmitting to me the passion for research.

A special thanks goes to my colleagues and friends at ULB: Alexis, Vincent, Geoffrey, Anthony, Axel, Manu, Max, Nico for cheering me up, sharing nice moments and for their advices.

A particular thanks to Emmanuelle for her support, constant energy and the constructive exchanges we have.

I would like also to thank Joris for the interesting discussions and for cheering me up during the last year.

My deepest gratitude goes to my parents who gave me the opportunity to succeed during my studies and for their encouragement to achieve the present thesis, and to my brothers for all the moments we share.

A special thanks goes to my step-parent for giving me support and their constant interest in my work.

Thank you to my friends, Audrey, Valentine, Christelle, Sarah, François, Chris, Laurent, who contribute to make me enjoy the life.

Last but not least, I am greatly indebted to my husband, Frédéric, for all his support, presence and devotion during these four years. Every day is fuelled by our happiness and love. Thank you. I know I can always count on you.
Abstract

The work presented in this thesis aims at validating an original multicriteria performances estimation tool, NESSIE, dedicated to the prediction of performances to accelerate the design of electronic embedded systems.

This tool has been developed in a previous thesis to cope with the limitations of existing design tools and offers a new solution to face the growing complexity of the current applications and electronic platforms and the multiple constraints they are subjected to.

More precisely, the goal of the tool is to propose a flexible framework targeting embedded systems in a generic way and enable a fast exploration of the design space based on the estimation of user-defined criteria and a joint hierarchical representation of the application and the platform.

In this context, the purpose of the thesis is to put the original framework NESSIE to the test to analyze if it is indeed useful and able to solve current design problems. Hence, the dissertation presents:

- A study of the State-of-the-Art related to the existing design tools. I propose a classification of these tools and compare them based on typical criteria. This substantial survey completes the State-of-the-Art done in the previous work. This study shows that the NESSIE framework offers solutions to the limitations of these tools.

- The framework of our original mapping tool and its calculation engine. Through this presentation, I highlight the main ingredients of the tool and explain the implemented methodology.

- Two external case studies that have been chosen to validate NESSIE and that are the core of the thesis. These case studies propose two different design problems (a reconfigurable processor, ADRES, applied to a matrix multiplication kernel and a 3D stacking MPSoC problem applied to a video decoder) and show the ability of our tool to target different applications and platforms.

The validation is performed based on the comparison of a multi-criteria estimation of the performances for a significant amount of solutions, between NESSIE and the external design flow. In particular, I discuss the prediction capability of NESSIE and the accuracy of the estimation.

The study is completed, for each case study, by a quantification of the modeling time and the design time in both flows, in order to analyze the gain achieved by
our tool used upstream from the classical tool chain compared to the existing design flow alone.

The results showed that NESSIE is able to predict with a high degree of accuracy the solutions that are the best candidates for the design in the lower design flows. Moreover, in both case studies, modeled respectively at a low and higher abstraction level, I obtained a significant gain in the design time.

However, I also identified limitations that impact the modeling time and could prevent an efficient use of the tool for more complex problems.

To cope with these issues, I end up by proposing several improvements of the framework and give perspectives to further develop the tool.
This thesis presents the results of my research; part of this work has been published in the following conference papers:


# Contents

Acknowledgments iii

Abstract v

Publication list vii

List of Acronyms xix

Introduction xxi

Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxii

1 Context and motivation 1

1.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Embedded systems design . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2.1 Functional constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2.2 Non-functional constraints . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.2.3 Technologies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.2.4 Platforms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.3 Industrial solutions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.4 An original solution: NESSIE . . . . . . . . . . . . . . . . . . . . . . . . . . 8

Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2 State of the Art 13

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.2 Literature survey . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.2.1 Previous survey . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.2.2 System-level tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.2.3 Integrated tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

2.2.4 Exploration tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

2.2.5 IP reuse environments . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

2.3 Tools comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

2.3.1 Space Exploration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

2.3.2 Simulations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

2.3.3 Functional verification . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
CONTENTS

4.6.1 Models validity .......................................................... 122
4.6.2 Results reproducibility : NESSIE vs DRESC .................. 127
4.6.3 Performance prediction based on multiple criteria .......... 136
4.7 Tools integrations .......................................................... 147
4.7.1 DRESC timing ............................................................ 148
4.7.2 NESSIE timing ............................................................ 149
4.7.3 Integrated flow ........................................................... 151
4.8 Conclusions ................................................................. 152
Bibliography ................................................................. 156

5 Case study 2 : 3D Stacking Paradigm 159
5.1 Introduction ............................................................... 159
5.2 Description of the 3D design problem ............................... 161
5.2.1 The 3D design flow .................................................... 161
5.2.2 Description of the case study ....................................... 163
5.3 Modeling of the case study ............................................. 166
5.3.1 The models ............................................................. 166
5.3.2 The scenarios .......................................................... 171
5.4 Results and discussion .................................................. 173
5.4.1 General trend .......................................................... 173
5.4.2 Prediction ............................................................... 175
5.5 Design time vs performance estimation ............................. 178
5.6 Conclusion ................................................................. 180
Bibliography ................................................................. 182

6 Conclusions and Future Work 183
6.1 Limitations of NESSIE .................................................... 184
6.2 Comparison with SoA tools ............................................. 187
6.3 Perspectives ............................................................... 189
6.4 Conclusions ............................................................... 192
Bibliography ................................................................. 193

A A complement to the IMEC case study 195
A.1 TCL/TK developments .................................................. 195
A.1.1 The timeline .......................................................... 195
A.1.2 The data token routing .............................................. 196
A.2 NESSIE Pareto graphs .................................................. 196
A.3 DRESC Pareto graphs .................................................... 197
A.3.1 Cycles vs Area ......................................................... 197
A.3.2 Cycles vs effective IPC ............................................. 197
A.3.3 Cycles vs ED .......................................................... 198
A.3.4 Area vs effective IPC ............................................... 198
A.3.5 Area vs ED ............................................................ 198
A.3.6 Cycles and Area vs effIPC/ED ..................................... 198
B A complement to the 3D stacking 211
B.1 Data tables .......................................................... 211
   B.1.1 Bus Load values .............................................. 211
   B.1.2 Data size values .............................................. 211
   B.1.3 Wire Lengths .................................................. 211
B.2 Ordered bar graphs .............................................. 213
   B.2.1 NESSIE results ............................................... 213
   B.2.2 NTUA results ................................................ 214
C A complement to the Future Work 221
C.1 Multiobjective exploration module .............................. 221
C.2 Graphical user interface ........................................ 223
   C.2.1 Generation of inputs ........................................ 223
   C.2.2 Processing of outputs ...................................... 223
   C.2.3 Other functionalities ....................................... 224
D Miscellaneous 227
## List of Tables

2.1 Classification of the 19 tools regarding eight criteria (automatic exploration, multi-criteria simulation, functional verification, synthesis, multiple AL, SW and HW description and mapping policy) .................................................. 16

2.2 Comparison of the 15 additional design tools based on eight criteria (automatic exploration, multi-criteria simulation, functional verification, synthesis, multiple AL, SW and HW description and mapping policy) ................. 32

2.3 Comparison of the 15 design tools based on their description languages ................................................. 34

4.1 Primitives states of the Model 1 .................................................. 105

4.2 Primitives states of the Model 2 ............................................ 105

4.3 Primitives states of the Model 3 ............................................ 106

4.4 Primitives states of the Model 4 ............................................ 106

4.5 CGA architecture variants defined for the exploration ................................................................. 121

4.6 Simulations models of the CGA architecture ................................................................. 123

4.7 Pareto solutions related to the cycles and the area .................................................. 137

4.8 Pareto solutions related to the cycles and the effIPC ........................................ 140

4.9 Pareto solutions related to the cycles and the ED .................................................. 140

4.10 Pareto solutions related to the area and the effIPC ........................................ 140

4.11 Pareto solutions related to the area and the ED .................................................. 144

4.12 NESSIE preselected solutions for cycles constraint below 80 ........................................ 145

4.13 Estimated criteria for preselected solutions - DRESC results ........................................ 145

4.14 Preselected solutions in DRESC flow for cycle constraint ........................................ 145

4.15 NESSIE preselected solutions for ED constraint superior to 50% ........................................ 147

4.16 Preselected solutions in DRESC flow for ED constraint ........................................ 147

4.17 Timing values of the classical flow (DRESC) and the integrated flow ........................................ 151

5.1 Parameters values for the wire power consumption .................................................. 170

5.2 Comparison of the preselected NESSIE solutions with the best solutions identified in the 3D external flow for the CIF resolution .................................................. 177

5.3 Comparison of the preselected NESSIE solutions with the best solutions identified in the 3D external flow for the 4CIF resolution .................................................. 178

5.4 Comparison of the preselected NESSIE solutions with the best solutions identified in the 3D external flow for the HDTV resolution .................................................. 178
6.1 Comparison of NESSIE with three SoA design tools - Daedalus, GASPARD and SynDEx - regarding eight criteria (automatic exploration, multi-criteria simulation, functional verification, synthesis capability, multiple AL, SW and HW targets, mapping policy) ........................................ 188

B.1 Data split - bus load in bytes .................................................. 212
B.2 Functional split - bus load in bytes ........................................... 212
B.3 Hybrid - bus load in bytes ....................................................... 213
B.4 Data split - nodes data exchange .............................................. 213
B.5 Functional split - nodes data exchange ....................................... 214
B.6 Hybrid - nodes data exchange ................................................... 214
B.7 Length of the NoC wires estimated by Cadence SoC Encounter and used in NESSIE - architecture variants V1 to V3 ........................................ 215
B.8 Length of the NoC wires estimated by Cadence SoC Encounter and used in NESSIE - architecture variants V4 to V6 ........................................ 216
B.9 Length of the NoC wires estimated by Cadence SoC Encounter and used in NESSIE - architecture variants V7 and V8 ........................................ 217
D.1 Correspondence between states names and IDs in NESSIE ................. 228
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Use of a High Level performances prediction framework in classical design</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>flows to reduce the design time</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>Design steps in a top down design flow</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Illustration of the mapping in a top-down design step</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Relative potential position of NESSIE compared to classical design flows</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>ARTS Framework</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Cofluent methodology</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>CoWare Platform Architect</td>
<td>22</td>
</tr>
<tr>
<td>2.4</td>
<td>Daedalus Framework</td>
<td>23</td>
</tr>
<tr>
<td>2.5</td>
<td>Topcased framework</td>
<td>25</td>
</tr>
<tr>
<td>2.6</td>
<td>GASPARD methodology</td>
<td>27</td>
</tr>
<tr>
<td>2.7</td>
<td>MARTE UML profile</td>
<td>28</td>
</tr>
<tr>
<td>2.8</td>
<td>Hardware-dependent software design methodology</td>
<td>29</td>
</tr>
<tr>
<td>2.9</td>
<td>M3SCoPE framework</td>
<td>30</td>
</tr>
<tr>
<td>2.10</td>
<td>M3Explorer tool</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Hierarchical structure of YETi</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>Tree representation of an analytical rule in YETi</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>UML class diagram of the generic rule in the YETi engine</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>UML diagram of the relation class in the YETi engine</td>
<td>43</td>
</tr>
<tr>
<td>3.5</td>
<td>Tree representation of an oriented behaviour</td>
<td>44</td>
</tr>
<tr>
<td>3.6</td>
<td>UML diagram of the behaviour class in the YETi engine</td>
<td>44</td>
</tr>
<tr>
<td>3.7</td>
<td>Schematic view of the YETi framework</td>
<td>45</td>
</tr>
<tr>
<td>3.8</td>
<td>Schematic view of the NESSIE framework</td>
<td>46</td>
</tr>
<tr>
<td>3.9</td>
<td>Illustration of a Petri network and its components</td>
<td>47</td>
</tr>
<tr>
<td>3.10</td>
<td>Use of Dummy nodes in a NESSIE’s Petri network</td>
<td>49</td>
</tr>
<tr>
<td>3.11</td>
<td>Hierarchical representation of the platform in NESSIE</td>
<td>51</td>
</tr>
<tr>
<td>3.12</td>
<td>Timeline building during the mapping process in NESSIE</td>
<td>52</td>
</tr>
<tr>
<td>3.13</td>
<td>Scheduling and allocation policy in NESSIE</td>
<td>53</td>
</tr>
<tr>
<td>3.14</td>
<td>Routing process in NESSIE</td>
<td>56</td>
</tr>
<tr>
<td>4.1</td>
<td>A schematic view of the global design flow for 3GPP MF MPSoC systems</td>
<td>65</td>
</tr>
<tr>
<td>4.2</td>
<td>Main components of a wireless receiver</td>
<td>66</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Design flow of the ADRES processor - from the ANSI C to the Gate Level</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Architecture of the ADRES processor</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Hardware components of the ADRES processor - from left to right: functional unit, register files, transition node, constant memory</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Detail of a functional unit (FU) structure and its local register files (PRF and DRF)</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>Interconnection schemes between FUs and RFs: (a) fully distributed RFs; (b) no distributed RF; (c) partially distributed RFs; (d) FUs Mesh; (e) FUs Meshplus; (f) FUs Morphosys</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>Interconnection detail between FU and local RF: (a) REG_INDEX3 interconnection scheme; (b) REG_INDEX2 interconnection scheme</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>Illustration of the software pipelining technique: (a) example of loop; (b) pseudo-instruction of the loop body; (c) loop pipeline</td>
<td></td>
</tr>
<tr>
<td>4.10</td>
<td>Example of a loop data dependence graph (DDG) with no inter-iteration dependence</td>
<td></td>
</tr>
<tr>
<td>4.11</td>
<td>Illustration of the modulo scheduling on a 3X3 array</td>
<td></td>
</tr>
<tr>
<td>4.12</td>
<td>Example of a MRT with 2 available slots</td>
<td></td>
</tr>
<tr>
<td>4.13</td>
<td>Kernel-only representation of the software-pipelined loop</td>
<td></td>
</tr>
<tr>
<td>4.14</td>
<td>Kernel-only representation of the loop on a 3X3 array</td>
<td></td>
</tr>
<tr>
<td>4.15</td>
<td>Characteristic metrics of a software-pipelined loop</td>
<td></td>
</tr>
<tr>
<td>4.16</td>
<td>Stages in a software-pipelined loop</td>
<td></td>
</tr>
<tr>
<td>4.17</td>
<td>Design steps of the DRESC compiler</td>
<td></td>
</tr>
<tr>
<td>4.18</td>
<td>Screenshot of the Schedule Viewer for a 2X2 array</td>
<td></td>
</tr>
<tr>
<td>4.19</td>
<td>Formalization of the design flow of ADRES - the Precompiler and DRESC</td>
<td></td>
</tr>
<tr>
<td>4.20</td>
<td>Levels of abstraction (AL) modeling inner loops and CGA structures</td>
<td></td>
</tr>
<tr>
<td>4.21</td>
<td>Validation framework of the IMEC case study</td>
<td></td>
</tr>
<tr>
<td>4.22</td>
<td>Models of for loops with Petri Networks in NESSIE</td>
<td></td>
</tr>
<tr>
<td>4.23</td>
<td>C algorithm of the matrix multiplication</td>
<td></td>
</tr>
<tr>
<td>4.24</td>
<td>Illustration of the inner loop of the function, the computational-intensive part of the code</td>
<td></td>
</tr>
<tr>
<td>4.25</td>
<td>DDG of the inner loop of the matrix multiplication function</td>
<td></td>
</tr>
<tr>
<td>4.26</td>
<td>Pseudo representation of the operations parallelism in the inner loop</td>
<td></td>
</tr>
<tr>
<td>4.27</td>
<td>Petri Net of the inner loop of the matrix multiplication function with feedback</td>
<td></td>
</tr>
<tr>
<td>4.28</td>
<td>Developed Petri Net of the inner loop of the matrix multiplication function</td>
<td></td>
</tr>
<tr>
<td>4.29</td>
<td>Illustration of the inner loop unrolled 4 times</td>
<td></td>
</tr>
<tr>
<td>4.30</td>
<td>Pseudo representation of the operations parallelism in the inner loop unrolled 4 times - in red: illustration of the multiple use of data</td>
<td></td>
</tr>
<tr>
<td>4.31</td>
<td>Illustration of the use of dummy nodes to manage temporary data - (a) when data are explicitly stored in RF; (b) when data are stored in FU</td>
<td></td>
</tr>
<tr>
<td>4.32</td>
<td>Number of cycles of 33 solutions simulated in NESSIE for the Model1</td>
<td></td>
</tr>
<tr>
<td>4.33</td>
<td>Number of cycles of the 75 solutions simulated in NESSIE for the Model2</td>
<td></td>
</tr>
<tr>
<td>4.34</td>
<td>Activity report of the 2X2 mesh with shared RF and the unroll1 loop scenario</td>
<td></td>
</tr>
<tr>
<td>4.35</td>
<td>Comparison of the Model1 and Model2 on the 33 shared RF CGA variants</td>
<td></td>
</tr>
<tr>
<td>4.36</td>
<td>Number of cycles of the 75 solutions compiled in DRESC</td>
<td></td>
</tr>
</tbody>
</table>
4.37 Schedule viewer of the 3X3 mesh with shared register variant for the unroll1 loop scenario .................................................. 130
4.38 Schedule viewer of the 3X3 mesh with CON1 distributed RF variant for the unroll1 loop scenario .................................................. 131
4.39 Schedule viewer of the 3X3 mesh with CON2 distributed RF variant for the unroll1 loop scenario .................................................. 132
4.40 Comparison of the cycles between the DRESC and NESSIE results for the Model1 ........................................................................ 133
4.41 Comparison of the cycles between the DRESC and NESSIE results for the Model2 ........................................................................ 133
4.42 Nessie results : cycles vs area (model2) ........................................................................ 138
4.43 Nessie results : cycles vs ipc (model2) ........................................................................ 139
4.44 Nessie results : cycles vs ED (model2) ........................................................................ 141
4.45 Nessie results : area vs IPC (model2) ........................................................................ 142
4.46 Nessie results : area vs IPC (model2) ........................................................................ 143
4.47 Nessie results: Area - IPC - Cycles ........................................................................ 144
4.48 Relative error (in %) between NESSIE and DRESC for the cycles ......................... 146
4.49 Relative error (in %) between NESSIE and DRESC for the cycles ......................... 147
4.50 Timings introduced in the DRESC flow .................................................................... 148
4.51 Timings introduced in NESSIE ............................................................................... 149
4.52 Timings introduced in the flow integrating NESSIE with DRESC ......................... 150
4.53 Comparison of the simulation time in NESSIE (red) and in DRESC (blue) for different unrolling factors, four CGA architectures with shared RF (2x2 mesh, 3x3 mesh, 4x4 mesh and 4x4 mesh+) and a fixed 64x64 matrix size 153
4.54 Comparison of the simulation time in NESSIE (red) and in DRESC (blue) for different matrix sizes and a fixed unrolling factor (8) and CGA (3X3 mesh shared RF) .................................................. 153
4.55 Comparison of the design space exploration in DRESC (a) and in NESSIE (b) ........................................................................ 154
5.1 Classical design flow of 3D stacking SoC .................................................................. 161
5.2 Architecture of the MPSoC with its newtork-on-chip interconnection .................. 164
5.3 Schematic representation of an AVC/H.264 video decoder ................................. 165
5.4 Levels of abstraction (AL) modeling the video decoder and the 3D MPSoC structures ........................................................................ 167
5.5 Schematic representation of the MPSoC platform in NESSIE ............................... 168
5.6 Wires power consumption of eight MPSoC variants and three video decoder scenarios for the CIF resolution ................................................. 173
5.7 Wires power consumption of eight MPSoC variants and three video decoder scenarios for the 4CIF resolution ................................................. 174
5.8 Wires power consumption of eight MPSoC variants and three video decoder scenarios for the HDTV resolution ................................................. 174
5.9 Relative error (in %) between NESSIE and NTUA results for the wires power consumption- CIF resolution ................................................. 175
5.10 Relative error (in %) between NESSIE and NTUA results for the wires power consumption- 4CIF resolution ........................................... 176
5.11 Relative error (in %) between NESSIE and NTUA results for the wires power consumption- HDTV resolution ........................................... 176
5.12 Timings of the flow integrating NESSIE to the external 3D stacking tool chain ................................................................. 179

6.1 Relative potential position of NESSIE compared to classical design flows . . 186
6.2 Framework integrating MCDA and automatic generation and exploration of scenarios to the NESSIE engine ................................. 190

A.1 TCL/TK interface of the Timeline - table representation ........................ 195
A.2 TCL/TK interface of the Timeline - pipelined view .............................. 196
A.3 Screenshot of the data token informations - textual format .................. 197
A.4 TCL/TK interface of the data token routing - table representation ........... 198
A.5 Nessie results: pareto of area vs cycles (3D) ...................................... 199
A.6 Nessie results: pareto of ipc vs cycles (3D) ........................................ 200
A.7 Nessie results: pareto of ipc vs cycles (3D) ........................................ 201
A.8 DRESC results: pareto of area vs cycles (2D) ....................................... 202
A.9 DRESC results: pareto of effIPC vs cycles (2D) ..................................... 203
A.10 DRESC results: pareto of ED vs cycles (2D) ....................................... 204
A.11 DRESC results: pareto of area vs effIPC (2D) ...................................... 205
A.12 DRESC results: pareto of area vs ED (2D) ......................................... 206
A.13 DRESC results: pareto of area, effIPC vs cycles (3D) ......................... 207
A.14 DRESC results: projection of the 3D pareto graph - area vs cycles .......... 208
A.15 DRESC results: projection of the 3D pareto graph - effIPC vs cycles ...... 209
A.16 DRESC results: projection of the 3D pareto graph - ED vs cycles ........... 210

B.1 Ordered NESSIE results of the wires power consumption (in uW) for CIF data resolution ................................................................. 218
B.2 Ordered NESSIE results of the wires power consumption (in uW) for 4CIF data resolution ................................................................. 218
B.3 Ordered NESSIE results of the wires power consumption (in uW) for HDTV data resolution ................................................................. 219
B.4 Ordered NTUA results of the wires power consumption (in uW) for CIF data resolution ................................................................. 219
B.5 Ordered NTUA results of the wires power consumption (in uW) for 4CIF data resolution ................................................................. 220
B.6 Ordered NTUA results of the wires power consumption (in uW) for HDTV data resolution ............................................................... 220

C.1 Framework integrating MCDA and automatic generation and exploration of scenarios to the NESSIE engine ................................. 222
C.2 Pre-processing graphical interface - generation of input XML files .......... 224
C.3 Post-processing graphical interface - Activity Report ............................ 225
### List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>ASIP</td>
<td>Application Specific Instruction Processor</td>
</tr>
<tr>
<td>AVC</td>
<td>Advanced Video Coding</td>
</tr>
<tr>
<td>CFSM</td>
<td>Codesign Finite State Machine</td>
</tr>
<tr>
<td>CIF</td>
<td>Common Image Format</td>
</tr>
<tr>
<td>CPI</td>
<td>Cycles Per Instruction</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processor Unit</td>
</tr>
<tr>
<td>CDFG</td>
<td>Control Data Flow Graph</td>
</tr>
<tr>
<td>CSDIF</td>
<td>Cyclo-Static Data Flow</td>
</tr>
<tr>
<td>CSP</td>
<td>Communicating Sequential Processes</td>
</tr>
<tr>
<td>CT</td>
<td>Continuous Time</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete cosine transform</td>
</tr>
<tr>
<td>DE</td>
<td>Discrete Event</td>
</tr>
<tr>
<td>DFG</td>
<td>Data Flow Graph</td>
</tr>
<tr>
<td>DoF</td>
<td>Degree of Freedom</td>
</tr>
<tr>
<td>EMIF</td>
<td>External Memory Interface</td>
</tr>
<tr>
<td>FCFS</td>
<td>First Come First Served</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GPP</td>
<td>General Purpose Processor</td>
</tr>
<tr>
<td>HCDIFG</td>
<td>Hierarchical Control Data Flow Graph</td>
</tr>
<tr>
<td>HDTV</td>
<td>High Definition Television</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>IDCT</td>
<td>Inverse Discrete cosine transform</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>IPC</td>
<td>Instructions Per Cycle</td>
</tr>
<tr>
<td>KPN</td>
<td>Kahn Process Network</td>
</tr>
<tr>
<td>MTRPLU</td>
<td>Maximum Transmission Rate Per Length Unit</td>
</tr>
<tr>
<td>MIPS</td>
<td>Million Instructions Per Second</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>MPSOC</td>
<td>Multiple Processor System-on-Chip</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>MPU</td>
<td>Micro-Processor Unit</td>
</tr>
<tr>
<td>NIU</td>
<td>Network Interface Unit</td>
</tr>
<tr>
<td>NOC</td>
<td>Network On Chip</td>
</tr>
<tr>
<td>OBIS</td>
<td>Optimal Buffer Insertion and Sizing</td>
</tr>
<tr>
<td>PN</td>
<td>Petri Networks</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SAIF</td>
<td>Switching Activity Interchange Format</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>TLM</td>
<td>Transaction Level Modeling</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread Level Parallelism</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuits</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
Introduction

Look back ten or twenty years ago and remember the state of the embedded electronic market in this not so distant past. Now, look at the present full-technology world we are living in.

For some years, we faced an incredible evolution of the electronic devices to offer an efficient support to more and more complex applications and follow the continual progress of the market demand.

However, if the research constantly proposes new technologies to improve the performance of the devices and to be able to target these growing applications, designers and design tools do not evolve as quickly as they should, which prevents the industries from creating efficient solutions (satisfying the specifications) meeting time-to-market and costs constraints.

Moreover, embedded systems add multiple constraints that must be taken into account during design time and require trade-offs that are not easy to manage: energy consumption, execution time, chip area, thermal constraint,...

To face these design challenges, new design tools and methodologies have appeared and are still developed to enable a better exploration of the design space (Multi-Criteria Decision Aid, joint exploration of the application and the platform,...), an earliest estimation of the performances (e.g. new high levels of abstraction in the design flow), and a reduction of the design time (less iterations during the design) and the cost.

However, in his thesis\(^1\), Alexis Vander Biest has shown that, if these tools implement some of these solutions, they also have limitations that could be improved (e.g. monocrterion simulation, partial exploration of the system, hard-coded policies,...).

This is why he has developed an original tool, NESSIE, coupled with a flexible model evaluation engine, YETi, to cope with these limitations and offer an early, generic and fast prediction of the performances based on a multicriteria exploration and simulation of solutions.

The goal of the work presented in this dissertation is to validate NESSIE in its current version and to analyze if the original and generic methodology implemented in the framework is useful in the following embedded system design problematic. The conclusion

\(^1\)\[1\], finished in 2008.
of the work will answer the following questions: does NESSIE work and does it make sense to further develop the tool?

Therefore, the thesis has consisted in the choice of two external and representative case studies with their design flow. It has resulted in the following candidates: the ADRES design flow at the Interuniversity Microelectronics Center (IMEC) and a 3D stacking design flow at the National Technical University of Athens (NTUA).

The objectives of the work are to demonstrate the modeling capability of the tool on different design problems, to analyze the multicriteria prediction efficiency, to identify the limitations of NESSIE and to discuss the modeling and design time gain resulting from these examples compared to existing design tools.

In this work, we have mainly contributed to the formalization of real design problems via the modeling of two external design flows, to a deep analysis of the usefulness of NESSIE based on these case studies and to the proposal of the future ingredients that will lead to a complete and competitive framework.

Thesis organization

The thesis is organized as follows:

Chapter 1 introduces the context of the thesis.

Chapter 2 defines the state of the art related to performance estimation tools and methods.

Chapter 3 describes the concepts and implementation of NESSIE and YETi.

Chapter 4 presents the main case study.

Chapter 5 presents the second case study.

Chapter 6 identifies future opportunities for the development of our tool and concludes this thesis.

Bibliography

Chapter 1

Context and motivation

Abstract

In this first chapter, we introduce the context of this work and why there is a need for an efficient performance prediction tool. We start by explaining how Nessie is related to classical design flows and detail typical top-down tool chains. To highlight the high constraints and the wide design space designers must face when dealing with electronic embedded systems, we explain the different aspects that characterize SoCs design. Then, we present the solutions that are typically implemented in industrial flows to find satisfying solutions. For these solutions, we position our original tool in the SoC design problematic. Finally, we explain the methodology that we have followed in this thesis to validate this tool and show how this work can contribute to its application in real embedded system design problems.

1.1 Introduction

The concern of this work is the reduction of the design time, and thus the design cost, induced in the classical top-down flows which target in particular complex applications for Systems-on-Chip (SoC’s). This reduction of the design time must be considered together with the efficiency of the designed solution. To cope with the limitations of classical tools, we believe that an efficient way to achieve this goal is to use a High-Level tool able to predict faster and a priori the performances of the targeted system, based on a multicriteria exploration of the solutions. We illustrate this in the figure 1.1. The idea is to base the exploration on the explicit modeling of both the application (SW\(^1\) on the figure) and the platform (HW), and on a flexible core able to map these two parts together at different possible levels of abstraction and addressing different design problems.

\(^1\)In this context, SW is used widely to represent the functionality of a system and is not restricted to processor-like platforms.
In a classical top-down design flow, the system is designed progressively, starting from a high level of abstraction where the system is described roughly to lower levels of abstraction where we have an accurate description of the electronic system. The path from one level to the other constitutes the refinement of the system. During the refinement, complementary information is added progressively to have a more detailed description of the structure.

In this work, we define a system as the entity composed of the electronic platform, or the architecture of the chip, and the functionality, or the application, that will be executed by the platform. It is important to distinguish the two parts as they can be defined separately and both impact the performances of the final system. When we talk about performances, we mean the set of criteria that have to be optimized when designing the system\(^2\).

The figure 1.2 represents the different components that compose a generic top-down flow. Three levels of abstraction are represented. As illustrated on the figure, a design step (e.g. the Level N+1) is composed of a decision tool which takes as inputs:

- the description of the system (ideally, the application and the platform) generated at the previous level (Level N);
- the specifications for the design (desired performances, constraints,...);
- criteria models on which the tool will decide;
- the primitives that will compose the refined description of the system.

\(^2\)The term *performance* is generally related to the execution time of an application. Here, we use the plural term *performances*.
The tool performs a mapping of the $N+1$ primitives of the functionality on the $N+1$ primitives of the platform. This mapping, illustrated in figure 1.3, consists in the allocation and the scheduling of tasks on hardware components in order to optimize criteria based on models (e.g. analytical models) fed by input parameters. When the tool has generated a solution, i.e. a structure for the platform and the functionality at the Level $N+1$, a simulation can be performed to evaluate the performances of the solution. If these performances satisfy the specifications, then the solution is refined at the next level (Level $N+2$), otherwise, the designer needs to iterate and change the choices made at higher levels of abstraction.

As illustrated on the right of the figure 1.2, as the description of the system is more complex when going down in the flow, the design step and the simulations are more time consuming. For this reason, it is important to avoid late iterations. We will see later that in practice, the tool chains are partially constituted of these ingredients.

The goal of a High Level performance prediction tool is to reduce the number of iterations and thus decrease significantly the design time. Regarding the existing frameworks (see chapter 2), NESSIE is an original solution, developed in our research department\(^3\), in order to explore faster and more efficiently the design space while allowing to target different design problems thanks to a general and flexible engine. Through this thesis, we have decided to use this tool and to put it to the test to see if we can improve the design of SoC’s.

To better understand why we have develop our own tool, it is important to detail the problematic of the electronic embedded systems design and the solutions that we have today.

### 1.2 Embedded systems design

The electronic field is expanding in a variety of domains ranging from the avionics, the automotive to the telecommunications, the multimedia, via biomedical or space applications, we can find embedded electronic systems that all have specific or common demands. They are composed of the following characteristics.

#### 1.2.1 Functional constraints

Applications that must be executed on todays embedded systems, e.g. multimedia devices or mobile phones, significantly increase in complexity and are data intensive. Moreover, these embedded systems are dedicated to different types of applications that have to run on the same chip.

Designers must also face the constant emergence of new standards (communications standards, data format, ...).

---

\(^3\)In the Embedded Electronics Research Unit of the Bio, Electro and Mechanical Systems Lab at ULB.
Figure 1.2: Design steps in a top down design flow
1.2. EMBEDDED SYSTEMS DESIGN

This has become a real challenge for the designers to design systems, i.e. define the right description of the application and choose the optimal platform, with respect to the desired standards and the constrained specifications.

1.2.2 Non-functional constraints

Besides these functional demands, the embedded systems add critical non-functional constraints that require trade-offs. The devices have to be compact, non bulky and for some applications almost invisible. The need of autonomy requires to minimize the energy consumption. For some applications, the systems have to be highly reactive and have strict real time constraints. Others have to support hard environment conditions (high temperature, radiations, ...).

These considerations have to be addressed together with the market demand which requires reliable devices, minimum design time and strict cost constraints.

All these conflicting aspects have to be taken into account by the designers that must find the right application/platform couple to optimize the system depending on the final use.

To support this demand, research continuously offers new languages, hardware solutions and technologies and improved design flows that enable the creation of embedded systems that could satisfy the market requests.

1.2.3 Technologies

Moore’s law is a well-known guide that pushes electronics researchers to find ways to decrease the transistors size more and more. As we reach the physical limits of the transistors, new technologies continuously appear on the market to face Moore’s challenge,
with different degrees of maturity. However, design tools do not evolve as fast as the technology they must handle. Moreover, they are not adapted to deal with the new design problems that emerge with these latter. This requires the development of new tools, that are often specialized for the design of dedicated systems, which takes time and is expensive for designers.

1.2.4 Platforms

The number of possible hardware components available to build an electronic platform represents a great amount of potential solutions to design a system: ASIC (Application Specific Integrated Circuits), processors, FPGA (Field programmable Gate Array), SoCs, MPSoCs, ... are so many candidates that have to be chosen by the designer or the decision tool. At a lower level, the decision must be done on the number of transistors, the memory types and topology, the interconnections scheme, the computing nodes, their floorplanning,... In addition, other parameters like the clock frequency, the power supply, the instruction set could be degrees of freedom which will have to be chosen during the design of the system.

To handle this generation of embedded systems, in particular the system-on-chip, industries use different solutions that satisfy some of the constraints but have several limitations.

1.3 Industrial solutions

As we have shown above, the todays embedded systems are really complex and the numerous degrees of freedom offered by the applications, the technologies, the platforms, represent a great design space. To handle it, several solutions can be used. By mixing these solutions, companies and researchers are able to design the desired systems more or less efficiently.

Design space exploration When we talk about design space, it of course suggests that an exploration of solutions is needed. In this context, the exploration time must be fair with respect to the time-to-market constraint. This time is directly linked to the number of solutions that are explored and the engine supporting the exploration. The exploration of solutions is composed of three steps: generation, evaluation and selection of solutions.

The generation of solutions impacts the solutions space. It is thus generally limited to a family of platforms (FPGA, MPSoC, processors), or even reduced to a limited type of components inside a family. If an automatic generator of solutions exists (based on heuristics,...), this task is often man-made and restricted to few solutions (less than ten).

Evaluation of solutions is obviously needed to make a further selection. Ideally, an automatic evaluation tool should take into account all the criteria that impact the
1.3. INDUSTRIAL SOLUTIONS

final performances of the system. In practice, it is impossible as the time dedicated to this task would be enormous. Design tools thus decide on few criteria, or even on one sole criterion which is often time related. However, designers often do not dispose of an automatic evaluation tool and base their decisions only on their expertise. In this case, the selection of solutions is often suboptimal and requires iterations.

Automatic evaluation and selection tools allow the designer to make objective decisions but can spend more time than a human decision. Nevertheless, considering the iterations, the automatization offers the best trade-off.

**Simulation**  To compensate the lack in integrated multi-criteria automatic exploration tools, simulators allow designers to evaluate chosen solutions. Simulations are performed on models of the system. These models are more or less accurate depending on the level of abstraction that is considered by the designer. As it is the case for the exploration of solutions, simulation time depends on the model of the system, on the number and type of criteria estimated. All the trade-offs thus appear between the simulation time and the accuracy of the estimated results. Some simulations are also performed for functional verification.

**Prototyping/emulation**  As simulations can take too much time to accurately evaluate performances or verify the functionality of the system, prototyping or emulation can be used before performing the final physical design of the chip. Rapid prototyping is enabled by the use of FPGA that is mostly used for SW development and system analysis. It is rather used for small devices like microprocessors to enable an efficient system analysis and functional verification via signal probing. Debugging capabilities are reduced compared to emulators that are coupled with a complete environment.

An emulator imitates the behaviour of the final system thanks to a dedicated hardware without the need to physically design this system. It is used for debugging and is mostly used to emulate complete SoC systems mixing both HW and SW. Emulators communicate with debugging and verification environment of the workstation running testbenches.

**Levels of abstraction**  Exploration, decision, simulation are all time consuming if the design space is wide and if we want accuracy of estimations and thus optimized solutions. This is clearly the case when we consider the system at a fine grain resolution where there are many degrees of freedom to fix before being able to physically produce a chip. As it is infeasible to explore such huge design space in its all, only a small part of the solutions are considered and simulated what inevitably leads to further costly iterations. To prevent this issue, design flows are by nature composed of several levels of abstraction. Nowadays, extra upper-levels enable to deal with the increasing complexity (e.g. Transaction Level Modeling) and facilitate the faster exploration of more solutions at the price of poor accuracy and reduced informations.

**Code generation/synthesis/refinement**  As design flows are most of the time composed of several levels of abstraction, it implies that the system, or part of the system is represented with different models that will progressively add information
on the final system. The refinement from one level to the next is a complex problem that can be automated. Code generation or synthesis are refinement processes that are typically performed by dedicated tools that include a kind of exploration policy. They base their decisions on few criteria like the execution time or the power consumption.

**Reconfigurability/programmability** A way to reduce development costs and design time is to enable platform reuse. To be reusable, a platform must be adaptable to several targeted applications and offer an efficient solution (in terms of performances) to the design. Typical reconfigurable and reprogrammable architectures are use to this end like FPGA or processors-based platforms.

The tools/methodologies/frameworks/environments found in the literature implement some of these solutions but have limitations. We explain this in Chapter 2 devoted to the state-of-the-art.

Based on this survey, we will show that our multi-criteria prediction tool, NESSIE, is part of the race to improve the design of embedded systems.

### 1.4 An original solution: NESSIE

NESSIE has been developed in a previous work [1] by Alexis Vander Biest, to cope with the limitations of the current tools and to give an original solution to the highly constrained SoC design problem. The ingredients that compose the tool, and that will be detailed in Chapter 3, are given below:

- joint application/architecture exploration based on a mapping core and on user-defined models;
- fast and multi-criteria simulation engine based on user defined-criteria and parameters;
- hierarchical representation of the system (application and platform) at several levels of abstraction;
- flexible design space definition thanks to user-defined degrees of freedom;
- flexible allocation and routing policy via the use of user-defined weights.

Moreover, the purpose of the tool is to be general and flexible enough to target a large set of applications and platforms at different levels of abstraction.

*The aim of this thesis is to validate NESSIE by confronting this original tool to external real design problems.*
1.4. AN ORIGINAL SOLUTION: NESSIE

Our work

The goal of this work is to answer two main questions:

1. does NESSIE work?
2. does NESSIE offer a new solution to the design of embedded systems?

These questions lead up to the following subsequent issues:

- How easy is it to model an application and a platform in NESSIE? (Q1)
- Is the mapping core sufficiently flexible? Does it perform realistic simulations? (Q2, Q3)
- Is the exploration policy efficient? (Q4)
- How fast are the simulations compared to other tools? (Q5)
- How can the tool be integrated into an existing toolchain or coupled with existing tools? (Q6)
- What are the limitations of the tool? (Q7)
- Which modules or further development should be implemented to improve the tool? (Q8)

To address these questions, we have chosen two external case studies that are representative of the current embedded systems problems.

The first case study concerns the design of a typical telecommunication application and a reconfigurable processor, ADRES, and its design flow used at IMEC. This design problem has been studied at a low level of abstraction and constitutes the main part of this work. This is presented in Chapter 4.

The second case study is considered at a higher level of abstraction and is composed of a video decoder application and an MPSoC platform applied on the problem of 3D stacking. The external design flow is available at the National Technical University of Athens (NTUA). This example is presented in Chapter 5.

To validate NESSIE, we have applied the following methodology for each case study:

1. analysis and formalization of the external design flow;
2. study of the modeling capability of NESSIE and modeling of the case study: definition of the levels of abstraction, modeling of the criteria, the platform, the application, definition of the design space (i.e. the degrees of freedom);
3. simulations of the selected set of solutions in both tools (NESSIE and the external decision tool) in order to:
CHAPTER 1. CONTEXT AND MOTIVATION

Figure 1.4: Relative potential position of NESSIE compared to classical design flows

- discuss the multi-criteria estimation capability of our tool (relative trend of the results, accuracy, ...)
- discuss its prediction capability
- analyze its limitations

4. quantification of the design time gain when using NESSIE upstream from the classical flow;

5. quantification of the modeling time;

6. identification of the required future developments;

7. comparison of Nessie with major SoA tools.

As a result of this thesis, we will show where NESSIE stands compared to the design tools used in the case studies in terms of \textit{design time} and \textit{performance prediction}. This is illustrated in the figure 1.4 where the most efficient tools are located in the upper right corner of the graph.

Our validation work aims at giving an answer to this issue and will condition the future development of NESSIE.

In the next chapter, we present the literature survey of existing embedded systems design tools.
Bibliography

Chapter 2

State of the Art

Abstract

In this chapter, we present a global survey of industrial and academic tools and methodologies developed to design electronic embedded systems. After a brief definition of typical concepts, we start with a quick overview of tools that have been widely analyzed in the previous thesis. Then, a deeper overview of others tools complete the survey. A table is made to compare the tools and identify their limitations, based on different criteria (design space exploration, simulations, functional verification, synthesis, levels of abstraction, HW/SW description, mapping policy). Finally, as a result of the discussion, we present the goals of our original tool NESSIE which will be validated through this work.

2.1 Introduction

The ITRS (International Technology Roadmap for Semiconductors) still highlighted in its report 2009[1] on Design that “Cost (of design) is the greatest threat to continuation of the semiconductor roadmap”. Moreover there is still a productivity gap between the transistor integration possibilities and the ability to effectively design them. This is why there is always a need for new tools development and methodologies to deal with the nowadays complexity and constraints (costs, time-to-market, performances,...). As the state-of-the-art will show, the new trend to deal with these design problems is to use more ”software” (or processors) in the SoC. Indeed, by reducing the need of manufacturing new Integrated Circuits (ICs), industries try to reduce the production cost and design time by choosing multi-cores, programmable and/or reconfigurable solutions. However, new design flows and tools are needed to be able to deal with such platforms. We will see that the industrial and academic tools presented here offer some solutions but are still limited and efforts still need to be done to reduce the design time and better explore the design space.
In the previous thesis, the author already highlighted the need to develop tools that reduce time and cost design. His literature survey thoroughly described 19 tools and methodologies developed in the context of electronic design aid. They were classified in five categories regarding their main characteristics: behavioural languages, hw/sw codesign tools, Y-chart related tools, design space exploration tools, UML.

We will not describe all these tools again, but rather cite them and complete the state-of-the-art with other tools, methodologies or projects. Before presenting the literature survey, we define some vocabulary. Indeed, in the literature, several words are often used to define different concepts. To avoid any misunderstanding, we will give our own definition of important vocabulary\(^1\) and spend some time to define important concepts related to the embedded system design.

**software** in the literature, software (SW) refers to the program executed by a processor (typically a C code) or the processor itself in an heterogeneous platform where there are processors communicating with FPGA/ASIC’s. The meaning is generally well understood via the context of the paper. As an extension, SW is also used as the functionality running on the hardware, i.e., what the chip will execute in order to produce outputs for given inputs stimuli.

In this thesis, we will use equally the terms *functionality* and *application* for this last definition. When the context is clear, we will also use *SW*.

**hardware** hardware is used in two contexts in the literature: the material support for the application, or the part of the platform in contrast with the processors (FPGA, ASIC, IC). In this work, we will use the term HW with the first definition, equally with the terms *platform* or *architecture*.

**system** the system is used when referring to a model or a physical representation of the application and the architecture together, i.e. the entire entity to design.

**model-based design/model-driven design** these methodologies take models as the central tool to specify, refine, verify and simulate both the SW and HW parts of the system. Via metamodels\(^2\) and model transformations\(^3\), the goal is to produce design and *correct-by-construction* code, integrate in the same environment both aspects of the system and enable automatic functional verification. The only standard model transformation tool is QVT (Query/View/Transformation), standardized by the OMG\(^4\). In this methodology, we find typical languages like Matlab and more recently the UML.

**system-level design** languages have been developed to capture systems jointly with hardware and software at a high level of abstraction: UML, Esterel, C-based lan-

---

\(^1\)These words have already be defined in the thesis of A. Vander Biest. We have of course kept the same meaning for consistency.

\(^2\)A metamodel adds semantic rules and syntax to models. Models must be conform to their metamodel.

\(^3\)Input and output models of the transformation must be conform to their metamodels respectively. Transformations are based on rules.

\(^4\)http://www.omg.org
languages (SystemC\textsuperscript{5}, SpecC). System-level allows the designer to describe a system solely as a composition of algorithms, without having to consider implementation details. The OMG\textsuperscript{TM} (Object Management Group\textsuperscript{TM}) has defined a UML profile for "Real Time and Embedded Systems" (RTES). This particular extension is called MARTE\textsuperscript{6} for "Modeling and Analysis of Real-time and Embedded" systems. It provides all the support for the specification, design and verification of such systems. The former profile was the UML for "Schedulability, Performance and Time" and has been replaced by MARTE. We will explain more deeply this profile and its use through GASPARD, a tool presented in the state-of-the-art.

2.2 Literature survey

In the previous work, the author had covered a wide state-of-the-art of the tools and languages used in the embedded systems design. We first take some time to present them.

2.2.1 Previous survey

The first state-of-the-art introduced five categories representing the main trends in the literature.

1. Behavioural languages : behind this category, we find tools that are dedicated to the representation and definition of the functionality of the system.
   \textit{Ptolemy II [2]; El Greco (CoCentric System Studio)[3]; Esterel [4] and other synchronous languages-based tools}; \textit{SystemC[5]}

2. HW/SW codesign tools : these tools generally focus on the modeling of the platform heterogeneity and include the specification, the simulation, the synthesis, ... of the entire system. Typically, Co-design HW/SW refers to flows that enable the partition of the application between parts mapped on processors (SW) and parts executed/accelerated by another component (HW) like a FPGA or an ASIC for example.
   \textit{Polis [6]; AADL[7]; Chinook[8]}

3. Y-chart related tools : in the Y-chart model [9], the functionality, the platform and their mapping are explicitly modeled.
   \textit{Mescal [10]; Artemis [11]; Spade [12]; Metropolis [13]}

4. Design space exploration tools : in this category, we find tools that are dedicated to the exploration of solutions or that provide exploration help to allow designers to extract interesting solutions from wide spaces.
   \textit{Milan [14]; the SoC Architecture explorer [15]; Epicure [16]; Rosetta [17]}

\textsuperscript{5}www.systemc.org
\textsuperscript{6}www.omgmar.te.org
5. UML: the tools considered in this category are based on the UML language or implement the related methodology (see model-based design).

*MARTE* [18]; *SysML* [19]; *Koski* [20]

We will not detail these tools as it has been done in the previous work. However, we have summarized in a table the different characteristics of these tools that allow to have a good idea of the available functionalities they offer.

The table 2.1 compares the tools based on several topics they could integrate: the automatic or manual (M) exploration of solutions (AE) on the application (SW) and/or the architecture (HW); the simulation, based on the estimation of one criterion (SO) or multiple criteria (MO); the functional verification (FV); the automatic synthesis and VHDL generation (Synt); multiple levels of abstraction or hierarchical representation (Multi AL); the application and/or architecture description (SW/HW); the automatic mapping with allocation (A) and scheduling (S).

Each tool has its own main strength and it is thus difficult to highlight them from the table. However, we see that no tool gathers all the functionalities. Moreover, the following remarks can be done:

- exploration capability is globally poor, either manual or limited to the hardware
2.2. LITERATURE SURVEY

exploration. Moreover, when existing, the exploration is performed inside the tool with a fixed policy.

- most of the tools perform simulations based on time-related criteria exclusively. The others often target MO estimations based on only 2 or 3 hard-coded criteria, like energy and area. However, the final performance of the solution depends on many criteria.

- if some tools have multiple AL description capabilities, they have a limited hierarchy.

- few tools perform automatic mapping including both allocation and scheduling. Moreover, the policies implemented are not customized to the platform.

- the tools generally target a particular kind of platform or application: control or data dominant applications, multiprocessors/multimedia platforms, processor with reconfigurable components (FPGA) systems...

We complete this survey below and detail some of the tools cited above that are the most used nowadays- by other tools that are representative of the current trend in the embedded system design.

Regarding the design evolution and the tools, we have decided to group them in System-level tools, Integrated tools, Exploration environments, IP reuse environments.

2.2.2 System-level tools

MPARM(2004)

Developed in the University of Bologna, MPARM [21] is a simulator built to accelerate the design and the architecture exploration of MPSoCs. It targets heterogenous multiple processors-based platform because it offers flexibility and parallelism for highly parallel computation applications like those provided in the multimedia field. The support for models and simulations is SystemC. It supports architecture components models like processors (IP’s), SoC busses, memories, the hardware support for parallel programming, operational operating system port and code development tools (via cross compilation tool chain). To deal with these different aspects, SystemC is used as the simulation environment. But to reduce the simulation time, the models themselves are not described in SystemC. ISS, implemented in C/C++, are used and integrated in the same platform thanks to the use of SystemC wrappers that are interfaces and synchronization layers. The use of these wrappers facilitates the addition of other processors and their exploration. The same principle is used for the description and embedding of memories and peripherals. However, co-simulation is also supported between C/C++ models and SystemC ones. The simulation gives results at cycle accurate and signal accurate level. It gives output statistics for the component performances (cycles), the memories/caches accesses (size, latencies, hits, misses,...), the interconnection accesses,... It also produces signal waveforms and memory access tracings. Moreover, power models can be included and frequency/voltage scaling is allowed. The architecture exploration is facilitated thanks to an easy tuning.
CHAPTER 2. STATE OF THE ART

of the parameters via simple code line switches with the use of scripting that enables fast
switch to different simulation runs. As interconnection is of main concern in MPSoC plat-
forms, the environment focuses on the interconnection possibilities and proposes a wide
choice of interconnection topologies to explore (shared bus, bridged configuration, partial
or full crossbars,...).

As explained, the MPARM environment enables the integrations of different ISS easily.
Therefore, it uses typically IP’s provided in the industry. The paper [22] proposes to
extend the capabilities of such an environment with the use of the LISATek tool that
generates ASIP in C/C++ or in VHDL.

PDesigner (2008)

PDesigner\(^7\) is an Eclipse based framework targeting multiprocessor platforms and giving
support for the modeling and the simulation. It works with several plugin/modules like
PBuilder, PDLibrary, IPZip, PArchC. The components can be described with ArchC or
SystemC. Once the architecture has been graphically created by the user by importing
processors, memories, connections, the user load applications on these processors (binary
files). Then, an executable simulator can be automatically generated from this description
by compiling the platform.

To simulate the platform, the executable simulator is run. To explore different platforms,
the user can easily change the types of the components just by right clicking on the
graphical components. However, it is not coupled with an automated tool that performs
the design space exploration. Different levels of abstraction can be chosen : untimed,
timing estimated or cycle-accurate. The module PBuilder allows the user to define the
platform graphically, at several levels of abstraction. The defined components are stored in
the PDLibrary, based on the SPIRIT distribution format to enable the easy sharing among
users. The HW components can be connections, processors, busses, devices, memories,
cache memories.

ARTS(2007)

ARTS [23] is a SystemC-based framework dedicated to the modeling and the simulation
of multiprocessor SoCs based on the HW/SW codesign paradigm (cosimulation of the SW
and HW parts of the system). It typically allows the designer to explore and analyse the
performances of the network connecting the different HW components (depending on traf-
ic, load conditions) and the effects of RTOS choice (scheduling, synchronization, resource
allocation policies). Indeed, the ARTS framework highlights on the real-time simulation
capabilities where different RTOS scheduling policies can be explored and preemptive
capabilities are included. The framework is illustrated in the figure 2.1. We see that
the framework separates the specification of the application properties (execution time,
memory requirements, power usage, deadline constraints,...) and the platform properties
(number and type of PE, RTOS schedulers, interconnection). To enable the simulation,
the user has to provide a model of the application via a graph representation and a model

\(^7\)http://www.cin.ufpe.br/~pdesigner
2.2. LITERATURE SURVEY

![ARTS Framework](image)

Figure 2.1: ARTS Framework

of the platform composed of computing, interconnecting and memory components. To form an RTOS model, the user has to select and interconnect a scheduler, a resource allocator, a synchronizer (provided in the library of the framework) that is added into the platform representation. All these models are described thanks to ARTS scripts. Then the user must also provide the tasks characterization (charac on the figure) which tells the tool which computing component of the platform can execute a given task and for which cost (cycles, memory need,...). Finally the mapping of the application model onto the platform model is performed by the tool via an event-driven model. The simulator generates several outputs like the task execution profile, the bus contention, the memory and energy profiles,... The user investigates the merit of the solution based on this information and iterates to explore alternatives by changing the mapping, the task graph or the architecture of the platform. The exploration of solutions is thus done manually. Cross-layer properties can be analyzed: impact of OS scheduling policies on memory and communication performances, impact of communication topology/protocols on the miss of the application deadline.

ReSP (2006)

ReSP\(^8\) (Reflective Simulation Platform) [24] is a high level open-source simulation platform. It is based on the SystemC language at the TLM level. The platform has been built with the Python language. Python enables non-intrusive reflective\(^9\) capabilities of

\(^8\)http://resp-sim.org

\(^9\)The reflection, in computer science, is a process by which a program (the software) can observe and modify its structure and behaviour. It is commonly used in scripting languages, like Python. The originality is that the modification is done dynamically and can be determined and executed at runtime.
the SystemC component models (enable dynamic modification of the hardware models easily). This is used as a virtual platform for HW/SW codesign. The approach aims at facilitating the integration and interoperability of external IPs and models (processors described in ArchC, memories, interconnections, timers,...). The framework is composed of a debugger, a profiler (to give statistics on the executed software, without instrumentation of the application code), a fault injector (to simulate the system behaviour with faults). Thanks to an automatic wrapper generator, the framework enables the easy integration of any SystemC modules to offer IP reusability. This platform, still young, offers a shared environment to simulate quickly at TLM level and integrate new IP's to extend MPSoC exploration. It is especially focused on timing estimation, but power analysis should be included to the framework.

**Cofluent Studio**

Cofluent Studio\(^\text{10}\) is an industrial toolset based on Eclipse that allows the user to model the system at a high level of abstraction, simulates this system and generates a SystemC description for verification. Models are captured graphically thanks to domain-specific language or UML. By defining the system at a very high level of abstraction, the idea is to enable a fast design space exploration thanks to a simple representation of both the application and the platform. It targets in particular complex MPSoCs. Application and platform are defined separately as illustrated in the figure 2.2. The application can be described graphically and with ANSIC/C++ code. The platform can also be described graphically and composed of different types of components (embedded SW, HW, mixed platform, SoC, FPGA, ASIC, ASSP, embedded systems or not). Variable parameters are coupled with these components to easily change the model properties and enable exploration. The mapping is defined by the user via drag and drop of the application components on the platform components. The tool generates automatically a SystemC description of the system that can be then simulated at the TLM level to analyze timing properties. Moreover, SystemC testbenches can also be generated to perform platform validation.

**Synopsys**

Synopsys\(^\text{11}\) is one of the main companies providing industrial tools for design, verification, simulation, design reuse of SoCs from system-level to silicon-level. That way, it has bought other know companies like CoWare (system-level design and verification solutions as CoWare ConvergenSC) or VaST Systems (provides virtual prototyping solutions as VaST) that developed also design tools. These tools are still used and supported. Here are two tools available in Synopsys.

*Platform Architect*  The *Platform Architect*[25] tool, illustrated in the figure 2.3 is dedicated to capture the architecture graphically, perform exploration and analysis at the

---

\(^{10}\)http://www.cofluentdesign.com

\(^{11}\)www.synopsys.com
2.2. LITERATURE SURVEY

TLM level, thanks to a graphical SystemC based environment. This is called "Electronic System Virtualization design" (ESV). Real software applications can be executed on the architecture components via the framework. It is coupled with the Model Designer Library and the Synopsys system-level model libraries that enable the capture of complex IPs and their verification at transaction level. The user drag-and-drop SystemC TLM components from different libraries (Model Designer, user-defined, third-party IP development tools). This interface integrates simulation facilities like the simulation building, running and the system-level analysis. The analysis at the transaction level of the platform is performed via the SystemC Explorer that provides a graphical debugging environment where the user can attach monitors to the components one wants to observe (cycle-accurate performance analysis). The framework enables both architecture (cycle-accurate performances, throughput, bottlenecks, bus switching, cache usage, busses and memory architecture) and functional (system response, task scheduling, processor loading to drive partitioning, software profiling) analysis.

Innovator(2008) Innovator is also a SystemC based environment developed to create and validate virtual platforms via the integration, analysis and verification of transaction-level models. A validated virtual platform can then be delivered to programmers as a stand-alone, run-time executable. Transaction models can be integrated from various sources: DesignWare System-Level library, SystemC models, custom C/C++ models.

2.2.3 Integrated tools

In this category, we target tools that are integrated in a more complete flow covering several design steps.
DAEDALUS FRAMEWORK

The open-source Daedalus framework [26], shown in the figure 2.4, targets the system-level design space exploration, synthesis, application mapping and prototyping of MPSoC platforms for multimedia applications. This framework proposes an integrated tool flow from system-level to RTL for FPGA implementation. The framework is close to the KOSKI design flow, presented in the previous survey, and the SystemC methodology. The framework is based on three tools: SESAME, ESPAM and KPNgen. The MPSoC platforms are defined with pre-verified and pre-defined IP via a library of components (programmable/dedicated processors, memories, interconnections, ...). The application is given as a C sequential algorithm composed of nested-loops. The KPNgen tool converts this sequential description into parallel Kahn Process Network (KPN) [27]. It targets typical multimedia applications where the specification is based on so-called "static affine nested loop programs". The SESAME environment uses the KPN and components models from a library to perform the mapping of the application model onto the architecture model that are defined separately. Different scheduling policies can be applied. The tool performs co-simulation of both the application and the architecture via trace-driven simulations and estimates the performances of the system based on timing parameters like latencies, memory accesses, communication transactions. Moreover, statistical informations are also estimated by SESAME to guide the designer during the system exploration: resource usage, contentions in the system, average bandwidth, critical path analysis, ...

12 Affine refers to the loop iterators and iteration constraints, static refers to the control in the algorithm
Then, the ESPAM tool takes the chosen solution described in XML files (platform specification, mapping specification, KPN description) and generates VHDL thanks to the corresponding RTL IP components of the architecture. It also generates C code for the application parts mapped on processors. The framework contains the Oracle Berkeley DB XML relational database management system to store all the related informations (parameters, models, results). Daedalus has been built as a module-based design flow in order to enable the interface with other tools, add design steps or customize the flow for specific domains. The framework implements also control and monitoring utilities to facilitate performance measurements, models calibration, design space exploration at implementation level.

**SynDEx**

SynDEx\(^{13}\) (Synchronized Distributed Executive)\[^{28}\] has been developed to perform fast prototyping, optimization and implementation of distributed real-time embedded applications and targeting multicomponent architectures for highly parallel and real-time applications (multimedia, transport, telecommunication). It is based on the automatic code generation of the system thanks to formal verification and manual or automatic (via the use of heuristics) exploration of different implementations. The core of the tool has been written in *Objective Caml* and the graphical interface in TCL/TK. SynDEx has been built as a support for the *AAA methodology* which proposes a codesign approach of the system (processors vs specific integrated circuits). The specification is done via directed acyclic graphs where edges represent the inter-operation dependence. The formal verification is possible thanks to an interface with synchronous languages (Esterel, SyncCharts, Signal). The tool is also interfaced with other languages like UML2.0 (MARTE profile), AIL.

---

\(^{13}\)[http://www.syndex.org]
(automobile design language), CamlFlow (a functional data-flow language)... The tool provides functional and timing simulation via the use of (multi-)workstations. Moreover, the Integrated Circuit SynDEx tool version enables the implementation of a task onto a dedicated integrated circuit that could be used in the multicomponent platform as a non-programmable component (ASIC and FPGA). This SynDEx-IC automatically generates synthesizable VHDL and is interfaced with the former one. It provides also visualisation of the timing characteristics of the system. The real-time executives are built as RTlinux, Osek, and dedicated executive for processors (PIC18F2680, DSP, microcontrollers,...).

AAA methodology Standing for Algorithm-Architecture Adequation, this methodology has been proposed by Y. Sorel (INRIA) in 1994 in the paper [29]. It is based on graph models that exhibit the potential parallelism of the algorithm and parallelism offered by the architecture. Timing information is given by the user (WCET, operations and data transfer periods, required memory,...). Then the implementation is performed via heuristics, taking into account the timing constraints, by distributing and scheduling the algorithm graph onto the architecture graph. The optimization is done with respect to the execution time of the system and the resource allocation (processor or IC). Graph transformations are used to perform this implementation and enable formal verification. The result is a so-called Synchronized Distribute Executive for the application, which is automatically obtained from a library composed of the executive kernels, i.e. the architecture components. Synchronized indicates that the implementation takes into account inter-component communications by generating synchronization semaphores, avoiding deadlocks.

SPICES SPICES\textsuperscript{14} stands for Support for Predictable Integration of mission Critical Embedded Systems. This project aims at the design, verification and implementation of avionics electronic systems (could be extended also to automotive, telecommunication problems that are also critical). As a consequence, it deals with critical real-time systems. This project proposes a methodology and integrated tool suite based on Model-driven Engineering. The work is based on the AADL framework[7]. It targets general-purpose processors and reconfigurable hardware. The goal is to provide formal methods, automatic code generation and link to SystemC description in order to fill the gap between high level models and physical prototypes and enables to formally verify functional and non-functional behavior of the critical system at early stages. The tool suite is coupled to the Eclipse environment in the TOPCASED framework[30] (The Open-Source Toolkit for Critical Systems, still under development) based on plugins to integrate the different tools. It relies on meta-modeling to manage any kind of models.
One of the goal is to enhance the AADL language to make it standard for the design, verification, implementation of such embedded electronic systems.

\textsuperscript{14}http://www.spices-itea.org
GASPARD

GASPARD\cite{31} stands for Graphical Array Specification for PARallel and Distributed computing\textsuperscript{15}. It has been created in the DaRT team-project\textsuperscript{16} and with the ”Laboratoire d’Informatique Fondamentale de Lille” (LIFL) to provide a prototype of environment and develop methodologies for the Model-Driven Design based on the MARTE profile defined by the OMG. It especially focuses on real-time MP-SoC platforms for parallel intensive applications like those found in multimedia. In this environment, both modeling, simulation, analysis, verification, code generation for SoC are integrated. The figure 2.6 represents the framework. The application and the architecture are defined separately via the PAPYRUS\textsuperscript{17} editing environment thanks to UML class diagrams. The association (allocation of tasks on components) is also specified by the user via this environment. Once these three aspects are modeled, several models transformations can be performed to generate different representation of the system depending on the target: synchronous languages (for validation), OpenMP (execution in C or Fortran), SystemC (TLM simulations), VHDL (synthesis for FPGA). The deployment step is a kind of link editor that links the components specified in the first step with the IP library, at the level of abstraction the user desires (depending on the target). More specifically, GASPARD uses a subset of the MARTE profile defined by the OMG to deal with repetitive applications/architectures. The figure 2.7 gives the packages included in the original profile.

- Design model: modeling of real-time embedded systems concepts
- Analysis model: annotation of application models for properties analysis

\textsuperscript{15}www.gaspard2.org
\textsuperscript{16}http://www.lifl.fr/DaRT/
\textsuperscript{17}http://www.papyrusuml.org
• Foundation: the two previous packages share concepts grouped in the Foundation package. Non-functional properties (NFP), timing notions (Time), resource modeling (GRM), components modeling (GCM), allocation modeling (Alloc).

• Repetitive Structure Modeling (RSM): this annexed package can be used in addition for the embedded and real-time application and also contains predefined model libraries. It allows to express in a compact way repetitive patterns both for the architecture or the application (tasks parallelism).

The modeling in GASPARD includes the RSM, the Alloc, the HRM (Hardware Resource Modeling) and SRM (Software Resource Modeling) profiles. For example, the loops in an application are represented into a factorization way to condense the description. The refactoring shown on the figure expresses the fact that loop transformations can be applied on the loop retroactively to adapt the application to the architecture. If this step is not automatic, it is facilitated by a dedicated toolbox.

Once the systems have been generated in the desired language, simulations can be performed (TLM, RTL,...)

CatapultC (Mentor Graphics)

CatapultC\[32\] is a tool provided by Mentor Graphics and dedicated to the automatic synthesis from ANSI C++ code or SystemC at high level to VHDL at the RTL level and targets FPGA or ASIC’s. This tool is coupled with automatic functional verification. Simulations can be performed at several levels of abstraction (functional, TLM, RTL, gate level). The RTL is optimized for performances, area and power.

Bluespec

Bluespec\[33\] provides industrial solutions for the high-level synthesis and emulation targeting ASIC’s and FPGA. It focuses on the architecture exploration and refinement. Based on high-level models, the Bluespec Compiler synthesizes it in Verilog RTL or in SystemC TLM. Simulations can be performed on the architecture at both level of abstraction. The tool does not take into account application aspects, but enables the analysis of different criteria like the area, the power, the performance and the latency of the platform. The high-level models are described in the Bluespec SystemVerilog language to define the architecture. This language enables very abstract source description based on scalable atomic transactions and parametrization of the hardware IP.

Cynthesizer (FORTE Design Systems)

The Forte company\[18\] provides environment for automating the generation of RTL code from high-level description (SystemC). It thus provides automatic synthesis from high level to RTL. Cynthesizer\[19\] uses a SystemC-TLM description of the system as input and

\[18\]http://www.forteds.com
2.2. LITERATURE survey

Figure 2.6: GASPARD methodology
Figure 2.7: MARTE UML profile

generates automatically RTL to enable the GDSII path. Simulations can be performed at the TLM level. The tool is interfaced with IPs to accelerate the architecture exploration.

An hardware-dependent software design methodology (2009)

The environment[34] presented in the figure 2.8 is dedicated to the design of complex and highly constrained MPSoC’s. In such platform, we talk about hardware-dependent software as the performances of the system are tightly coupled with the way the application (=software) is defined for a given processor (=hardware). In the environment, the application is developed independently from the architecture (platform-agnostic specification) at the TLM level. The model is a hierarchical graph composed of processes containing sequential C code. The processes communicate via abstract channels. The targeted architecture is specified by the designer separately. The user defines also the allocation of the software/hardware parts, the mapping of the processes on the components (with the tasks priorities and the selection of the scheduling policy for each processor), the communication topology and its parameters. The system compiler then automatically maps the application onto the platform and generates the application code and a system model that can be used for virtual prototyping. The generation can be done at different levels of abstraction.
2.2.4 Exploration tools

MULTICUBE

The MULTICUBE \((MULTI)^3\)[35] project\(^{20}\) has started in 2008 and is dedicated to the multi-objective design space exploration of multi-processors SoC architectures for embedded multimedia applications. It targets MPSoCs based on Network on chip architectures. The goal of this project is to create a framework that enables an automatic design space exploration to tune the SoC architecture for a given application, via a set of different metrics like the energy, the latency, the throughput, the bandwidth, the QoS,... It implies the definition of several heuristic optimization algorithms to reduce the exploration time and enable a quick selection of the best solutions based on a Pareto curve. The idea is also to give run-time information on the allocation and scheduling of the tasks on the multi-component platform. The exploration is performed at system level through the use of the SystemC language. The framework targets open-source and proprietary tools and is strongly industry-driven (STM, DS2). Two main tools compose the exploration framework: the M3SCoPE simulation tool and the M3Explorer exploration tool.

\(^{20}\)http://www.multicube.eu
CHAPTER 2. STATE OF THE ART

Figure 2.9: M3CoPE framework

**M3Explorer**  M3Explorer is an open-source exploration tool built as a retargetable tool for any configurable platform if the user defines the design space in the appropriate XML file. As shown in the figure 2.10, the tool is driven with command-line and scripts and returns a pareto curve of the configurations for the given design space defined in the input XML file (parameters, metrics, solutions generation rules). This structure enables the construction of other automated exploration strategies. The kernel exchange information with the use case simulator to run configuration and use resulting metrics to guide the optimization and design of experiment (full search, random, two level full factorial, scrambled,...) modules. It can be coupled with plug-ins to perform different optimization or multi-criteria decision (Pareto DoE, adaptive windows pareto random search, MO simulated annealing, MO particle swarm optimizer, non-dominated sorting genetic algorithm, simple evolutionary MO optimize). It also supports response surface models (linear regression, spline, radial basis functions, shepard, neural network) to represent the solutions. This technique is used to determine an analytical dependence between several design parameters and response variables. If this tool offers different exploration strategies, it focuses especially on architectures aspects and does not consider the functionality of the systems.

2.2.5 IP reuse environments

**SoCLib**

SoCLib\(^{21}\) is an open-source platform providing a library of SystemC simulation models for virtual prototyping of multi-processor system-on-chip. The IP cores are available at two levels of abstraction: CABA (Cycle Accurate/Bit accurate) and TLM-DT (Transaction Modelling with Distributed Time). This platform can be associated with third party tools to perform simulations, synthesis,...

\(^{21}\)http://www.soclib.fr
2.3. TOOLS COMPARISON

IP-Xact

IP-Xact\cite{IP-Xact} is a design environment dedicated to the sharing of standardized IPs, targeting new applications, dealing with TLM and advanced verification methodologies. The SPRINT project\cite{Sprint} (Open SoC Design Platform for Reuse and Integration of IPs) is the foundation of this open-source platform. IP-Xact has been defined by the SPIRIT consortium as a standard XML format for automated IP configuration and tools integration amongst multi-vendor flows. It is driven by several leading semiconductor and EDA companies such as Texas Instruments, ST, LSI, Freescale, NXP, Infineon, Cisco, Cadence, Synopsys and Mentor Graphics. Platforms like the Magillem platform assembly uses the ip-xact standard and offers an integrated design environment to guide designers during platform assembly and configuration by enabling automated design integration within multivendor tool flows.

2.3 Tools comparison

We have compared these additional tools based on the same criteria used in the section 2.2.1 (automatic exploration capability, the simulations, the functional verification, the synthesis, the number of levels of abstraction, the SW and HW description, the mapping policy). These are presented in the table 2.2. We discuss this table for each criterion below.

\footnote{http://www.ecsi.org/sprint}
CHAPTER 2. STATE OF THE ART

<table>
<thead>
<tr>
<th>Tools</th>
<th>AE SW</th>
<th>Sim SO</th>
<th>FV</th>
<th>Synt</th>
<th>Multi AL</th>
<th>SW</th>
<th>HW</th>
<th>Mapping A S</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPARM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PDesigner</td>
<td>M</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARTS</td>
<td>M</td>
<td>M</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A A</td>
</tr>
<tr>
<td>ResP</td>
<td>M</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CoFluent Studio</td>
<td>M</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform Architect</td>
<td>M</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Innovator</td>
<td>M</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Daedalus</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A A</td>
</tr>
<tr>
<td>SynDEx</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GASPARD</td>
<td>M</td>
<td>M</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CatapultC</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BlueSpec</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synthesizer</td>
<td>M</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HW-dep SW framework</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multicube</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Comparison of the 15 additional design tools based on eight criteria (automatic exploration, multi-criteria simulation, functional verification, synthesis, multiple AL, SW and HW description and mapping policy)

2.3.1 Space Exploration

Design space exploration is of main importance when talking about embedded systems design. As they are highly constrained, they need to be optimized for multiple, conflicting criteria and they concern very complex systems offering a wide range of solutions that are impossible to explore efficiently without an automatic tool. However, if many tools claim they provide exploration possibilities, in fact, they rather facilitate the exploration of solutions but this still has to be done manually. Moreover, the support for exploration is most of the time enabled for the architecture of the system and not for the application.

2.3.2 Simulations

As we explained in the context of this thesis, simulations are the only way to estimate the performances of the system at a given level of abstraction. Performing simulations early takes less time but are less accurate than those done at lower level of abstraction. In addition, to be able to evaluate the quality of a solution, simulations must be done on all the criteria constrained by the specifications. In practice, we see that a lot of tools just estimate one criterion, generally time-related. Moreover, when the simulators enable the estimation of several criteria, these are often hard-coded and limited to the energy consumption of the platform.
2.3. **TOOLS COMPARISON**

2.3.3 **Functional verification**

Functional verification is of main interest when refining the system and going from one description of the application in a given language to a more detailed description of this application in another language. Indeed, to guarantee that the functionality is conserved (the outputs remains unchanged for the same inputs stimuli), tests or simulations have to be run. To avoid manual verification, the *functional verification* can be integrated in the flow and guarantee thanks to automatic models transformations that the functionality is conserved. In the table we see that this methodology is used for some of the tools working in several levels of abstraction but not for all.

2.3.4 **Synthesis**

Synthesis is used in tools dealing with several levels of abstraction and concerns thus most of the tools located in the second part of the table. This synthesis capability is of importance to automatically generate a lower description of the system (typically in VHDL) which would require a lot of time and efforts for the designer, from a high level of description (like SystemC).

2.3.5 **Levels of abstraction**

This criterion covers the capability of the tool to provide support for a hierarchical representation of the system (HW/SW) and the automatic refinement of the system from one level to another. Amongst the tools, several offers the possibility of describing the system on several Abstraction Layers. However, this is often limited to one part of the system, the application or the architecture. Moreover, the refinement is not always automatic and modeling the application or the architecture at a given level requires to describe it manually or to take existing IPs.

2.3.6 **SW and HW description**

In this column, we consider the explicit description of the application (SW) and the architecture (HW). Regarding the survey, we have seen that several languages are used amongst the tools.

2.3.7 **Mapping policy**

Finally, we have chosen to compare the tools based on the mapping policy implemented in their core. Indeed, we have pointed out that the final performances of a system depend on both its application and architecture. But of course, it is directly linked to the way the architecture will execute the tasks composing the application. Depending on the AL, the considered components will be more or less fine or coarse. The mapping policy will then define the way tasks will be allocated and scheduled on the components. If the mapping could be quite obvious for coarse grained architecture, an efficient allocation and scheduling will need an automatic and adapted policy to enable realistic behaviour and to
be cost effective. In the table, we see that most of the tools do not perform an automatic allocation and scheduling. Moreover, the allocation is often performed by the user manually, even if there is an automatic scheduling. Finally, the policies implemented are most of the time hard-coded and adapted to specific problems.

To underline the trend regarding the languages used in tools nowadays, we have also summarized them in the table 2.3. The main languages that have been considered are: UML, SystemC, C/C++, VHDL/Verilog. The column Others is used when other languages are used in the tools. We refer to language used as input models but also to languages automatically generated by the tools as outputs. We clearly see that the trend is to use system level languages, fast simulation and modeling of the complex systems. Moreover, we also see that only few tools are linked directly to lower levels of abstraction that enable the physical description of the embedded systems via the VHDL at the RTL.

Regarding the state-of-the-art presented in this chapter, the development of an integrated and a more general tool makes sense. Our tool, NESSIE was developed to cope with the limitations noticed here.

The goal of this thesis is to see if NESSIE is competitive regarding this state-of-the-art through its validation on real case studies. Of course, we do not claim to propose the ideal solution of the embedded SoC design problem. Rather, as one of the conclusions, we will identify where our tool stands compared to existing design flows/framework. For that matter, we will take as a reference most significant tools presented here, i.e. those which correspond the most to the methodology we propose in NESSIE: Daedalus, GASPARD, SYNDEX. Indeed, they both aim at describing the application and the platform separately.
2.4 Conclusion

In this chapter, we have presented 15 tools that noticeably complete the literature survey presented in the previous work. Through this survey, we have shown that two years later existing tools are not yet able to effectively (i.e. efficient exploration of solutions, minimal design time,...) deal with present embedded system design, although they propose solutions that enable to find satisfying solutions. We have seen, notably, that a lot of the tools, projects or methodologies target MPSoC systems which offer a good trade-off to reduce development time by using programmable components and reuse IP’s.

The prediction tool, NESSIE, that has been developed during the previous thesis, is considered original compared to the existing solutions and aims at reducing the design time by exploring better (unified and flexible representation of both the application and the platform) and faster the design space.

In Chapter 4 and 5, which are the core of this work, we confront our tool with two real case studies, constituted of external design flows, and targeting current design problems based on MPSoC platforms. Based on these concrete examples, we will put NESSIE to the test in order to validate the proposed framework and to determine if the tool is useful and still relevant in the present design context.

As an introduction to this study, the next chapter is dedicated to a deeper presentation of our original tool NESSIE.

Bibliography


CHAPTER 2. STATE OF THE ART


CHAPTER 2. STATE OF THE ART


Chapter 3
True Codesign and Nessie

Abstract

In the third chapter, we present our original performance prediction tool NESSIE and the models evaluation engine YETi. We first explain why we have developed our own evaluation tool and detail the structure of YETi, its hierarchy and main characteristics. Then, we deeper present NESSIE, its methodology, the different components of the tool and the policies implemented. We follow this description by an explanation of the structure of the input and output files used by NESSIE. We conclude the chapter by some questions that emerge from the presentation of these engines and that justify the validation work presented in the next chapters, which are the core of this thesis.

3.1 Introduction

The state-of-the-art has shown that some existing methodologies and tools are close to the philosophy introduced in Chapter 1. This is the case of Gaspard or SynDEx. However, what is proposed in Nessie is a bit different and we believe that our tool can be used as a Very High-Level Tool above existing tool chain or as a complementary tool to give designers extra information that could guide them during the decision phases. In both cases, we present Nessie as a performances prediction tool able to explore automatically and quickly several solutions, based on multiple, user-defined criteria. As introduced in the first chapter, thus the main features of our framework are the following:

- joint application/architecture exploration ("true codesign") based on a mapping core and on user-defined models;
- fast and multi-criteria simulation engine based on user defined-criteria and parameters;
- hierarchical representation of the system, targeting in a generic way both the application and the platform at several levels of abstraction;
• flexible design space definition thanks to user-defined degrees of freedom;
• flexible allocation and routing policy via the use of user-defined weights.

To support these features, the framework has been divided into two engines:

**YETi**\(^3\) which stands for "Yet anothEr Tool for the representation of analytical relations", is an engine that has been built to support flexible models evaluation and easy models integration and composition. In addition, input parameters and models sensitivity analysis have been included in the tool to offer users easy exploration and comparison capabilities.

**NESSIE** Besides this calculus engine, Nessie supports the exploration engine based on a hierarchical mapping core.

In the section 3.2 and 3.3, we respectively present deeper these two tools.

## 3.2 The model evaluator: Yeti

The core of a decision tool or simulator is the evaluation of models. Models can be represented in different ways: analytical expressions, table-based relations, iterative methods, algorithms, lookup tables, ... The most complete and interesting tool that was identified in the previous work was **GTX** (standing for MARCO GSRC Technology Extrapolation system) developed in 1999. This framework was developed to enable models integration in a same environment and allow users to compare and execute them more easily than previous tools. The main interesting features of GTX were:

• separation of the model specification and their implementation and execution. This allows users to easily enter, load and save models from libraries.

• the ability to construct extended and complex models by chaining rules that can be closed-formed expressions, lookup tables, if-then-else structures, external executable rules, code rules.

• multi-platform graphical user interface for the definition of parameters, rules, rules chains and the plotting of the results.

• input values can be specified with single value or sweep simulation.

• output values can be constrained by minimum and maximum bounds. Solutions are automatically sorted based on these constraints.

Although the tool offered a lot of advantages compared to other ones, some important limitations were noticed:

• during the models definition, users have to fix an output parameter, other parameters becoming the inputs. Turning some inputs in output is not allowed. However, for some design, it could be interesting to change the parameter that has to be evaluated (e.g. the technology instead of the frequency).
3.2. THE MODEL EVALUATOR: YETI

• input parameter sensitivity can only be performed by changing the value around the input nominal value and running several times the same rule chain. This solution spends lot of exploration time and could be improved.

• the grammar is too permissive and does not guarantee parameter name uniqueness.

• nothing has been implemented to support easy models sensitivity analysis.

YETi has been developed to overcome these limitations and facilitates the models evaluation and comparison. To support model flexibility, YETi has been built as a three level hierarchical model description framework. Fast extrema evaluation has also been enabled. Moreover, thanks to the use of XML files to deal with inputs and outputs, a strict grammar has been included. The C++ language has been used to implement this framework because of object-oriented capabilities it offers that keeps the YETi structure intact.

A three-level hierarchical model description The methodology used to build the framework of the evaluation tool is shown in the figure 3.1. Each level, from the first to the third one, composes the next level. They are interfaced together thanks to the parameters that are used in the models. These parameters are defined by a unique name, a floating point value and constraints (the lower and higher bound values for the parameter).

Generic rules They give YETi the information on the way one output parameter has to be evaluated based on one or several input parameters. Currently, the user can choose amongst two kind of rules : analytical rules and table rules. An example of analytical expression is given by the equation 3.1.

\[ P_{\text{dyn}} = C_{\text{switch}} \cdot f_{\text{clock}} \cdot V_{dd}^2 \]  

(3.1)

Analytical expressions are represented into YETi by means of trees (as shown in the figure 3.2) where the nodes are the parameters of the expression and the output oriented edges represent the operations. The transformation is done with the shunting
yard algorithm. The evaluation is performed with a depth - first algorithm. This representation enables to deal with parameters constraints. The user can specify the lower and upper bound values of the input parameters and YETi propagates these constraints along the tree to evaluate directly the corresponding output values. The only constraint for the user is to give YETi expressions where parameters appear only once to allow the reversal of the expression.

Tables can be used when only few points are available and when a closed-formed model can not be defined. YETi allows the user to represent multi dimensional structures thanks to an original flexible algorithm developed in the previous work.

The UML diagram of the class representing the Generic rules is shown in the figure 3.3. The structure of the classes provides rules extension. Indeed, analytical rules and tables derive from the generic rule class. We see that currently seven types of operations inherit from the analytical element class and are managed by the tool when using analytical expressions (exponential, division, multiplication, plus, minus, logarithm, inverse). But, if necessary, other operations could easily be added by the user thanks to this structure.

Relations They give the links between all the parameters that will compose a generic rule. For the equation 3.1, the relation is given by 3.2 for example. One relation has several possible generic rules, depending on the type and on the chosen output parameter. This level of representation is the core of the model reversibility in YETi.

\[ R_{\text{dynPow}}(P_{\text{dyn}}, C_{\text{switch}}, f_{\text{clock}}, V_{\text{dd}}) \] (3.2)

The UML diagram of the class Relations is represented in the figure 3.4. We see that a one-to-n composition rule links this class with the genericRule class.

Behaviours They are composed of several relations and enable the representation of complex models thanks to the combination of different relations. Contrarily to relations, behaviours can have several outputs. It is thus necessary to indicate to Yeti the outputs of the behaviour the user wants to evaluate. An example is shown in figure
### 3.2. THE MODEL EVALUATOR: YETI

**Figure 3.3:** UML class diagram of the generic rule in the YETi engine

**Figure 3.4:** UML diagram of the relation class in the YETi engine
3.5 where three other relations have been added to the relation given by the equation 3.1. In this figure, we see also two different orientations chosen by the user depending on the parameters YETi must evaluate ($P_{tot}$ and $IPC$ in orientation 1 or $V_{dd}$ and $FU$ in orientation 2).

The UML diagram of this class is represented in the figure 3.6. We see that a one-to-n composition rule links this class with the $relations$ class.

**XML grammar** The big picture of the framework is schematically represented in the figure 3.7. It shows how the C++ core is interfaced with the inputs (behaviour, input values, behaviour orientations) and outputs (value/constraints simulation, results) that the user will directly manipulate. Alexis Vander Biest has used XML (eXtended Markup Language). It offers standard interfaces defined for object-oriented languages. Thanks to external validating parsers, the grammar of the defined structure can be checked independently of the C++ framework. Moreover, .xsd schemas have been used to defined
the structure of the .xml files. The schemas define admitted values and attributes for the associated file what enables a first automatic grammar check during the file completion. The myBehaviour.xml file contains the entire hierarchical structure composing a model in YETi. A typical behaviour file is separated in two sections: a relationList and a orientationList.

- the relationList contains all the relations that will compose the behaviour. For each relation, the user must specify the list of the parameters that will be used in the relation, associate a parameter with this relation (the left member of the relation) and the rule that linked the parameters together (analytical rule or table rule).

- the orientationList gives all the possible orientations that will define the behaviours. In each orientation tag, the user will thus compose a behaviour with the relations defined before. First, he gives the inputs and outputs parameters of the current behaviour. Second, for each relation, he must give the orientation of the relation for the behaviour, i.e. the oriented parameter of the relation. This information tells YETi if the relation has to be reversed based on the rule and the association given in the first section of the file.

To conclude this section, it is interesting to specify that YETi can be used as a stand-alone tool. In that case, the user has to give the input parameters via the use of a script. However, in this work, it is used as a kind of models library and calculator called by the main engine of our methodology, Nessie.

3.3 Our simulation environment: Nessie

The core of our prediction tool is NESSIE, which performs the mapping of applications onto architectures at user-defined levels of abstraction. Therefore, the framework is based on a hierarchical description of these parts of the system to enable more or less accurate
exploration of the SoC that has to be designed. Again, the C++ language has been chosen to implement the framework which is interfaced with XML files for the same reasons given in the previous section. The big picture of Nessie is given in the figure 3.8. The C++ core is schematically represented by the grey rectangle. It is interfaced with four types of input files (routingWeight, allocationWeight, simulation, behaviours) and three types of output files (nessieCriteriaResults, timeLine, activityReport). Below we explain how the mapping policy has been implemented to allow high flexibility for the exploration. Then we will present how the user interacts with the tool via the input and output files shown on the figure.

3.3.1 The application

Before understanding how an application is mapped on an architecture in the tool, we have to understand how these parts can be defined. Several Model of Computation exist in the literature and are used in traditional design tools. The choice of a particular MoC depends on the information one wants to extract from an application. As Nessie is not dedicated to target one sole kind of application, the framework has been built in such a way that users could add new MoC if it is really needed by the design. Moreover, different MoC could be used for the different level of abstraction as long as it respects the consistency constraints of the input/output data size. However, currently, one sole MoC has been implemented: it refers to Petri Networks. Indeed, Petri nets have several properties that enable to represent a functionality: sequentiality, parallelism, data dependency, control dependency. The figure 3.9 shows the components of a Petri net.

- **Places**: the places represent the tasks/functions composing a functionality at a given level of abstraction. Places store produced tokens.

- **Tokens**: token are generated in places when these have been executed. On the contrary, it is removed from the places, when a transition has consumed them.
3.3. OUR SIMULATION ENVIRONMENT: NESSIE

Figure 3.9: Illustration of a Petri network and its components

- **Transitions**: the transitions control the execution of the places and enable the description of the sequentiality or parallelism between tasks. A transition is fired when all the required token have been generated in the input places.

- **Edges**: there is two kind of edges. The incoming edges (from places to transitions) gives the number of token required to fire the transition. The outcoming edges give the number of time the place will be executed after the transition has been fired. Together with the transitions, edges define the tasks inter-dependence.

For example, on the figure, the two inputs places have stored respectively 3 and 2 tokens. The transition can be fired because the edges conditions are valid (respectively with the weights 3 and 1). After the firing, the consumed token are removed from the input places. It remains just 1 token in the second input place. The output place will be executed twice as indicated on the outcoming edge (weight=2) and store 2 tokens after each execution.

**Special features**

To ensure realistic representation of functionalities and to guarantee determinism of their behaviour, some special features and constraints have been added in the standard Petri network representation.

**Primitives** places must represent realistic tasks, functions, operations,... As such, *DataIn* and *DataOut* attributes have been associated with the places definition and represent the data size that a functional primitive consumes/produces. These attributes are defined by the user.

In addition, an application can be characterized by parameters that could be required to calculate the criteria of the system. This is why parameters tags are associated to the Petri Network places constituting the primitives of the structure at the considered level of abstraction.
Starting transition

To define a Petri network, the user must define an initial state and thus give the first places that will begin the Petri network. These places will constitute the outputs of a so-called startingTransition which has the particularity that it has no input places.

Determinism

Petri Nets are asynchronous. Indeed, contrarily to synchronous languages, there is no method to order the firing of concurrent transitions. If several transitions are enabled, the output places will be executed one by one. Moreover, there can be a conflict if several transitions have the same input place and are both able to consume a same token. The order of firing will determine the way the Petri Net will be executed. To avoid this non-determinism, the building of Petri Nets is restricted to places with one single output edge. However, to enable the sending of a same token to several places, dummy nodes have been created.

Dummy nodes

Dummy nodes are not associated with a specific operation, and are thus not associated with platform blocks. They do not add any cost (in terms of criteria). When Nessie meets this kind of node, the token at the input is instantaneously transmitted at the output of the place. They can be used to transmit a common data to several transitions. This is illustrated in the figure 3.10 for the successive equations given by 3.3. In this example, the parameter $a$ generated by the place $P3$ corresponding to the operation $+$ of the first equation is used twice, in two different operations. Two dummy nodes have been used to distribute this parameter at two places.

$$a = b + c;$$
$$d = 2*a + g;$$
$$e = 3*a + f;$$

(3.3)

3.3.2 The architecture

As the application/functionality is part of the specification and can thus be viewed as the demand, the architecture acts as the offer that must be able to execute this application at a certain cost (including performances, design time, design and production cost,...). To characterize these features in our methodology, the representation of the architecture must be composed of compatibility lists, indicating which task each component can execute, and be associated to costs for each state to which the component can belong. Moreover, as Nessie is not dedicated to one sole type of platform/component, the framework must support the representation of any kind of architecture. Three elements have been defined in the framework to model the architecture of an embedded system.

Core

The core of a component constitutes its main part and defines each primitive at a given level of abstraction. Looking at real components, a primitive can typically be able to memorize or transmit data, but of course execute operations or do nothing. In Nessie,
3.3. OUR SIMULATION ENVIRONMENT: NESSIE

several states, that are not mutually exclusive, have been implemented to deal with such behaviours.

- **idle**: when a block is not used for transmission, memorization or computation and is ready and waiting for an action, it is typically in idle state.

- **sleeping**: the sleeping mode exists in some platforms and is used when the block is switched off and does not consume power. It implies a wake up time.

- **transmitting**: the transmitting capability is included in a HW primitive when it is able to send a data from one block to another.

- **memorizing**: when a block is able to memorize data, it is associated with a memorizing state.

- **computing**: a block may have several computing states, each time it is compatible with a functional primitive. In this state, the block executes the specified functional primitive.

As we explain below, each state must be associated with a *behaviour*, as defined in the section 3.2 in which the user defines the costs related to the considered component state.

**Ports**  Ports have been included in the component definition to deal with the sending and receiving of data. They represent the input/output interface of the core. As we will explain in the criteria definition, these ports are associated with *latency* and *bandwidth* to allow typically the modeling of buffers for example. As it is the case for the core of a component, ports are coupled with states illustrating different possible behaviours.

- **sending**: this state refers to the emission of a data token.

![Diagram of NESSIE's Petri network showing use of Dummy nodes](image-url)
• receiving : this state refers to the reception of a data token.
• inactive : a port is inactive when it does not send or receive a data token.

Links Finally, to describe the structure of the architecture, logical links are used. They connect cores together through the ports. No cost is associated with these entities contrarily to transmitting blocks. Links can be bidirectional or unidirectional.

Criteria Obviously, the core and ports states will condition the criteria evaluation as it represents different behaviours of the architecture components. To be able to calculate the global cost of a structure, all the states must be associated with the user-defined criteria (eg: area, power consumption, frequency,...). Moreover some criteria are also mandatory to enable the scheduling of the tasks and the routing of the data token. Below are given these specific criteria and the states for which they are required.

• Latency : this criteria is associated with the transmission or memorization of data. It represents the time between the activation of the operation and its real execution.
• BW : the bandwidth has to be given for the same states than the Latency. This is the data rate per time unit that can be memorized or transmitted.
• time : this criteria is required for the computing modes. It gives the time needed to execute an operation on a given platform block.

Except the computing states, the criteria (user-defined or mandatory) will be evaluated by the use of a Yeti model that has to be defined inside dedicated behaviours XML files. For the computing states, the criteria can either be evaluated via a Yeti model or during the explicit mapping of the functional blocks onto the platform blocks. In the second case, the user has to define two evaluation rules into the simulation file.

1. time-dependent rule : this rule specifies, for each platform block, how the criterion is computed over the whole timeline. It is expressed according to the time. Currently, three rules are available: additive, integrate and maximum.
2. composition rule : this rule specifies how the criterion is computed for all the platform, ie how the criterion is combined between all the blocks. Currently, two rules are available: additive and maximum.

3.3.3 A hierarchical modeling

The methodology implemented in Nessie is based on a common hierarchical representation of both the application and the architecture. Indeed, the goal was to allow the user to describe the system at any level of abstraction and to combine several levels in the same simulation to make a deeper and more accurate exploration of the possible solutions. The figure 3.11 illustrates this concept for the platform. In the figure, platform primitives are
identified by $P_{i,j}$ where $i$ is the related level of abstraction to which the primitive belongs and $j$ is the ID of the primitive in the lists of primitives. In the example, two levels of abstraction are considered. In the higher, the level 2, the primitive 5 is represented. For this primitive, two lower structures have been described at the level 3. These structures are composed of instances of the primitives defined in this level 3. The same principle is applied for the application part of the system. In this case, the primitives are the places of the Petri Network that can be themselves composed of lower structures.

### 3.3.4 The mapping core

Now that we have an idea of the way both the application and the architecture can be modeled into Nessie, we will spend some time to explain how the mapping of these two parts is managed by the framework.

**Timeline**

The mapping is based on an event-driven mechanism. Indeed, for each application/platform couple that Nessie has to simulate, a timeline is dynamically built with the events that constitute the different actions during the mapping at given time steps. This is shown on the figure 3.12. The timeline contains all the past and future events of the mapping and are triggered successively. Each time step is thus associated with a vector of events related to different platform blocks and states. The triggering is done in a first-come first-served
order. The different events that could be generated are the following:

- **state change**: this event is triggered when a core or port state changes. This implies that the criteria values can be updated based on the current time step value and the previous value. In addition, when the state changes from computing to idle, a new token is generated and stored in the platform, and the corresponding token is generated in the Petri net. For state change events, the HW block ID and type are specified, the new core state and also the sink block if the new state is related to a port or to a transmitting block.

- **hw release**: the release event is triggered when the block is ready to execute a new task. If it is in blocking mode, the block will be released when the token will be sent to another block.

- **data token reception**: this event notifies that a data token arrived to a platform block.

- **data token memorization**: when a platform block has memorization capability, it notifies when a produced data token has just been memorized and that it can been used inside the block or sent elsewhere.

Based on these events, the allocation and the routing are performed step by step following a so-called online method (contrarily to offline methods that examine the problem in its whole before performing the allocation and scheduling).

**Scheduling and Allocation**

Scheduling and allocation are often performed together as they depend on each other and should both be efficient to optimize the performance of the system. Scheduling is first performed depending on the data and control dependency, and on the tasks priority, according to criteria. Allocation then consists in identifying hardware blocks able to execute the application and allocating each task to these blocks. In Nessie, the user predefines the
platform blocks and fixes the compatibility list. The main part of the allocation process is thus already defined before Nessie performs the mapping. However, several things are not predetermined: choice of an execution order, of a platform block amongst several able to execute the same task, and of the best route to transmit a data from a block to another. The figure 3.13 summarizes the policy that is currently implemented into Nessie to perform the mapping of Petri network places onto hardware components. It is based on information exchange between the Petri network and the platform via the tokens.

**scheduling** Several policies exist in the literature to schedule tasks to be executed on a given platform. In Nessie, a simple policy has been chosen. First, the tasks (places in the Petri network) that have not yet been executed and are ready are stored in a *Ready for allocation* queue in the order they appear in the Petri network, from left to right top-down in the graph. The scheduling is then implicitly performed by fetching, in a FIFO order, the tasks from the *ready to allocate* queue. This means that these fetched tasks are progressively allocated to released hardware blocks and stored in another queue, the *Waiting for data token* queue.

**allocation** The allocation in Nessie consists in selecting a platform block able to execute the elected task and that minimizes the allocation weight. The allocation policy is based on the first-come first-served order. An elected functional block is allocated to a platform block based on two criteria. First, the block has to be compatible with the functional task. Second, if several blocks can execute this task, Nessie chooses the more efficient one based on the allocation weight defined by the user in the corresponding file. This weight can be a constant or depend on the criteria (energy, area, time,...). It can not be defined by the input parameters defined in the degrees of freedom of the simulation! If the weight is equal to 1 then, all the blocks are equivalent and Nessie will choose the first in the vector containing all the instances constituting the HW structure.

This communication mechanism between the application and the architecture parts and
online scheduling and allocation method allows to deal with heterogenous models of computation and to consider any kind of platform or application.

Routing and token management

We will spend more time to explain how the routing of tokens is managed in Nessie as it implies several problems. To remove tasks from the Waiting for data token queue, all the needed tokens have to be routed from the source (the hardware block that has produced the token) to the hardware block on which the task has been allocated (the consumer of the token). Once all the required tokens have been routed to the waiting place, the task is added in the Executing queue. The associated hardware block can then execute the task. At the end of this execution, a new data token is generated in the platform, that must be sent to the block allocated to the next task. A corresponding token is also generated in the executed place of the Petri Network that can eventually fire a transition to enable next tasks.

Routing policy  the routing is based on Dijkstra’s algorithm which finds the shortest path from a source to any reachable node in an architecture. Typically, it is represented by a graph linking vertices (communication nodes) to non-negative edges (communication links) associated with weights defining their cost. Edges can represent either unidirectional (oriented) or bidirectional (non-oriented) links. The use of this representation in Nessie has been completed by some important concepts. First, routes are considered only between a producer and a consumer. Second, a token can be sent from the producer to the consumer only through nodes having transmitting capability. Third, routes are available only if the composing nodes are in idle state, not busy and not reserved. Finally, to evaluate the cost of a path, the weight of each node composing it will be added together. Logical links do not add any cost to the path. Note that only non-negative weight are used to compute the route.

As it is the case for the allocation policy, the user can adapt the routing policy by defining a weight in the routingWeight.xml file. This file is common to all the levels of abstraction. To avoid deadlock in some situations, several parameters can be used in the Yeti model defining this weight (the user must respect the names inside the model).

- latency : time between the complete reception of a data token and its complete sending for a platform block.
- BW : bandwidth of the platform block.
- numberOfNeighbours : number of neighbours around the block. This can be taken into account in case of network congestion. Indeed, a component with a lot of neighbours blocks several routes if it is used.
- numberOfCompatibleSWtypes : gives the number of different task types the platform block is compatible with. If it can execute several functional blocks, it can be interesting to let it free for computing rather than for routing.
3.3. OUR SIMULATION ENVIRONMENT: NESSIE

- **hasInterconnectCapability, hasComputationCapability, hasMemorizationCapability**

  used as boolean parameters (1 if yes, 0 if no) to indicate that the possible modes of
  the platform block.

The user can also decide to set the weight to 1 which corresponds to a hop count for each
route. Nessie will thus choose the available route with the smaller hop count (equal to the
number of crossed blocks).

The broadcast can be useful when a same data token has to be consumed by several
blocks. In this case, it would be more time efficient to allow a communication block
to transmit directly all the data token at the same time rather than considering each
consumer separately. This functionality has thus been implemented in Nessie.

**selection policy** among all the ready consumer/producer couples, the choice of the
one that will be served first depends on the choice of the consumer and the producer
for a needed token. In Nessie, the choice of the consumer is done in the FCFS order
from the **waitingForTokens** queue. Then, amongst the producers able to deliver a re-
quired token for this consumer, Nessie will choose the one that gives the more profitable
route. This profitability is measured via a metric taking into account the number of con-
sumers \( \# \)\_consumers\( (P_j, D_k) \) that can be reached given the current occupation state
of the platform block for a route starting from \( P_j \) for the token \( D_k \) and the total distance
\( \text{Dist}(P_j, Consumers(P_j, D_k)) \) of the path from \( P_j \) to all currently consumers of the data
token. This metric is given in the equation 3.4. The goal of the metric is to choose the
solution with the smallest total distance per data token sent.

\[
\text{Metric}_{\text{token/producer selection}} = \frac{\# \text{consumers}(P_j, D_k)}{\text{Dist}(P_j, Consumers(P_j, D_k))} \tag{3.4}
\]

Once the producer and the route has been selected, the route is reserved, meaning that
it can not be used until the token has been received by the associated consumer. We
have to notice that to calculate the criteria along the path, Nessie takes the minimum
BW available amongst all the blocks in the path and bases the time calculation of the
data transfer on this minimum value. The figure 3.14 shows the different events that are
classically generated during the routing of a token from a platform block to another.

**memorization** when a token has been produced by a platform block, meaning that a
task has been executed, it is placed in its **produced token** queue. Then, if the platform
block has memorization capability, the token is automatically stored inside the block which
can then execute other tasks. If it has no memorization capability, the token remains in
the platform block, blocking it, meaning that it can not execute another task as long as
the data token has not been sent to the following task.

**deadlocks** deadlocks can appear when using a computing block that does not have
memorization capability as they enter in blocking mode and can not be used anymore
to transmit or compute another task until the blocking token has been consumed by
3.4 Input/Output files

In this section, we give more information about the content and structure of the input and output files. This will be useful to understand practically, along the case studies, how the modeling in Nessie is introduced by the user and how results can be analyzed.

3.4.1 Simulation.xml

The simulation.xml file contains the main informations about the systems, parameters and criteria that are used to model applications and architectures. The file is composed of four parts: the criteriaList, the SWdescription, the HWdescription and the DOF (Degrees of Freedom).

criteriaList

In this part of the file, the user gives the name of the criteria that have to be evaluated. For each criteria, he must indicate how these criteria are time dependent (timeDependent
attribute) and how they have to be combined on the entire platform (combinationRule attribute).

**SWdescription**

This part is composed of the SWhierarchy and the SWstructuresList.

- in the SWhierarchy, the user defines all the primitives (SWsubTypes) that could be used in the Petri networks, at each level of abstraction one is interested in (identified by the abstractionLevelNumber starting at 0). Each SWsubTypes is identified by a number ID and must be filled by a SWparametersList if necessary. Moreover, the user must specify the data output size.

- in the SWstructuresList, we find all the Petri network structures that the user wants to define. Each structure has three attributes: an index that numbers the structure amongst the list of structures, the ID of the SW primitive that the structure describes and the AL at which this SW primitive belongs. In each structure, the user must define the list (netList) of the places (petriNet and dummyNet) that are identified by an ID number and the type of primitive they are an instance of. Then the list of the transitions (transitionsList) are given and will define the structure of the petriNetwork as each transition is associated with its input and output places and the number of tokens entering and going out the transition.

**HWdescription**

The structure of the HW part is quite similar to the SW. However, extra information has to be filled in by the user.

- when defining the HW primitives (HWsubTypes), the user must give for each state associated with the block the path of the behaviour.xml file. A special section is dedicated to the definition of the computing states for which the user must specify the ID of the SWsubType in the compatibility list. A third part is dedicated to the definition of the IO ports and the related behaviours. Finally, a transitionTimeTable is foreseen to give latencies between states switches.

- as for the SW structures, the user begins by giving all the HW instances that will be used in the structure. Then the structure is defined in the linkList: each link is associated with a bidirectional attribute (true if the link is bidirectional) and with the source and sink HW block IDs.

**DOF**

In this last section of the file, all the degrees of freedom of the simulation are specified: parameters, HW structures, SW structures.
• the user must specify the value(s) of the parameters that will be used in the closed-form expressions. The values can be entered as a list of values, a single value or a sweep between an initial and a final value with a given step.

• in the tag structureDOF with the attribute typeOfStructure HW, the user gives the structures ID that have to be simulated, amongst all the HW structures he has defined in the file.

• as for the HW, the SW structures counter part have to be chosen by the user for the simulation.

During the simulation, Nessie will produce results for all these DOFs.

3.4.2 AllocationWeight.xml and RoutingWeight.xml

These files have the same structure than the behaviour.xml presented in the section 3.2. The user must just use the allocationWeight and routingWeight names as associated and output parameters of the relation in each file respectively.

3.4.3 NessieCriteriaResults.xml

This output file contains all the results related to a given simulation. It means that there is one file for each simulation and that it contains the results for all the DOF defined in the simulation file. For each solution, there is a flag telling if the solution is valid (flag=’1’, no deadlock) or not. Then, we find the criteria and the DOF.

• the criteria are listed in three parts: the time, the timeDependentCriteria and the timeIndependentCriteria, together with their values.

• in the DOF section, we find the parameters with their value for the current solution and the HW and SW structure ID used to compute this solution.

3.4.4 TimeLine.xml

The timeline contains all the events that have been created and triggered to perform the mapping of a given solution. It is composed of each time step for which are given all the added events, i.e. the events that have been added on the timeline progressively, and of the triggered events, which are the events, previously added, that are effectively executed when the timeline is covered. The more the time step, the longer the file. Contrarily to the criteria Result file, there is one separated timeline.xml file per solutions.

3.4.5 ActivityReport.xml

Finally, Nessie produces an activity report of all the HW blocks composing a structure. There is thus one activity report for each solution. In this file, the user can find all the components with their ID. For each, the core states and the ports states (for each port) are given. Each state is associated with two attributes: the relativeTimeOccupation which
gives for the current block the time percentage for the current state compared to the other
states and the absoluteTimeOccupation with gives the total absolute time spent in the
current state for the given HW block. The size of this file can rapidly grow if the number
of HW components increases in the structures.

3.5 Conclusion

In this chapter, we have presented our original multi-criteria prediction tool, NESSIE,
and its model estimation engine YETi. We have seen that the framework is based on a
simple mapping and exploration policy to enable fast simulation of solutions. Moreover,
we have presented the different ingredients that have been implemented to allow the use of
NESSIE on different kinds of application and platform and make the tool non specific to a
given target (user-defined parameters, degrees of freedom, allocation and routing weight,
criteria and HW/SW primitives).

In the next chapters, we will apply NESSIE on real case studies to see if this simplicity
and generality are not an issue for the modeling of existing design problems. In particular,
through the analysis, formalization and modeling of these case studies, we will identify the
limitations of the tool and discuss its place in the race of the complex embedded system
design. This work will allow us to give an answer to the following questions resulting from
Chapter:

• are the simple policies implemented for the mapping not too simple regarding real
design problems?
• is the allocation and scheduling indeed flexible, suitable?
• are the simulations as fast as hoped?
• are the deadlock manageable? are they an obstacle for the simulations?
• is the hierarchical representation effective?
• how easy is it to model systems into Nessie?
• is the tool able to predict performances?
• is there a gain to use Nessie in place of or in complement to existing tools?

The rest of the dissertation, which constitutes the core of this work, is divided in three
parts.
In Chapter 4, we present our main case study, based on a design flow used at IMEC, and
which has been considered at a low design level.
In Chapter 5, we present the second case study, targeting the design of new 3D stacking
MPSoCs. This design problem will be studied at a higher level of abstraction.
Finally, as a result of these chapters, we present in the "Future Work" the conclusions and
the perspectives proposed to improve NESSIE with further developments.
Chapter 4

Case Study 1 : IMEC

Abstract
In this chapter, we present our main case study, which constitutes the core of the work for validating NESSIE. Therefore, we start by presenting the design context and the external IMEC flow that has been chosen. This is followed by an analysis and a formalization of the flow (mainly made up of a dedicated compiler, DRESC) in order to identify the system that will be modeled in our tool, which is the ADRES processor applied to a matrix multiplication algorithm. Then, a feasibility study is presented to discuss the modeling capability of our tool applied on this case study. We continue by defining the solutions that will be explored before presenting the results in a dedicated section. The analysis and consequent discussion of the results are done on a comparison of a critical criterion between solutions generated by NESSIE and those estimated in the lower flow, by DRESC. This discussion is completed by a multicriteria evaluation of pre-selected solutions to study the prediction capability of our tool. The results will show that NESSIE is able to estimate with a high degree of confidence solutions for this case study, modeled at a low level of abstraction. Moreover, we finish the analysis by a quantification of the design and modeling time in both flows what highlights advantages and limitations of NESSIE.

4.1 Introduction
In Chapter 2, we have presented the original tool Nessie which has been developed in order to improve the design time and the quality of the designed system. This tool acts as a performances prediction tool and should ideally be able to explore a wider space of solutions than a classical design tool in less time and as results provide dependable solutions via the evaluation of multiple criteria. The goal of this thesis is to validate NESSIE, i.e. to highlight its benefits and/or its
limitations, with an external case study on a realistic and current problematic. The case study was provided by IMEC (the Interuniversity MicroElectronics Center)\(^1\) which works with a complete top-down flow on the design of MPSoCs for wireless telecommunication systems. The validation upon this case study has been divided in three parts which are detailed in the next sections. First, we present and analyze the design flow used at IMEC for the design of an MPSoC wireless receiver for 3GPP LTE telecommunication standards. Second, based on this analysis, we formalize the design problem and explain the modeling feasibility and the choices made to simulate it into NESSIE. Finally, we present and discuss the results from the viewpoint of performance prediction capability and gain on design time.

4.2 MPSoC design applied to telecommunication systems

In this section, we globally present the design flow of an entire MPSoC platform, starting from a Matlab application description down to a gate representation of the final system. We first give few words on the context of this design problem to understand why this case study is a good candidate to validate NESSIE and what, more precisely, we have focused on for the modeling.

4.2.1 Context

One of the major highly constrained embedded system axis concerns the design of new generations of multiple users wireless telecommunication terminals. Such a system is typically composed of a base station, the transmitter, which sends signals to several mobile receivers. A lot of research have been done in this field to build algorithms and protocols that guarantee an efficient transmission of signals and improve the reliability of the communication. Indeed, signals are subjected to interferences, reflections, noise and fading. A well-known paradigm to face these problems is the MIMO concept\(^1\). MIMO stands for Multiple Inputs, Multiple Outputs, and consists in considering several antennas at both the transmission and reception of the communication system. This solution is particularly adapted to deal with data rates increase. We will not enter in the detail of such telecommunication algorithm because this is not the purpose of the work. However, it is important to notice that algorithms defined to deal with such systems are very complex and thus computational intensive. This is the counterpart to pay if one wants to improve the quality of the application. Nowadays, a lot of research is done on MIMO system to reduce the complexity of such algorithms.

Aside the signal transmission efficiency, the wireless telecommunication world faces the rapid incursion of new standards (such as 802.11.n, 3 GPP LTE, Mobile WiMAX, DVB-H, etc) and content formats (such as MPEG-2, MPEG-4, AVC/H.264 and Scalable Video Coding). To deal with such issue regarding development time and costs constraints, research and industries are working on rapid adaptiveness of mobile devices that could adapt themselves at real time to several standards depending on the environment. This is why

\(^1\)www.imec.be
the use of programmable and/or reconfigurable architectures for the digital baseband processing has been proposed as the ultimate way to be cost-effective in the future processing technologies. The so-called Tier-2 SDR, the software-defined radio paradigm, bases the platforms on the use of flexible processors and multiprocessors on the chip. Moreover, to achieve high performance and high energy efficiency in such architectures, designers leverage on parallelism like ILP (Instruction Level Parallelism) or DLP (Data Level Parallelism). In such context, MPSoC platforms are a worthwhile platform candidate. At a lower level of abstraction, the use of flexible and parallel intensive processors is required to optimize the execution time and the consumed energy which are two main critical criteria in such embedded systems.

IMEC precisely designs such SDR systems that are constrained by multiple criteria and offer a lot of degrees of freedom. In particular, in their case study, the specified functionality is a 3GPP MF wireless receiver and the platform is an MPSoC mainly composed of ADRES processors[2], which are parallel intensive programmable and reconfigurable architectures created in IMEC. They perform the design via a top-down tool chain starting from a Matlab specification of the application down to the physical design of the platform. This design problem, integrated in a complete design flow, is our main external case study to put NESSIE to the test. In particular, we will perform the validation of our tool on the ADRES design flow, which is the main component of the MPSoC platform and already gives a wide multicriteria design space to explore. Based on the flow, presented in Section 4.3, we will divide our study into the three following parts:

- first, we analyze and formalize the ADRES platform and its design flow.
- second, we study the modeling feasibility in NESSIE and propose a validation case.
- Third, we present the results estimated by NESSIE, compare them to results produced in the ADRES flow and discuss their validity.
- Finally, we quantify the design time gain when using NESSIE in the existing flow.

As an introduction to this study, we will present the global flow used to design the SDR 3GPP LTE MPSoC system to point out the complexity of such embedded systems design and confront this concrete flow to the state-of-the-art concepts presented in Chapter 2.

4.2.2 A global picture

A schematic view of the global tool chain used at IMEC to design an MPSoC platform is shown in Figure 4.1. The flow can be seen as four levels of abstraction, starting from the specifications to the gate level at which the platform can be physically produced. The different stages that progressively refine the system are explained below. As formalized

\[ \text{3GPP family of standards consists on 3 alternative air-interface (physical layer) standards, which all share higher layer protocol stack and core network / back office architecture. These air-interface standards are FDD W-CDMA, High Chip Rate UMTS TD-CDMA, and low chip rate TD-SCDMA.} \]

\[ \text{Catalytic image from [3] - Coware image from[4]} \]
in Chapter 1, each stage is characterized by the type of decision tool (automatic or not, which mapping algorithm, which criteria, exploration capability), the inputs of the tool (application/architecture descriptions, degrees of freedom, models, components library), the outputs of the tool (application/architecture descriptions) and the simulators associated to the decision tool. So far, we have identified these different components for each stage of the global flow.

**Specification to Matlab**

The specifications typically inform the designer of the desired functionality (i.e. the application) but also about non-functional constraints (the data rate and the Bit Error Rate (BER) but also the energy consumption and the execution time). Moreover, design time deadlines must be respected. All this information should condition the design from the first steps of the flow.

As it is commonly the case, the first step of translation of the specifications consists in describing the application of the system in Matlab. The language used by Matlab is particularly dedicated to signal processing and is really convenient for a fast representation of the functionality. This description is often performed by the designer which uses coding rules and functions that limit the complexity of the algorithm and preserve the functionality of the application. Also at this level, the designer partitions the application in blocks based on the duty-cycle \(^4\). The typical partition of the functionality is shown in Figure 4.2. We find a Digital Front-End (DFE), composed of the radio front-end and the packet detection that have high duty-cycles, the (de)modulation (in the baseband engine), the forward error correction (FEC) and the medium access control (MAC) that have lower duty cycles.

**Analysis:** we notice that :

- at this level, the designer does not take the platform into account when describing the application. He considers the platform as a generic processors-based black box and partitions the application to target either high-energy efficient components with low flexibility or energy-scalable components, like programmable/reconfigurable units. Of course, the kind and number of processors, the way they will communicate will directly impact the performances of the system. Unfortunately, at this stage, the application and the platform are not defined jointly.

- in addition, the coding of the application is based on two main criteria: the complexity, which is implicitly linked to the execution time of the application when we make the assumption that processors will be used, and the BER which gives information about the accuracy of the signal computation. However, the complexity is not always representative of the performance of the final system. Moreover, other

---

\(^4\text{In telecommunications and electronics, the duty cycle is the fraction of time that a system is in an "active" state. [...] The duty cycle is the proportion of time during which a component or device is operated. (wikipedia)}\)
Figure 4.1: A schematic view of the global design flow for 3GPP MF MPSoC systems.
critical criteria are of importance for embedded systems and should be considered at this stage as the early decisions will impact the next design steps.

Matlab to SystemC-TLM

The next stage aims at jointly taking into account the application and the architecture still at a high level of abstraction using the Transaction Level Modelling\(^5\). At this level, the application is described in ANSI C and the platform in SystemC. The refinement of the application from Matlab to C is automatically performed via the Catalytic tool\(^5\). The platform, on the contrary, is defined initially by the designer through a virtual platform specification in a graphical interface thanks to the CoWare Platform Creator\(^6\). The designer selects the hardware components in a library (ConvergenSC library) and composes the platform himself. If the components are not available in SystemC, they can be directly replaced by a lower description, in VHDL, as the tool enables co-simulation of SystemC and VHDL components.

To perform the functional simulation of the system, software part of the application are executed via Instruction Set Simulators (ISS) that are encapsulated into SystemC wrappers which communicate with the other components of the platform. If the ISS is not available, it can be modeled by ASIPs (Application Specific Instruction set Processors) generated by tools like LISATek\(^6\) or via a third party ISS as it is the case for the ARM processors. For ADRES processors, two kinds of software simulator can be used\(^7\): a precompiled simulator generated with the dedicated DRESC compiler\(^7\) based on an XML description of the architecture or an Analytical Virtual Machine (AVM), a C++ program which reads the XML description file of the processor and an object file generated by DRESC.

**Analysis:** we have observed that:

- the automatic generation of the ANSI C code is done for generic DSP and is generally not optimal for other processors.

---

\(^5\)we refer the reader to Chapter2 for the definition of the TLM.

\(^6\)www.coware.com

\(^7\)We detail their characteristics in Section4.3.4.
the performances estimation is done at two TLM levels: untimed level or bus cycle accurate level. The simulations are based on time criteria and do not take into account other important criteria like the energy consumption for example.

- the mapping is done by the designers who allocates the application on the architecture components following the partition done at the previous level. This shows the importance of the decisions made at the Matlab level. This mapping choice is based on implicit cost models, i.e. by the expert according to its qualitative experience, what leads most of the time to wrong or too approximative decisions and a lot of iterations. Tools like NESSIE are devoted to the use of explicit models to decide on quantitative estimations and avoid the iterations when refining the system.

C to binaries/assembly and SystemC to VHDL-RTL

To progressively go to the physical description of the system, the application and the architecture are refined differently depending on the type of component.

1. For processor’s applications, the C code is compiled by dedicated tools. This is the case of ADRES for which the algorithm will be compiled by DRESC. The output of DRESC is a binary file. This will be detailed in the next section.

   Analysis: the decisions made by the compiler will be independent of the RTL refinement of the processor itself. The main criterion that is considered when simulating the code is the execution time which is optimized by the compiler regarding the description of the processor at the instruction level.

2. For the processor architecture and the other components, the SystemC description is refined in VHDL. Different ways are used depending on the availability and complexity of the IP’s. The description can be done manually, automatically via the tool (platform creator) or just by taking IP’s available in standard libraries (ARM family eg). For the ADRES processor, a parser is used to get a VHDL description of the processor based on a XML description file.

   Analysis: multiple criteria can be estimated at this lower level, by co-simulation or by emulation on FPGA: resource usage, energy consumption, cycle accurate timings. These late multicriteria estimations are time expensive and often leads to costly iterations. This is due to the earlier decisions that have been taken on implicit models and on mono criterion simulations of the system, implying that the chosen solution does not directly satisfy the specification.

VHDL-RTL to Gate level

Then a synthesis takes place to get a gate level description of the architecture of the platform from the VHDL language. Several dedicated tools exist as in the Mentor Graphic suite. This step in particular has not been studied and will not been detailed here.
4.2.3 Link with the SoA

The design flow presented above is quite representative of a classical tool chain used in the industry of embedded systems where validation of solutions is performed \emph{a posteriori}, coupled with no automatic exploration tool (i.e. solutions are tested one by one and chosen by the user) and simulations performed on high levels of abstraction which are essentially based on timing constraint and do not take into account other required criteria like the energy consumption or the total area of the chip.

\textit{With our tool NESSIE, we want to tackle this drawback by estimating the performances a priori.}

However, to limit the design time and to be able to converge to satisfying solutions, the IMEC flow uses typical ingredients that we have pointed up in the State of the Art (chap 2).

- \textit{reduce the design space} : by choosing a kind of platform, designers reduce the design space a lot. In particular, they have chosen to deal with MPSoC platform offering parallelization capability, enabling performance optimization, cost reduction, easy adaptation of the system at run-time. Moreover, amongst the possible processors, reconfigurable ones have been favoured because they enable scalability by optimizing energy and performances at run-time depending on the external context and reduce production cost.

- \textit{add high level of abstraction} : even if the design space has been reduced, many possible solutions still exist. To enable a fast exploration of solutions, designers start the modeling of the system at the TLM (Transaction Level Model) level via the use of the SystemC language and IP’s. However, all constraints are not modeled in SystemC.

- \textit{use of platform-based design and codesign methodologies} : due to the inherent heterogeneity of MPSoC platforms, both processors (SW) and integrated circuits (HW) are combined and have to be designed together. This justifies the use of so-called HW/SW codesign tools which consist generally in performing HW (FPGA/ASIC) / SW (processors) partitioning of the global application code onto a virtual platform. The refinement of the HW and SW is considered separately (VHDL vs C development). Then, co-simulation and co-verification can be performed by jointly considering the HW and SW parts of the platform. Co-emulation is also used for the RTL components in a next phase to accelerate the simulation step. Moreover, the use of IP’s, as proposed in platform-based designs minimizes the RTL verification effort and also reduces the exploration time.

Of course, the designers are aware of the fact that time can be reduced and that better solutions are surely not considered. They would like to use better exploration policy and dispose of a tool able to quickly predict the right solutions to explore deeper. The philosophy implemented in Nessie has aroused a lot of interest. In order to experiment
our tool on a reasonable case study, we have decided to perform a deeper analysis of their flow on a component of the MPSoC platform reduced to the execution of a small part of the application. As a lot of information is available about it in IMEC, we have chosen to deal with the ADRES processor. As we will show, this processor deserves much interest regarding the numerous degrees of freedom it offers to the designer. Moreover, it illustrates very well how correlated are both the application and architecture structure considering the final performance of the system.

The next section is entirely devoted to the presentation of this platform and the corresponding design flow.

4.3 The ADRES design flow

With the exception of the Matlab level, the complete design flow of the ADRES processor is shown in Figure 4.3. The flow is globally composed of the same levels of abstraction than those presented for the entire MPSoC platform. For the application, we find again different automatic design steps from a C description to binaries. For the processor, the architecture is described in an XML file and refined in VHDL down to the gate level thanks to dedicated tools. The refinement from the Matlab specification to the ANSI C description of the application is not represented in Figure because this phase is done manually by the designer as it targets a specific processor. However, this design step is critical for the final performance of the system.

In our work, we have restricted the study to the higher levels of the flow composed of this Matlab to C level and of the DRESC compiler generating the binaries, shown in the left side of Figure 4.3. These two design steps are detailed below.

- Precompiler stage: the first stage that we consider consists in refining the Matlab description of the application into a C code. In the particular case of ADRES, this stage is performed manually by the designer. Again, the designer must have an idea of the architecture that will be chosen to create a sufficiently efficient code. However, the choice of the ADRES architecture is done independently of the C code definition and is maintained until a satisfying algorithm solution is found at the end of the flow. Finally, this stage has been called precompiler because it aims at preparing the C code that will be compiled with the dedicated compiler DRESC at the next design step.

- Compiler stage: when a C code has been produced, it is compiled by DRESC which also takes the XML description file of the ADRES template as input. This tool performs the mapping of the application on the processor in order to minimize the execution time. As we will detail in the next parts of Section, the data intensive part of the application is mapped on the highly parallel part of the processor according to a so-called software pipelining technique. To evaluate the quality of the mapping and the chosen solution, DRESC is coupled with simulators that gives an estimation of the cycles, the resource usage, the mean number of instructions per cycle,... Based on these metrics, several iterations are classically done from the precompiler stage
Figure 4.3: Design flow of the ADRES processor - from the ANSI C to the Gate Level
where the user will adapt the application description, keeping the ADRES template. Finally, when the estimations satisfy the designer, binaries of the application can be produced by DRESC to feed the physical platform of the system that will be synthesized.

It is important to notice that in this flow, the main criterion that is estimated is the execution time. The energy consumption will be evaluated at lower stages. Even if the energy consumption is linked to the execution time in the case of ADRES\(^8\), no quantification of this criterion is performed at these stages. In addition, we already see that a lot of time can be won if decisions made at the precompiler stage were based on a smarter exploration of the solutions.

In the next parts of Section, we will further detail the components of the design flow:

- The architecture of the reconfigurable processor ADRES
- The software pipelining technique and the modulo scheduling paradigm
- The DRESC compiler: design steps, mapping algorithm
- The simulators linked to the DRESC compiler and the metrics

### 4.3.1 Architecture of the ADRES processor

ADRES stands for "Architecture for Dynamically Reconfigurable Embedded Systems". Its structure is depicted in Figure 4.4 where we distinguish two functional parts: a VLIW (Very Long Instruction Word) and a CGA (Coarse Grain Array). These parts share a common row of functional units and global register files to communicate together. As explained in paper [2], the VLIW is dedicated to the execution of the control part of an application, while the CGA is dedicated to the parallelization of data dominant part of an algorithm, targeting computational-intensive programs where "a program spends 90\% of its execution time in only 10\% of the code".

ADRES is in fact a template that can be fully defined by the user in an XML file. Thereby, the number and properties of the components can vary, as well as the topology of the interconnection.

In the next parts, we present respectively the ADRES components and several interconnection schemes. Then, we give some words about the execution pipeline in both VLIW and CGA modes.

**The components**

\(FU\) : the functional units are the computation blocks of the processor. They are configurable as the user can specify in the description file which instruction set is admitted

\(^8\)The ratio \(P_{off}/P_{on}\) (static power on dynamic power) is high whatever the array size \((P_{off,FU} - P_{on,FU} \ll P_{on,FU})\). To reduce the power consumption, we must thus reduce the number of cycles.
Figure 4.4: Architecture of the ADRES processor

Figure 4.5: Hardware components of the ADRES processor - from left to right: functional unit, register files, transition node, constant memory
for each FU separately. The first row of FU determine the instruction parallelism in the VLIW mode. In this mode, the FU are able to access the external memory thanks to load/store instructions. Moreover, to switch between the two modes, other special operations are used and enable control flow transfers (CGA, HALT, BR(L) and JMP(L)). If operations are extracted from 32-bits wide RISC-like instructions in the VLIW mode, they are split in sub-operations for ALU and register transfer in the CGA mode. The inputs of the FU’s are named src1, src2, src3 for the 32-bit operands and pred for predication bit. The outputs are dst1 for the 32-bit results and pred.dst1, pred.dst2 for predication transfer. The size of the CGA is defined by the user who chooses the number of rows and columns in the processor.

**RF** : there are two types of Register Files: data RF (32-bits or 64-bits values) and pred-icicate RF (1-bit values). Global DRF and PRF have to be present in the ADRES template and need at least 2n read and 1n write ports for a n-issue VLIW. These global registers can thus be accessed in the two modes. They enable the data exchange between the VLIW and the CGA parts of the processor. Optionally, local RF (data and predicate) can be added in the array, only accessible in the CGA mode.

**TRN** : transition nodes routes data. They can act as latches (one input - one output) or as muxes (several inputs - one output). The latches add delays and can store predicates (1 bit wide) or data (32 bits wide). It is used to route a data from an FU to another. The muxes are configured thanks to configuration words, add no delay and are used to choose an input amongst several.

**CM** : constant memories store immediate operands in the CGA mode. In the VLIW, the immediate operands are included in the instruction encoding. The width of constant memories is specified in the template of the processor. Each CM has to be connected to at least one FU via muxes.

---

**Figure 4.6:** Detail of a functional unit (FU) structure and its local register files (PRF and DRF)
The register distribution and the interconnection schemes

Figure 4.7 illustrates typical interconnection schemes between functional and register files on a 4x4 architecture template. On the top of Figure, we see variants of RF distributions. Figure (a) shows a fully distributed RFs topology where each FU, but the first row, has its own local RF that is also connected to a remote FU.

Other possible distribution schemes are possible. Indeed, we can have scenarios where no RF but the global VLIW RF is included (figure 4.7(b)) or partially distributed as in Figure (c).

We could also have a shared RF to which all FUs could access in both read and write modes.

In addition to these possibilities, FUs can be interconnected together in different ways: full mesh as shown in figure (d) where only the direct neighbors communicate together, mesh-plus in figure (e) where extra connections are added and Morphosys which add connections to remote neighbors (figure (f)).

A detailed view of the connection between FU and RF is shown in Figure 4.8. In Figure (a), the local RF can be written by diagonal FUs and is only accessible for reading by its close FU. On the contrary, in Figure (b), only the close FU can write in its local RF, but diagonals FU’s can access the RF by reading.

The execution sequence

In VLIW mode, operations are fetched from main memory, possibly through the use of a L1 instruction cache. These instructions are then decoded and issued on the FU’s assigned to the VLIW. These functional units are connected to Global Prediction Register Files and Global Data Register Files. In addition, the VLIW is connected to an external Data Memory and an Instruction Cache. The Control Unit manages the instruction execution that is performed in several steps. The stages of the pipeline are: Fetch, Decode, Execute (typically 4 slots in parallel), write back.

In the CGA mode, the FU’s act as a data flow. All the entities of the array are configured via configuration memories. No runtime decoding is necessary in this mode as the configuration bits directly fix the state of the entities. The stages of the pipeline are: Fetch from configuration memory, Execute. The output is always registered (either in RF or in output buffer).

The description of the architecture shows that ADRES is a highly parallel architecture enabling high data flow and intensive pipelining. Indeed, thanks to the CGA, several operations can be executed at the same time and a wise interconnection scheme enables to chain the execution of successive dependent operations. To take full advantage of this structure, the DRESC compiler tries to extract the potential parallelism from the application. This is done thanks to a software pipelining technique, a notion that we discuss in the next section.

4.3.2 Software Pipelining and the modulo scheduling

The paper[8] defines the Software Pipelining as follows:
4.3. THE ADRES DESIGN FLOW

Figure 4.7: Interconnection schemes between FUs and RFs: (a) fully distributed RFs; (b) no distributed RF; (c) partially distributed RFs; (d) FUs Mesh; (e) FUs Meshplus; (f) FUs Morphosys.

Figure 4.8: Interconnection detail between FU and local RF: (a) REG_CON1 interconnection scheme; (b) REG_CON2 interconnection scheme.

"Technique [...] where iterations of a loop in a source program are continuously initiated at constant intervals without having to wait for preceding iterations to complete"
This technique for scheduling instructions has appeared to exploit ILP in inner loops (the most data intensive part of the application) by enabling parallelism between loops iterations and refers to a class of so-called global cyclic scheduling algorithms\[9\]. This is particularly adapted for multimedia/telecommunication applications where a lot of repetitive operations have to be performed and algorithms are composed of a large amount of loops, in particular for loops.

An example of typical software pipelined loop is shown in Figure 4.9 (c). The C code of the loop is given in Figure (a) and its pseudo code in figure (b). Traditionally, the body loop is executed sequentially for each iteration. When applying the software pipelining, instructions of the loop body of successive iterations can be executed at the same time. As classical hardware pipelines, the software pipeline is composed of a prologue that fills the pipeline, the kernel, which constitutes the steady states of the pipeline, and an epilogue which drains the pipeline.

The ability to parallelize the loop depends on the data dependence between the operations inside an iteration (intra-iteration) and between iterations (inter-iterations), and also on the available resources of the platform that should execute the loop.
The way resources are taken into account will depend on the scheduling algorithm that will be chosen to perform the software pipelining of the inner loops. Several techniques and methods exist to generate software pipelined loops\[9\]. The \textit{Modulo Scheduling} is currently one of the most effective framework and covers a wide variety of methods, algorithms and heuristics that can be defined to generate software pipelined loops. The DRESC compiler is based on a metaheuristic that performs \textit{iterative modulo scheduling} for C algorithms.

The \textbf{Iterative Modulo Scheduling technique}

Modulo scheduling is a technique to generate software pipelining on innermost loops. This methodology is widely used because of its efficiency and the fact that the management of the execution of the code is all determined contrarily to other software pipelining methods. The goal is to find a schedule for one iteration of the loop and apply the same schedule for the other iterations at regular intervals such that no inter- or intra-iterations dependence is violated and resource usage conflicts between operations is avoided. The regular interval that separates the execution of consecutive iterations is called the \textit{Initiation Interval} (II). We illustrate this technique on the example of Figure 4.10 representing the DDG of a loop body composed of four operations. In this DDG, the first number on the arcs gives the dependence between two successive iterations, the second gives the dependence between two successive operations in the body loop.

An example of scheduling of the DDG on a 3x3 CGA is shown in Figure 4.11. As the same schedule is applied for each iteration, which means that the HW resources are used identically for each iteration of the loop, a minimum II of 1 is required in the pipeline. This supposes that the resources can execute a new operation each cycle, what is the case in the ADRES processor which is a fully pipelined architecture. The only condition that constrain the execution is that two operations cannot start or finish at the same cycle on the same FU. However, in this example, we consider that there is no dependence between iterations and
Figure 4.11: Illustration of the modulo scheduling on a 3X3 array
no resource constraints regarding the number and type of operations of the DDG. In a general case, the II will depend on the loop-carried dependence (inter-iterations) and on the available resources. This is characterized by the recurrence-constrained II (RecII)\(^9\) and the resource-constrained II (ResII) respectively.

The Modulo Resource Table (MRT) is an easy way to compute the ResII. In this table, the columns represent the available resources (slots) and the II is given by the cycles between two iterations in the rows. An example is shown in Figure 4.12 where two slots are available to execute operations of the DDG (fig 4.10). We see that a new iteration can begin every two cycles and that the ResII is equal to 2.

The scheduling algorithm: concretely, to find a valid schedule, the DRESC compiler relies on an Iterative Modulo Scheduling algorithm. The principle is to calculate a II based on the MRT and the DDG. This II will be the starting point on which the algorithm will iterate until a schedule is found.

This starting II, called Minimum II (MII) is thus equal to \( \text{Max}(\text{ResMII}, \text{RecMII}) \). The final II will generally be bigger than the MII due to added delays (extra data transfers, predicate operations,...). Depending on the parameters of the algorithm, the scheduling time for a given II will be more or less long and the final II more or less big. The smaller the II, the higher the ILP what also minimizes the execution time of the pipeline. The complexity of such algorithm is \( O(N^2) \).

Once a valid schedule has been found, the compiler must manage the prologue and epilogue of the pipeline. Indeed, in the steady state, the same resources will be configured in the same manner for each iteration with a II period. However, for the prologue and the

---

\(9\) A loop contains a recurrence if an operation in one iteration of the loop has a direct or indirect dependence [data dependence like flow, anti or output or control dependence] upon the same operation from a previous iteration."[9]
epilogue, some resources are not used and have to be cancelled. This is illustrated on Figures 4.13 and 4.14 where we have represented the complete instructions that should be executed if there were no prologue and epilogue (parts surrounded with dashed lines). The red operations in Figure show the operations that have to be cancelled by the compiler when starting and stopping the pipeline on the CGA. Thanks to the use of rotating registers and predicate operations which are used to manage the prologue and epilogue, the compiler can act as a kernel only execution.

The metrics: the determinism and the regular scheduling resulting from this software pipelining technique enable the characterization of the schedule and the loop with several metrics. These are illustrated in Figure 4.15. Apart from the II, the different important values characterizing the modulo scheduling are:

- Iterations: it corresponds to the number of iterations of the loop. The more the number of iterations, the smaller the overhead of the prologue/epilogue on the kernel in the pipeline, the better the parallelization of the loop.

- Schedule length: represents the total amount of cycles required for executing a single iteration of the loop. This length depends on the data dependence and the latency of operations. For a given loop, the smaller the achieved length, the smaller the overhead of the prologue/epilogue in the pipeline and the faster the loop execution.

- Stages: number of steps or iterations in the pipeline needed to execute all the operations of the body loop in the steady state (another representation of the loop is
4.3. THE ADRES DESIGN FLOW

Figure 4.14: Kernel-only representation of the loop on a 3X3 array
CHAPTER 4. CASE STUDY 1: IMEC

Figure 4.15: Characteristic metrics of a software-pipelined loop

Figure 4.16: Stages in a software-pipelined loop
shown in Figure 4.16 where we clearly see the number of stages needed to execute the same pattern in the kernel and the total number of cycles of the pipeline. It can be calculated thanks to the equation 4.1

\[ \text{stages} = \text{ceiling}(\text{schedule length}/\text{II}) \]  

(4.1)

- Cycles: represents the total number of cycles to execute the pipelined loop on the architecture. It can be deduced from the II and the schedule length by the equation 4.2. It can also be expressed in function of the number of stages by (eq 4.3).

\[ \text{Cycles} = \text{II} \times (\#\text{iterations} - 1) + \text{schedule length} \]  

(4.2)

\[ \text{Cycles} = \text{II} \times (\#\text{iterations} + \text{stages}) - \text{II} \]  

(4.3)

- Instructions Per Cycle (IPC): this metric gives the average number of instructions executed per cycle. It shows the parallelism achieved by pipelining the loop. The IPC can be calculated in two ways. First, the effective IPC (eq 4.4) is the total number of operations divided by the total number of cycles to execute the loop. Second, the scheduled IPC (eq 4.5) is calculated without the epilogue/prologue and is equal to the number of operations in the loop body divided by the II (it is also equal to the number of operations in the pipeline kernel divided by the number of cycles to execute the kernel).

\[ \text{IPC}_{\text{eff}} = \frac{(\#\text{operations}_{\text{body}}) \times (\#\text{iterations})}{(\#\text{total pipeline's cycles})} \]  

(4.4)

\[ \text{IPC}_{\text{sch}} = \frac{\#\text{operations}_{\text{body}}}{\text{II}} \]  

(4.5)

- Scheduling density (SD) : gives the average amount of computation resources used per cycle during the steady execution of the pipeline. In ADRES, this is given by the equation 4.6.

\[ \text{SD} = \frac{\text{IPC}_{\text{sch}}}{\#\text{FU}s} \]  

(4.6)

These metrics are good indicators of the quality of the scheduling and the efficiency of the software pipelining to parallelize the inner loop. In the next part of this section, we detail the different stages that enable the mapping of a C algorithm on an ADRES template. Indeed, apart from the pure scheduling process, some preparation stage are performed on the algorithm. In this part, we also present the simulators the designer can use to estimate the metrics and evaluate the efficiency of a solution.

\[^{10}\text{“Ceiling” is the next larger integer.}\]
4.3.3 DRESC compiler

We have seen above that the DRESC compiler relies overall on an iterative modulo scheduling algorithm. However, the compiler receives as input a complete C-algorithm in which it has to separate the parts that will be mapped on the VLIW and on the CGA respectively. It implies several analysis and preparation steps that are represented on Figure 4.17 and further explained.

IMPACT framework

The IMPACT framework[10] is a front-end which will produce an intermediate code, called Lcode, which results from code transformations needed before the compiler is able to manage it. This third party framework has been built in the Center for Reliable and High-Performance Computing, University of Illinois to compile C code for multiple-instructions-issue processors by exploiting the instruction level concurrency of the code. It originally comes from the IMPACT-I C compiler, a retargetable compiler with code optimization components. It has been adapted to target the ADRES processor. We see that this front-end needs the STD_PARMS file that contains parameter settings steering all code transformations made by IMPACT. Other files are also required to make application profiling.

Typical actions that are performed by the front-end are:

- profiling of the application
- whole-program inlining: which consists in replacing all functions calls by the body of the callee.
- data flow analysis
- hyperblock formation by inserting predicates

All the next steps will manipulate *.DRE files that are DRESC proprietary files. Moreover, we also see that the different phases of the compilation take the information stored in two main files.

- dresc_params_v3 : in this file, the user can specify the option related to the tool chain. A default file dresc_params_v3.default exists that contains all the possible options and that is used in case no option are specified by the user. Some of these options will be cited for lower steps.
- dresc_arch : contains the specification of the functions and loops that will be mapped on the ADRES processor, with the scheduling options for these functions/loops.

Lcode to DRE convertor

This first step converts the *.HS file type generated by the IMPACT framework to a *.DRE file manageable by DRESC. At this stage, some adaptations of the code are also made based on information given in the "dresc_params_v3" file. These transformations are typically:
Figure 4.17: Design steps of the DRESC compiler
• if specified by the user, conversion of all 32-bit multiplications (mul and mul_u) into
16-bit multiplications (mul_16s and mul_16u)

• translation of sign extension and zero extension instructions into alternatives by
means of masks and shift operations.

CGA loop preparation

Once we dispose of the *.DRE files, the pieces of code can be considered separately for
each ADRES part. First, the CGA is considered. In the "dresc_params_v3", the user can
specify the functions in the program that have to be taken into account when preparing the
code for the CGA. This is indicated by putting the prefix DRESC_ before the function
name that the user wants to consider. Before the inner loops of the function can be
scheduled on an ADRES array, code transformations and preparations still need to be
performed. Indeed, inner loops have to be extracted from the other part of the code,
predicated operations must replace branches and be normalized to be interpretable by the
compiler, loop headers have to be added to start and stop the loop execution, variables
(live-in, live-out, constants) have to be detected to be mapped on right components...
At the end of this step, a "workplace.precga" directory is created and contains the function.DRE file. This file gives a DDG representation of the algorithm by block, one inner
loop being a block for example.

CGA loop mapping

The mapper will map a data flow representation of the code onto a "data routing rep-
resentation" of the architecture called the module routing resource graph (MRRG). The
latter is a space-time graph obtained based on modulo reservation table and on a routing
resource graph. The mapping algorithm is in fact based on a congestion-negotiation and
simulated annealing principles as we can find in FPGA placement and routing algorithms.
As we have explained in Section 4.3.2, the algorithm performs iterative modulo scheduling.
A starting II is computed first based on the data dependences and the available resources.
Then the compiler maps the DDG of the loop onto the MRRG. If no valid schedule has
been found, the II is increased and the mapping is tried again. DRESC allocates the
operations in order to minimize the route between two successive operations and allocate
thus the next operation on the closer possible FU. In addition, priority is given to critical
path operations.
The mapping options can be specified by the user in the dresc parms_v3 file. It impacts
the mapping time but also the efficiency of the achieved schedule.

autodetect : if true, the compiler will try to detect itself loops for CGA mapping. Other-
wise, only user-defined loops will be considered.

starting II : user can specify the starting II value from which the scheduling algorithm
will run.
4.3. THE ADRES DESIGN FLOW

random_seed: simulated annealing is based on initial random condition. User can try several random_seed to help finding a valid schedule.

random_seed_range: the range specifies the upper bound in a range of random seeds that will be tried by the compiler.

relax_factor: this value (floating point > or equal to 1.0) is used to extend the time scheduling possibilities of operations in relation with the MRRG graph. This parameter directly impacts the schedule length of the pipelined loop. The smaller the relax factor, the better the performances for a valid schedule. (trade-off between finding a schedule and optimize performances).

relax_factor_step and_range: step and range inform the compiler to try a range of relax_factors during the scheduling.

A workplace.cga directory is built and also contains a .DRE file. This file, similar to the previous one contains the DDG of the code but also more information about the scheduling of the inner loops.

- II, MII, RecMII, ResMII
- loop_stop_sched_cycle: gives the number of cycles of the last iteration of the loop. Indeed, it could be smaller than the length of the other iterations, as some extra operations can be needed to switch from one iteration to another, which is not the case when the loop ends.
- num_stages: gives the number of stages for the loop pipeline
- random_seed: gives the achieved random_seed
- relax_factor: gives the achieved relax_factor

The file also gives the scheduling of the operations on the FUs for each operation of the loop and the register files, constant memories and transition nodes where the data goes to and comes from.

VLIW code selection

Once the required inner loops have been mapped on the CGA, the compiler will take the generated .DRE file to replace all the instructions related to the VLIW part of the code by instructions supported by ADRES. Typically, the selector replaces conditional branches by a comparison and a predicated unconditional branch, verify the operand types (immediates vs register operands), merge successive control blocks to improve the scheduler freedom,... Options can also be specified by the user for this code selection in the file dresc_parms_v3 to make more or less changes. Then an updated .DRE file is generated in the workplace.codeselect directory.
CHAPTER 4. CASE STUDY 1: IMEC

### VLIW code scheduling

This step performs the scheduling of the different control blocks (CB) on the VLIW. It first builds a data dependency graph of each CB, builds a queue containing all ready-to-schedule operations and progressively schedules the operations using some priority function. A cycle \( C \) is incremented until a schedule is found for the current operation. The directory `workplace.sched` gives the resulting mapping information of the complete code, including the VLIW part.

### Register allocation

Once the code has been mapped on ADRES, all virtual registers have to be assigned to a limited amount of hardware registers. At least one integer register file and one predicate register file needs to be present in the architecture. Options can also be specified in the `dresc_parms.v3` to deal with the prologue/epilogue, the rotating registers, the data forwarding,... The file `.DRE` in the `workplace.ra` directory is completed by the right register numbers.

### Assembly and linking

Finally, DRESC produces a binary output by assembling the DRE files and linking them. Different linker properties can also be specified in the `dresc_parms.v3` file. Four hexadecimal files are generated to be used in STRL simulation (see next section).

- `a.out.MAP`: contains information for debugging purposes. It gives the addresses of the code section start and end.
- `a.out.DATA`: this file gives the contents of the statically-allocated data section of the linked program.
- `a.out.TEXT`: this file contains the contents of the VLIW instruction memory. Each line represents one instruction bundle.
- `a.out.CGA_CONFIG`: it describes the layout of the array mode configuration files.

Binary files are also generated for VHDL simulations. These contains extra information like the configuration memory files and have a `.bin` extension.

### 4.3.4 Simulations and statistics

Three kind of simulators can be used at different steps in the tool chain. These simulators enable to check functionality or estimate performances to evaluate the efficiency of the mapping.

**Compiled code simulator**

This simulator can be used at each step in DRESC to simulate the *.DRE output files. These files are converted into C++ code taking into account the architecture files and
4.3. THE ADRES DESIGN FLOW

creates a UNIX executable. The results of the simulation are stored in a *stat.out* file containing the total instruction count, the *noMOV* instruction counts (in which routing instructions in CGA mode have been removed), the cycles, the IPC, the reads/writes, the load and store instruction count. This information will be important to analyse the efficiency of the mapping and the choices made to describe the C algorithm of the application but also the template of the architecture. We will explain in Section 4.3.5 how these results will influence the decisions made during iteration step.

**ADRES Virtual Machine**

This virtual machine is able to execute *.DRE* files. The advantage over the CC simulator is that the virtual machine can be used into a larger-platform-simulation environment that contains several AVM in order to integrate several components together. The DRE files are converted in a byte-code format that can then be executed. The simulation is based on the dresc_arch file and on scheduling information that are converted in object files.

**Esterel simulator**

This simulator is used to get an accurate estimation of the energy consumption of the ADRES platform. It uses files produced by the assembly and linking step of DRESC and the architecture description of the processor on which the simulator is built. This simulator is much slower than the others. It is typically used to generate traces for hardware verifications.

Besides these simulators, another tool is available to understand the mapping performed by DRESC and help in exploring solutions.

**Schedule Viewer**

The schedule viewer is a graphical representation of the CGA of the ADRES processor giving the way DRESC has performed the scheduling of the operations for one iteration of a loop. An example is shown in Figure 4.18 for a 2X2. Different information are given in this viewer: the types of operations are discerned thanks to colors. The *II* (which graphically corresponds to the number of stacked layers) and the *stages* are also displayed. This representation is very convenient to have a quick idea on the way resources are allocated to the platform.

In our work, we only used the compiled code simulator which gives fast results and all needed metrics. The schedule viewer is a practical tool that enables an easy visualization of the mapping. It is particularly suited for mapping interpretation and comparisons with Nessie.

Typically, the designer uses the information stored in the stat.out file and the schedule viewer to understand why a solution is not efficient and adapt the C code structure of
the application algorithm. Iterations are often needed as we have explained earlier and come-back to the precompiler step is mandatory. We explain in the next section how the user can take advantage of the information given by DRESC and the simulators and what are its degrees of freedom.

### 4.3.5 Precompiler decisions

The efficiency of the execution of an application on the ADRES processor is directly dependent on the structure of the C code produced at the precompiler stage. In this design step of the ADRES flow, the exploration of solution is performed manually and sequentially:

- first, an ADRES template is chosen based on previous design experiences.

- second, the application is described with a C code, based on a previous Matlab application description and partition. The structure of the C code is based on implicit models and is changed according to lower metrics estimations. As we further explain, this code definition is not simple and the impact on the final performances is not guaranteed.
As only inner loops are mapped on the CGA, the structure of the loops in the algorithm will impact the amount of code that will be executed on the VLIW and on the CGA respectively. As the CGA is the accelerator part of the processor, the trend is to map most of the code on the array. However, the efficiency of the mapping also depends on the structure of the inner loops themselves and on the structure of the CGA.

The structure of the algorithm offers several degrees of freedom.

- the structure of the code: the number of inner loops, the number of nested loops, the proportion of control in the code...
- the number of operations in a loop
- the number of iterations of a loop
- the data dependency in a loop
- the memory accesses, the types of variables

From the side of the architecture, the degrees of freedom are:

- the number of FU’s
- the instruction set available for each FU - the number of load/store slots.
- the number of RF
- the topology of RF
- the interconnexion scheme

All these degrees of freedom have an impact on the scheduling metrics.

- prologue/epilogue: the bigger the data dependency (intra and inter iterations), the bigger the epilogue and prologue, the worst the CGA mapping. Increasing the number of iterations can then hide the epilogue and prologue penalty and increase the mapping efficiency.

- SD: a satisfying scheduling should be above 70%. The number of operations in the loop is also important to choose the correct instance of ADRES regarding the number of FU’s. If there are not enough operations compared to the available resources, the scheduling density will be low. In this case, it is interesting to look at the II which will give information on the added cycles compared to ResII and RecII, due to intermediate registers or predicate operation. If increasing the number of operations may improve the SD, we have to take into account that it also increases the resource constraint, then the MII and II. In some cases, the impact on the SD could be undefined.
CHAPTER 4. CASE STUDY 1: IMEC

- II: if the ResII is high, it implies that a lot of resources are already used in the CGA, so that the number of operations in the loop body compared to the number of FU is too important. Often ResII is bigger than RecII (low data dependencies). If there is a big difference between MII and final II, it can come from conditional operations inside the inner loop. Then, it could be interesting to remove these operations from the loop body.

- Relative cycles: the ratio of the number of cycles between VLIW and CGA shows the relative use of the ADRES processor and so the parallelization of the algorithm. To maximize performance on ADRES (minimize total number of cycles and energy), the designer must take care of the execution of most of the code on the CGA with the smaller II.

- IPC: the higher the effective IPC, the better the mapping. To increase it, the algorithm has to be accelerated on the CGA which implies that more loops have to be inner loops. Moreover, these loops have to have a sufficiently high number of iterations to hide the epilogue and prologue which is a penalty to an efficient parallelization of the algorithm. Increasing the IPC shows that less cycles are needed to execute the same number of instructions. This metric is thus a good indicator for the code extracted parallelism. The ratio of EffIPC on SchIPC shows the mapping efficiency. The nearest to 1, the better it is. The total effective IPC is also important to know if the algorithm has been correctly accelerated on the CGA. Increasing the IPC is possible by increasing the number of inner loops in the code, but also, by increasing the body vs epilogue/prologue ratio to improve the average number of executed operations on the CGA. This can be achieved by increasing the number of iterations for the concerned inner loop.

To change the structure of the code without changing its functionality, so-called loop transformations are applied. Well-known transformations are presented below with the impact they could have on the loop pipeline.

**Loop unrolling** consists in developing a loop by duplicating the operations in the body loop according to the unrolling factor and reducing the number of iterations by this same factor. The data dependency will probably increase and so will do the II. The effIPC on the CGA will decrease because the epilogue/prologue overhead will increase.

**Loop merging** consists in creating a new inner loop by merging together two loops. This transformation is a way to increase both the number of operations and the number of iterations. This transformation enables the improvement of the SD and the effIPC on the CGA.

**Loop coalescing** transforms nested loops into one bigger inner loop. It thus increases the number of operations in the new inner loop and improves the SD (more iterations) and the effIPC (more code accelerated).
4.3. THE ADRES DESIGN FLOW

Loop outlining outlines nested loops. It increases the number of inner loops and the number of iterations in the new loops which improves the code ratio accelerated by the CGA and the effIPC by reducing the prologue/epilogue overhead.

IF extraction while and if conditions lead to the addition of extra predicate operations when scheduling a loop on ADRES. Such a condition located inside a loop will add cycles for each iteration of the body loop and thus increase also the II. To avoid that, the designer must take care of the way the algorithm is coded, and if necessary, remove the if condition outside the loop. This will improve the SD and the effIPC.

These different transformations are done step by step manually based on results estimated a posteriori by DRESC. In practice, successive iterations are required to change the chosen solution at this higher level and a lot of time is spent to perform these changes (hours or days depending on the complexity of the algorithm).

4.3.6 Conclusion

Through the description of the two ADRES design steps (precompiler and DRESC compiler), we have highlighted that the design flow is based on an a posteriori estimation of the performances of the solution and that several costly iterations are needed to converge to a satisfying solution. This is due to:

- a manual and qualitative exploration of solutions at the precompiler stage where the architecture of the processor and the structure of the algorithm are not considered jointly, but sequentially, based on an implicit model of the performance. The large number of degrees of freedom, from the viewpoints of both the application and the architecture, creates a huge solution space and forces the designer to restrict the exploration of solutions, which is done essentially on the application structure.

- a mono-criterion slow exploration at the DRESC level where only time-related criteria are estimated. As other criteria, like the energy consumption, will be estimated later in the flow, additional iterations are expected.

Our tool NESSIE is precisely dedicated to the a priori estimation of performances thanks to a fast and multicriteria exploration of both the application and the platform, in order to reduce the design time and converge to better solutions by covering more solutions. This case study is therefore a very good candidate to validate our tool.

In the next section, we present the choices that have been made to model the case study into NESSIE to put our framework to the test. We also explain the methodology that has been used for the validation, before presenting the results in Section 4.6.
4.4 Formalization of the flow

This section is divided in two parts: first we present what we will model into NESSIE, second, we explain how we get the models to feed NESSIE with inputs and to analyze the outputs.

Figure 4.19 shows schematically the ADRES flow that we have described and analyzed in the previous section. The characteristics of this flow are summarized in this top-down representation:

- in the first design step, the designer decides on C code variants that are produced iteratively based on implicit models of the performances. Quantitative estimation of the criteria is performed at the next design step.

- in the second design step, DRESC takes one variant of the application (C algorithm) and one variant of the architecture (XML description of an ADRES instance) and
automatically performs the mapping to generate the system as binaries. At this level, simulations are done to estimate a critical criterion, the cycles, and scheduling metrics (IPC, SD, ...).

- both design steps are time consuming and several iterations are performed to find a satisfying solution.

To improve this design process, we propose to estimate a priori, i.e. before the second design step, multiple criteria for several solutions, to give DRESC the "right the first time" application/architecture couple and avoid further costly iterations. NESSIE acts thus as a prediction tools that takes as input models of several C algorithms and several ADRES architectures, performs a multicriteria estimation, and preselects the best solution(s) amongst all the variants to guide the designer choice for the DRESC compilation. However, we have to keep in mind the way the NESSIE framework has been defined and the resulting modeling guidelines. To model a system in our tool, we have to define:

- the application and the architecture at the same level of abstraction (AL). Indeed, the tool must be able to perform the mapping of SW primitives on HW primitives. The number of AL must also be defined.
- the criteria and the parameters of the systems, and the relations that will allow NESSIE to estimate the criteria.
- the degrees of freedom that will be considered and that will form the design space exploration.

In compliance with these guidelines, and according to the available information in the IMEC flow, we have decided to model the system at two AL. Moreover, we will focus on the modeling of the CGA of the ADRES processor. Indeed :

- the computational-intensive part of the code, which is the core of the application and the critical part, is executed on the CGA. The VLIW is dedicated essentially to the control.
- getting the information required to model the CGA and its application counterpart, and define their compatibility, is straight-forward, compared to the other part of the system.

We illustrate in Figure 4.20 the two AL that we will consider. They are detailed below.

**AL 0** the first level of abstraction is required in NESSIE and defines the system as a black box, which is the unique primitive of the level. For our case study, this black box is a generic CGA from the side of the architecture. It is an inner loop from the side of the application, as only inner loops are mapped on the CGA.
AL 1 at the lower AL, we will define the structures that compose the CGA and the inner loop. From the side of the CGA, the lower structure are defined thanks to the components presented in the previous section (FU, RF, CM, TRN, connections). From the side of the loop, we need a data dependence representation of tasks compatible with the HW components. These tasks are the operations composing the instruction set of the processors (e.g. additions, multiplications, loads, shifts, ...) and represent the application at a lower description level than the C algorithm.

To validate our tool based on this case study, we will follow the methodology presented below.

4.4.1 Validation methodology

The validation of our tool is composed of several steps.

1. we will discuss, in Section 4.5, the capability to model the case study defined above in NESSIE. We will see that, despite some limitations of the framework, the modeling is feasible.

2. we will test the prediction capability of NESSIE. Therefore, we will produce results for several criteria and loops/CQA variants in our tool. The same solutions will be compiled in DRESC and based on the compiler metrics, we will compare the trends of the results produced by both tools. This is the purpose of Section 4.6.

3. finally, we will evaluate the modeling and simulation time spent in NESSIE and discuss the design time in the two flows: the first being the classical ADRES flow, the second being composed of the NESSIE prediction tool upstream from the DRESC compiler. This is presented in Section 4.7.
4.4. FORMALIZATION OF THE FLOW

As a support of these analyses, we introduce in the following part, the framework that we have used to enable or facilitate the generation of the XML inputs of NESSIE and the analysis and interpretation of its XML outputs in the purpose of validation.

4.4.2 Interface between DRESC flow and NESSIE

Starting from the information available in the IMEC flow (ADRES XML description file, C algorithm of an application, latencies of operations, ...), we have to produce the XML input files of NESSIE (simulation, behaviors, allocation and routing weights) and be able to take profit of its XML output files (criteria results, activity report, timeline).

Two aspects are considered: the generation of the NESSIE inputs based on the information available in the classical flow and the interpretation of the NESSIE results to be able to give reliable information to the designer for refinement in DRESC.

The former aspect consists in easily and quickly translate C code into Petri Net and extract rapidly architecture information from the dresc_arch file to the netlist and HW structure.

In the latter aspect we must easily take advantage of the timeline information given by NESSIE to understand the mapping performed and interpret results, but also of course the criteria results and the activity report to discuss the resource usage and compare it with DRESC results.

Input generation

The application: starting from a C code representing the application functionality, we need to get a lower representation of the code, composed of the instructions executable by FUs, and which gives the data dependence between the operations (DDG). Based on such a DDG, we must create the corresponding Petri Net.

To get the graph, we have two alternatives: first manually extract a DDG from the C code, second use a tool that automatically extracts the DDG.

The former possibility is really time consuming if the user is not sufficiently familiar with the DRESC compiler and its instruction set.

The latter possibility is performed during the first stages in the compiler. Indeed, the .DRE file produced by the pre-cga stage already contains a data dependence graph of the operations for the inner loops. This step is not time expensive (few seconds). If this makes us dependent on the tool, it has the big advantage to accelerate the modeling step. We thus have chosen this alternative to extract easily the DDG of inner loops.

Once we dispose of this DDG, expressed in the DRESC format, we still need to convert it into places and transition with judicious primitives and make a clear concordance between the operations names and the ID’s used in NESSIE. During the work, we did not dispose of a functional and automatic way to produce the Petri Network. They thus have been drawn by hand before their description in the XML simulation file of our tool. This process must be done for each variant of the application we want to explore. In Section 4.7, we discuss the time consumed for this task for the case study.
The architecture: contrarily to the application, the architecture information is well detailed at the desired level of abstraction in the dresc_arch file. Nevertheless, extracting the relevant information from this file can be error prone and tedious (a basic architecture, like a 2x2 with shared RF, implies an XML dresc file of 750 lines what is already difficult to manage). This is why we have created a small graphical interface which extracts, sorts and presents the information such that the primitives and their interconnection scheme can be easily understood.

The interface has been written in TCL/TK, a scripting language that has been extended with graphical functionalities and that is often used in design tools (e.g. Mentor Graphics\textsuperscript{11}). As it is the case for the application, the primitives that must be defined in Nessie are identified thanks to IDs. An intermediate representation is thus useful to make the concordance between the components and the ID’s. This phase has been done manually, as for the Petri Networks.

Regarding these remarks, we already expect that an exhaustive exploration of the design space is not possible. This discussion is done in Section 4.7.3, based on the simulation results obtained in NESSIE.

Output analysis

The three output files produced by Nessie are devoted to different goals.

- criteria results: this file will contain the multiple criteria for all variants explored by NESSIE. The results are easily manageable from this file and can be rapidly plotted.

- activity report: this file will allow us to understand how NESSIE has allocated the tasks to the hardware resources. This information will be of use when discussing the validity of the results produced by NESSIE. During the thesis, a user interface has been developed to manage these files and present graphically, via colors bars, the relative resource usage.

- timeline: this file is of importance to understand how NESSIE has managed the execution of the loop on the CGA. The discussion of the relevancy of the results will notably depend on the comparison of the two mapping strategies used respectively in NESSIE and in DRESC. However, such file grows rapidly even for small application/architecture structures (4500 lines). To deal with the big amount of information stored in the timeline file, we have developed a small interface for visualization support that represent the events in a table, sorted by time step and removing all tags and extra information. Moreover, we have added information in the timeline to be able to trace the data token routed inside the platform. This is an important information to understand bottlenecks, deadlocks and see the data flow into the architecture.

\textsuperscript{11}A view of the produced window is shown in the Appendix A.
Having clarified the validation framework, we will now detail the modeling of our case study in NESSIE, i.e. the primitives of the SW and HW, the criteria, the behaviors and the allocation and routing weights. This will be illustrated on the concrete example of a matrix multiplication algorithm, for which different variants of loops will be defined and several structures of CGA will be explored.

4.5 Modeling of the case study in NESSIE, applied on a matrix multiplication

This section is divided in three parts. First, we discuss the modeling capability and limitations of NESSIE for the case study (inner loops vs CGA) regarding the criteria, the different ADRES components, the DRESC mapping policy and the loops characteristics.
Second, we illustrate this discussion on the matrix multiplication algorithm and detail the choices made to describe the Petri Networks. In this part, we also present the application scenarios we have chosen for the exploration of solution.

Third, we present the criteria and the CGA variants that have been chosen for the simulations in NESSIE and the four CGA models that will be tested.

4.5.1 Modeling into Nessie

To validate our tool, we will model several variants of for loops and CGA. These variants will be simulated based on multiple criteria. The definition of these criteria depend on the availability of cost models or the possibility to create these models. But it also depends on the ability to compare the results estimated in NESSIE with results in the ADRES flow, what is related to the type of simulator used in this flow.

The criteria estimation

The critical criteria for our case study are:

- the cycles: the user manual of DRESC gives information about the operation latencies, the registers accesses and the TRN nodes of the ADRES processor. Based on these values, we are able to feed the behavior.xml files for each component of the architecture. Moreover, DRESC simulates this criterion and we are thus able to compare the cycles in both tools.

- the energy consumption: on the contrary, we have not been able to get the models or typical values to compute the energy consumption of the platform. In addition, the simulation of this criterion is performed later in the IMEC flow and we have not studied this part in our work. This criterion will thus not be considered in our model.

- the area: examples of components area have been obtained for the CGA. Although no simulator computes these areas in the design steps we consider, we are able to feed NESSIE with realistic values. We have decided to introduce this criterion in the NESSIE simulation to illustrate the design trade-offs that our case study offers.

For the purpose of validation, we have decided to include, in addition to the two criteria (cycles and area) that we will evaluate in NESSIE, metrics that are estimated in DRESC. Indeed, these metrics give important information on the way the scheduling of a loop has been done, its efficiency regarding the execution parallelism and the resource usage. However, as NESSIE does not perform software pipelining, as it is the case in DRESC, we cannot calculate automatically all of these metrics. This is the case for the $II$, the $SD$, the scheduling $IPC$ and the $length$ which are deduced based on the pipeline kernel, which cannot be detected by NESSIE, and with the assumption that each iteration of the loop will be mapped the same way on the platform, which is not the case in NESSIE.

We have consider two metrics:
4.5. MODELING OF THE MATRIX MULTIPLICATION IN NESSIE

- effective IPC: to calculate the effIPC, we just need to know the total number of cycles and of operations.

- effective density (ED): we have defined this metric instead of the scheduling density (SD) as the effIPC divided by the total number of FUs. This gives the mean resource occupation for the entire pipelined loop, taking into account the prologue and epilogue.

In addition to these considerations, we must take the remarks given in Chapter 3, concerning the memory management and the mapping policy, into account and evaluate what are the consequences on the modeling of our case study.

Specificities

- **Memorization management**: Nessie does not perform automatic allocation and management of external memorization. When there are several memories or register files to manage, as it is the case in ADRES, the modeling of the memorization is not trivial. In ADRES, two kinds of components are able to store data. The external memory, which exclusively stores live-in and live-out variables. The register files (shared or distributed) that store temporarily data: either data that will be used several times (the RF keeps a copy of the data) or data that have to be sent to the consumer FU that is not connected to the producer FU. In the last case, the producer sends the output data to its potential local RF.

  The storage of the output data thus depends on the kind of data, but also on the place where it is produced and consumed. To force the memorization, the user has to know where the operations will be mapped, or to make the storage independent of the mapping. As ADRES is a wide homogeneous platform, we cannot determine a priori where data will have to be memorized.

  We must also notice that currently there is no notion of limited memory capacity in NESSIE. Obviously, the RF and the external memories have a limited size. Their dimensions depend on the application and the amount of data that have to be stored.

- **Mapping policy**: In its current version, Nessie performs the allocation independently of the routing. A consequence is that Nessie will not automatically choose HW blocks candidates for allocation that are close to the producer. It can be a problem for the memorization of data. Moreover, as deadlock can appear when no route is found, solutions can be invalid. Finally, we have to take into account the fact that Nessie chooses the first available HW block which minimizes the criteria. As the components are generally quite identical in the CGA (depending on the ISA repartition), it can lead to unused HW blocks in the architecture. We will see later if the mapping policy is relevant regarding the case study.

- **Allocation and routing weights**: By changing the allocation weight in the corresponding .xml file, we can influence the allocation decision. If invalid solutions are
identified, we could also adapt the routing weight to change the route choice. However, due to the high homogeneity of the ADRES platform, we are quite limited with the weight possibilities. Indeed, the allocation weight can exclusively be expressed based on inputs parameters and output criteria which are equivalent for all the FU’s. Moreover, this weight is common to all the levels of abstraction what can be a limitation as the allocation priority could vary from one AL to another.

In DRESC, the scheduling and allocation is based on an optimization of the number of cycles which takes into account the available resources and their relative location i.e the route. The route is directly dependent on the location of the producers and consumers. The scheduling of the local register is thus also a consequence of the scheduling and allocation of the operations. The user is not able to know where and when data will be stored in local registers. This implies that Nessie should be able to dynamically determine if a data has to be stored in registers or not. Otherwise, forcing the memorisation of data could lead to non realistic mapping which should be non relevant regarding the real flow.

Below we present the different models we have foreseen to represent the CGA.

**The architecture**

To model architecture structures, we need first to create all the primitives for the two levels of abstraction we have defined in Section 4.4. For each HW primitive, we have to associate relevant core and ports states and behaviours, but also the compatibility list of the SW primitives that will be executed.

The common properties of the blocks are:

- all the blocks have an idle mode. This mode is the default mode when the block does not compute, memorize or transmit data. The behaviour associated to this mode gives the models to evaluate the criteria, i.e. the area, the cycles, the effective IPC and the effective density. The area will depend on the primitive, the cycles is 0 as it thus not add time for the execution of the application.

- we do not consider the sleeping mode in this architecture.

**FUNCTIONAL UNITS:** FU s are computational primitives. As such, their main state will be the computing state, associated to the SW primitives they are able to execute. This will be defined depending on the application and depending on the template possibilities the user wants to explore (i.e. the way the instruction set is distributed on the CGA). The FU area depends on the operation set defined in the ALU. The cycles are given in the DRESC manual for each operation type. The user can decide to add transmission capabilities to FU. Indeed, special operations, called MOV operation are devoted to transmit input data to the output of the FU in order to route them to other units. In that case, in NESSIE, the transmitting state will be enabled and a behaviour will be defined. This
operation takes 1 cycle\textsuperscript{12}. Finally, when a data has been produced by the FU, it is normally sent to a local RF, to the global VLIW RF or to the output buffer, depending on the future destination of the data. It is thus not memorized internally. However, we have explained in Section 4.3.1 that the FU is able to begin the execution of a new operation each cycle what means that the FU is pipelined and manages the successive execution of the operation internally. In Nessie, if a HW block has not memorization state, it is not allowed to execute a new operation until the previous produced token has been sent to another block. If the primitive has memorization capability, it can start a new computation if the previous data token has been produced, which can be performed more than 1 cycle latter. The second case seems the more close to the real behaviour. However, as normally the FU does not memorize the produced data, the cycles associated to the memorization state would be 0.

\textbf{OUTPUT BUFFERS:} the output buffer is a transition node\textsuperscript{13} dedicated to route data from one FU to another. Normally, a data produced by the FU is sent directly to the output buffer (or to an RF) which latches the data 1 cycle later. Two models are possible. First, we model the buffer as a memory on which we will store the data until it is consumed by the next block. Second, we model it as a transmission. In that case, the data will be transmitted only when the consumer will be ready to compute it. If the next FU has memorization capability, the data can be sent directly. Otherwise, the data will block the previous FU until the next one is freed. In the first model, we know that the explicit memorization on the buffer is not convenient and forces the user to predict when the memorization has to be performed what is not possible in practice.

\textbf{REGISTER FILES:} Register files should typically be memorization primitives. As explained early, to model it, we need to explicitly define when the memorization has to be performed. There are two limiting factors. First if there are more than one RF, Nessie will not always choose the relevant one which should be the RF directly connected to the producer FU. Second, logically, we could define some data that have to be stored in RF. It is the case of temporarily data will be loaded several times for execution. For the other data, the storage in RF will depend on the consumer and thus on the relevant route to choose. In this case, a way to avoid the explicit allocation of the data is to consider the RF as a route node and thus as a primitive with transmitting state. In that way, Nessie will choose logically the better route between two FUs that could pass through the output buffer or the closest RF. We have decided to consider two models. First the RF has memorization state, which implies a computation state compatible with a pseudo-operation, and a transmitting state. For both the computation and transmitting states, the cycle is 1, what correspond to the latency added by the RF. Cycle 0 is associated to memorization state as the cost has already be included in the computation state. Second, the primitive is only a transmission with latency equal 1. It implies that all the data will be managed by the FUs. In other words, data that will be used several times will also be

\textsuperscript{12}The latencies are given in the DRESC manual available at IMEC.
\textsuperscript{13}Information is based on the DRESC manual available at IMEC.
sent by the same FU several times. In case of FU that have no memorization capability, we could rapidly have deadlocks because FU are blocked until token have all be sent. We will see later, when discussing the results in Section 4.6, that it can lead to some overhead during mapping. Finally, the fact that there is no limited size for the memorization can be neglected. Indeed, we will suppose that there is enough memory to store data.

**CONSTANTS:** Constant memories are components that contain the immediate operands and that must be accessible in reading mode. To force the access to a given constant memory, we can model it as explicit memories. This implies that a pseudo-operation has to be defined from the side of the Petri Networks to be able to store the immediate operands into the constant memories before being able to use the data. The memorization state would not consume any cycle. So it is also for the computation state associated with the pseudo-operation. However, as different constant memories are connected to dedicated FU’s, the location of the immediate operand must be consistent with the location of the consumer operation (which depends on the allocation choice), what cannot be determine a priori before the mapping.

One way to resolve this problem is to consider a big constant memory which cumulates the area of all the constant memories and is connected to all the FUs.

**MUXES:** As the buffers, the muxes are routing nodes. They are used in ADRES when several wires feed a same input of a register files or FU. This component adds no delay but adds some area. In our model, it is not essential for modeling it explicitly as the primitives (FU, RF,...) can have several input ports that manage the numerous inputs coming from different wires. However, if this primitive is not created, the added area has to be included in the connected primitive (eg : in the FU for the muxes managing the FU inputs). Moreover, as we do not represent the control flow in Nessie, we do not need to take care of the muxes.

**CONNECTIONS:** In the dresc_arch.xml file, the user must define the connections between each block. This will represent the wires in ADRES. The more the wires, the more the space area used in the processor. Wires also add delays. However, this information is not easily available and the overhead compared to the other components can reasonably be neglected. If necessary, we can add latencies and bandwidth to the input and output ports of the interconnected components. If we do not create dedicated primitives for the connections, these components are equivalent to the logical links in Nessie.

Now that we have presented the models possibilities for each primitive, we summarize the 4 models we have decided to test in Nessie. For each model of CGA, the buffer has be modeled as a pure transmission. The constant memories have been modeled as memorization primitives with computation state. The simulation models vary thus by the FU and RF primitives and the logical links that will be considered in the structure. We
4.5. MODELING OF THE MATRIX MULTIPLICATION IN NESSIE

<table>
<thead>
<tr>
<th>Component</th>
<th>Computation</th>
<th>Transmission</th>
<th>Memorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RF(*)</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Buffer</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>External memory</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CM</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Wires</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Primitives states of the Model 1

<table>
<thead>
<tr>
<th>Component</th>
<th>Computation</th>
<th>Transmission</th>
<th>Memorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RF</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>External memory</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CM</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Wires</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Primitives states of the Model 2

have considered that FU and RF have always transmission capability to be able to model respectively, the MOV operations and the routing through RF.

1. the Model 1 should be the closer to the real behavior. It allows memorization in FU and RF. This model can be used only if there is one RF. Otherwise, the mapping would be not realistic.

2. the Model 2 removes memorization capability in RF. It thus report the data management totally on the FUs. The advantage is that this model can be used to represent any kind of RF topology.

3. in the Model 3, the FU are not able to memorize data. The RF have memorization capability. The same remark than the model1 is thus applied here: this model will not be used for multi-RF architectures. The risk with that model is to have deadlocks that could be avoided with the two previous models as FU are blocked until they have sent their produced data.

4. in the Model 4, we also remove the memorization capability from the RF, which makes this model usable in multiple RF architectures. However, as for the model3, we could have deadlock rapidly.

**The algorithm**

The number of different primitives will of course depend on the algorithm defined in C. However, three kinds of primitives will be defined.
### Table 4.3: Primitives states of the Model 3

<table>
<thead>
<tr>
<th>Component</th>
<th>Computation</th>
<th>Transmission</th>
<th>Memorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RF (*)</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Buffer</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>External memory</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CM</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Wires</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### Table 4.4: Primitives states of the Model 4

<table>
<thead>
<tr>
<th>Component</th>
<th>Computation</th>
<th>Transmission</th>
<th>Memorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RF</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Buffer</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>External memory</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CM</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Wires</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

- the primitives representing the instructions executed by the FUs.
- the primitives representing the immediate operation stored in Constant memories
- the primitives corresponding to pseudo-operation to store data in RF
- the primitives corresponding to pseudo-operation to store data in VLIW RF

In the Petri Network, dummy nodes will also be used if data have to be sent at several different operations (ie places).

The structure of the Petri Network must be chosen in order to model a finite loop or to enable a relevant estimation of the criteria and to closely represent the application behavior.

To model a loop, three strategies are possible:

1. model and schedule one iteration of the loop and deduce the criteria for the entire loop, as it is the case in DRESC.
2. describe the structure of the loop body and include a feedback to perform the iterations. This is the more succinct representation.
3. develop entirely the loop. This can be very tedious for big loops if no automatic tool is developed (see section 4.7 for details).

The first possibility has been rejected. Indeed, to calculate the cycles, the $ED$ and the $effIPC$, NESSIE must be able to calculate automatically the length of the body loop, the
II and detect the kernel. It is obviously not possible as our tool is based on a simple mapping policy which does not perform modulo scheduling.

In the other cases, an initial state has to be identified what means define the places that will be the outputs of the starting transition. Typically, an inner loop will use data coming from the VLIW and the constant memories. To model it, we can use the pseudo-operations that represent the memorization of these data respectively for the ones stored in the VLIW RF and the ones stored in the constant memories.

The use of dummy nodes can be identified too. Indeed, a data that will be used several times by different operations will be stored either in a RF (HW models 1 and 3) or in FUs (models 2 and 4). Dummy nodes will thus be included after places that correspond to operations that will send a token to several places. It will correspond to pseudo-operations mapped on RF in one side, and operations that will be executed by FUs in the other side. Moreover, immediate operand could be used several times, for each iteration. This also implies that dummy nodes need to be added after pseudo-operations corresponding to the storage of data that will produce a token used by several places.

Now, we will detail what the differences in the structure of the Petri Network are when using model 1 or model 2. Figure 4.22 shows how the Petri Network can be defined.

Feedback loop The way the feedback is defined has an impact on the way the Petri Network will be executed. In Figure 4.22(a), we have produced in fact an infinite loop. Indeed, the first iteration is composed of the place \( M_1 \) which fires the transition \( T_1 \). This transition produces a token that enables the execution of the place \( + \). This place fires the transition \( T_2 \) and the place \( M_2 \) can be executed. When the transition \( T_3 \) is fired, the feedback is enabled and the place \( + \) can be executed once again firing the transition \( T_2 \), etc... To define a finite loop, we have to use another structure, as the one shown in Figure (b). Indeed, after the starting transition \( T_0 \), the places \( M_1 \) and \( M_2 \) are executed. The transition \( T_1 \) is fired. The transition \( T_2 \), however, cannot be fired as it needs the previous execution of the place \( M_3 \) to get the second token. This place \( M_3 \) can be executed after the \( T_1 \) firing and then produce the required token. The transition \( T_2 \) is then enabled and the place \( + \) can be executed. When the \( T_3 \) is fired, the place \( M_3 \) can once more be executed. At the same time, the place \( M_2 \) can be executed 9 times as indicated by the incoming edge. That mean that the transition \( T_2 \) can be fired 9 times and that the addition will be executed 9 times while the place \( M_3 \) will be executed 10 times. In such structure, the number of iterations and the end of the Petri Net are thus defined thanks to edges weights. There is no ending transition in the meaning transition with no output.

Developed loop In such structure, there is no feedback but an ending transition. Figure 4.22(c) gives an example showing the developed loop of Figure (b). In this case, we see the place \( + \) is executed sequentially after the place \( M_3 \). When the transition \( T_4 \) is fired, the place \( M_3' \) corresponding to the second iteration of the loop is executed, and the operations are executed the same way for each iteration. The structure ends with an ending transition. We see that the edges weight do not depend on the iterations of the loop.
Figure 4.22: Models of for loops with Petri Networks in NESSIE
4.5. MODELING OF THE MATRIX MULTIPLICATION IN NESSIE

4.5.2 The matrix multiplication example

As application, we have chosen a function which is at the base of most of the telecommunication applications. It is an algorithm that performs multiplications between two matrices and gives the resulting matrix as output. If this function could seem quite simple, we will see that is implied already an amount of primitives to deal with and that it constitutes a good example on which analysis can be performed and extrapolation of results can be discussed.

The algorithm is shown in Figure 4.23. It is composed of two function : the main where the algorithm starts the execution and initializes the matrices and the DRESC\_mat\_mul which is called in the ”main” and that performs the matrices multiplication following the equation $r = x \times y$. At the line 22, the matrices are defined: these are arrays containing 4096 short elements. In the main function, the matrices x and y are initialized with values. Then, the ”DRESC\_mat\_mul” function is called. The arguments of the function are :

- a pointer to the array x (short)
- the number of rows of the matrix x, r1, which is also the number of rows of the matrix r (integer)
- the number of columns of the matrix x, c1, which is also the number of rows of the matrix y (integer)
- a pointer to the array y (short)
- the number of columns of the matrix y, c2, which is also the number of column of the matrix r
- a pointer to the array r
- the right shift coefficient to apply on the result sum

As we see in the call, the matrices are squared with r1=r2=c1=c2=64. In the function, several variables are initialized.

- i, j, k : counters to sweep up respectively the rows of x, the columns of y and the columns and rows of x and y. (integers)
- sum : stores the intermediate sums to compute the current element of the result matrix r. For each column of y, sum is reset to 0. (integer)
- temp : temporary variable that contains the current element by element multiplication (integer)
- *p1, *p2 : pointers to short elements of the matrices x and y respectively.

If we look at the structure of the code in the function, we see that there are three nested loops. The external one sweeps up the rows of x, the second one sweeps up the columns
```c
void DRESIC_mat_mul(const short *x, int r1, int c1,
                     const short *y, int c2, short *r, int qs)
{
    int i, j, k;
    int sum, tmp;
    short *p1, *p2;

    for (i = 0; i < r1; i++)
        for (j = 0; j < c2; j++)
            {
                sum = 0;
                for (k = 0; k < c1; k++)
                    {
                        p1 = x + k + i * c1;
                        p2 = y + j + k * c2;
                        tmp = *p1 * *p2;
                        sum += tmp;
                    }
                r[j + i*c2] = sum >> qs;
            }
}

short x[4096];
short y[4096];
short r[4096];

int main()
{
    int i, j;

    for (i = 0; i < 64; i++)
        {
            for (j = 0; j < 64; j++)
                {
                    x[j*64 + i] = i + j;
                    y[j*64 + i] = i - j;
                }
        }
    DRESIC_mat_mul(x, 64, 64, y, 64, r, 8);
    return 0;
}
```

Figure 4.23: C algorithm of the matrix multiplication
4.5. MODELING OF THE MATRIX MULTIPLICATION IN NESSIE

Figure 4.24: Illustration of the inner loop of the function, the computational-intensive part of the code

of y and the inner most loop sweeps up the columns of x and rows of y to perform the multiplication between the individual elements of the matrices. As most of the code is in the last loop, we will spend more time on it.

1. at line 13: p1 stores the address of the current element of the matrix x for multiplication. This is computed by adding the relative address of the element in the matrix x with the address where is located the array x.

2. at line 14: p2 stores the address of the current element of the matrix y for multiplication based on the same principle than x.

3. at line 15: then the pointers to the elements stored at the address given by p1 and p2 are multiplied together.

4. at line 16: the intermediate result is stored in the sum variable.

Then the addresses are incremented at the next iteration until all the columns of x and rows of y have been swept up.

As we have explained earlier, we will consider only the parts of the code that will be mapped on the CGA. In this example, we see that the prefix DRESC (see figure 4.23) has been put before the multiplication function. This function will thus be considered when analyzing the code for CGA mapping. In this function, we will in particular focus on the inner loop which is isolated in red in Figure 4.24.

In accordance with the previous section, we have extracted the DDG of this loop thanks to the pre-cga step of DRESC. A schematic representation of the graph is presented in
**Figure 4.25:** DDG of the inner loop of the matrix multiplication function

Figure 4.25. The places represent the operations composing the code of the loop. The edges gives the dependence between the operations. And the weight on the edges give the latency of the operation preceding the edge. For example, the operation $X$ takes 2 cycles to be executed by a FU. Finally, the dashed square places represent the variables, immediate operands or constants that are used by the operations.

Different observations can be made on the loop based on this graph.

1. There are four kinds of operations: multiplication, left-shift, addition and load. These operations have different latencies (see manual).

   - 4 ADD : $dest = Rsrc1 + src2$ with 1 cycle
   - 3 MUL : $dest = (sext_{16\rightarrow32}(Rsrc1_{0\rightarrow15}) \ast sext_{16\rightarrow32}(src2_{0\rightarrow15})_{0\rightarrow31}$ with 2 cycles
   - 4 LSL : $src1 << src2$ with 1 cycle
   - 2 LD_C2 : $dest = sext_{16\rightarrow32}(mem_{16}[src1 + src2])$ with 6 cycles latency\(^{14}\)

\(^{14}\)At the end of the cycles, the result, loaded from the memory, is found at the output of the FU.
4.5. MODELING OF THE MATRIX MULTIPLICATION IN NESSIE

\[ O_1 = i \times c_1 \]
\[ O_2 = \text{LSL}(k, 1) \]
\[ O_3 = \text{LSL}(O_1, 1) \]
\[ O_4 = O_2 + x \]
\[ O_5 = \text{LD}(O_3 + O_4) \]
\[ O_6 = k \times c_2 \]
\[ O_7 = \text{LSL}(y, 1) \]
\[ O_8 = \text{LSL}(O_6, 1) \]
\[ O_9 = O_7 + j \]
\[ O_{10} = \text{LD}(O_8 + O_9) \]
\[ O_{11} = k + 1 \]
\[ O_{12} = O_5 + O_{10} \]
\[ O_{13} = \text{sum} + O_{12} \]

**Figure 4.26:** Pseudo representation of the operations parallelism in the inner loop

2. Some operations use common variables or constants.

3. There are two feedbacks. One representing the increment on the iteration counter. The other represent the accumulation of the variable \( \text{SUM} \) with the values of the previous iterations. These feedbacks represent the inter-iteration dependence.

Figure 4.26 summarizes these operations in the pseudo-code. If we look at the initial C code, we see that some operations are not so obvious at the lower level. In fact, a left-shift operation has to be applied on the starting address of each element because the data are stored by byte in the memory. But, as we work with short elements which take 16 bits, the address of the next element is always two locations next. The left shift operation thus multiplies the calculated address by two to take into account the fact that two emplacements are used for each element (O2, O3, O7, O8). The operations O5 and O10 correspond to the load of the elements situated at the addresses p1 and p2. These addresses are computed by the load operation which takes two operands as inputs, i.e. those coming from O3 and O4 for p1 and from O8 and O9 for p2, respectively.

Besides this information, we can deduce from the .DRE file the nature of the operands, what could also be deduced from the C code presented in Figure 4.23.

- bidirectional (reg_bidir): k, sum. It means that these variables are use in read and write mode inside and outside the loop. These variables will be mapped on the shared VLIW register to be able to make both VLIW and CGA exchange these data.
- live-in (reg_src) : c1, c2, i, j, x, y. The difference with the bidirectional variables is that the live-in are just used in read mode in the loop.
- constants : 1, 1, 1, 1, 1. Five immediate operands are used in the loop, used for the LSL operations and to increment the k variable for the next iteration. These operands will be stored by DRESC in constant memories.
- temporarily data will typically be mapped on local register files.
Having presented the loop structure of our example, we will now give some information about the modeling of this loop before presenting the different scenarios we have decided to simulate for the case study.

Modeling

Two models have been defined to represent the structure of the loop with a Petri Network. They are shown respectively in figures 4.27 and 4.28.

**Dense model** The first figure is a dense representation of the loop composed of the operations constituting the loop body and of feedbacks to include a finite number of iterations. The structure of the Petri Network is characterized by the following points:

- The initial state, ie the starting transition, is composed of the variables and constants used in the loop (places 0→9, 11). More precisely, these places are pseudo operations that will be mapped either on the global register or on constant memories to feed the effective tasks given by the other places of the Petri Net. The variables \(K\) and \(S\) (places 26 and 27) are initialized in a more complex structure as they need to be updated at each iteration.

- Only the variable \(K\) has to be used several times (three) by different operations. This is why we have used three *dummy nodes* at the transition \(T_{15}\). It enables to use \(K\) for operations 14, 16 and 25.

- The structure is composed of two feedbacks: one for the incrementation of the iteration counter \(K\) (from \(T_{12}\) to \(T_{15}\)) and one for the calculus of the sum by accumulation (from \(T_{16}\) to \(T_{18}\)). Their principle has already been explained sooner. We can just notice that the pseudo operations \(M1\) (place 10) and \(M2\) (place 13) are typically mapped on the global register.

- There is no temporarily data that are used multiple times in the loop. For this loop structure, we thus do not need to include pseudo operations to deal with the storage in local registers (see explanations for model 1 of the architecture).

- In the Petri Net, to indicate the number of iterations (32) that have to be performed, we have given some edges a weight different of 1. This is the case for all the variables coming from the starting transition (places 0→9 and 11). For the ten first, we have given the weight 32. This implies that the variables will be "memorized" 32 times each and thus that 32 tokens will be generated to execute the next operations. For the place 11, the weight is equal to 31 as only 31 increments will be done on \(K\) that will be used 32 times altogether.

**Developed model** In the second figure, we do not use feedback but develop entirely the loop by describing explicitly the operations of each iteration successively. It implies some changes in the structure of the Petri Network compared to the previous one.
• As no feedback is needed to update the variables $K$ and $S$, these can be initialized as the other variables at the starting transition (places 12 and 13).

• All the operations (places 14 to 28) are repeated in the Petri Net for each iteration. These repetitions extend the net after the transitions $T_{14}$ and $T_{15}$.

• As the operations of the body loop are duplicated, different places (for each iteration) will use the same variables ($c2, y, c1, ...$) contrarily to the first model where the same places uses several time the variables. Dummy nodes have thus to be used to distribute the variables (places 0 → 9) to all these operations. The number of dummy nodes included is equal to the number of duplications of the operations, i.e. the number of iterations (32). It also implies to add transitions to send the tokens to these dummy nodes, as shown in Figure 4.28 downstream the first places.

• Finally, as the loop is entirely developed, all the edges have a weight equal to 1.

**Scenarios**

As we have decided to only model inner loops, we have chosen the different structures of the loop which do not change the overall structure of the code, meaning the way all the
Figure 4.28: Developed Petri Net of the inner loop of the matrix multiplication function
loops are nested. Moreover, as it can take a lot of time to create Petri Networks for each scenario (see section 4.7), we have decided to limit the number of possibilities modeled from the side of the application.

To obtain different scenarios, we have applied the unrolling method with two factors: 4 and 8. With the loop shown in Figure 4.24, we thus consider three scenarios for the application. The algorithm related to the loop unrolled 4 times is presented in Figure 4.29. The code for unrolling factor 8 can easily be deduced from it.

We have underlined in red the changes that the unrolling has implied. The consequences are clearly that the number of iterations has decreased from 32 to 8. The number of operations on the contrary, has increased as they have been duplicated 4 times. The DDG of these scenarios have also been extracted. We give their characteristics for both unrolling factors below.

Unroll 4 Compared to the Petri Network explained for the basic loop, the structure of the unrolled loop adds some information and has an impact on the number of places and transitions. These are explained below. To illustrate them, we give the pseudo code for the instructions in Figure 4.30.

- The types of operations are the same but the number has changed: 11 add, 6 mul, 5 lsl, 8 ld. Altogether, there are thus 30 operations compared to the 13 in the first scenario. Globally, if we compare Figure 4.30 with Figure 4.26 of the first scenario, we clearly see that the unrolling has increased the number of operations.

```c
void DREC_mat_mul(const short *x, int r1, int c1,
            const short *y, int c2,
            short *r, int q5)
{
    int i, j, k;
    int sum, tmp1, tmp2, tmp3, tmp4;
    short *p1, *p2;

    for (i = 0; i < r1; i++)
        for (j = 0; j < c2; j++)
            {
                sum = 0;
                for (k = 0; k < c1; k++)
                    {
                        p1 = x + k + i * c1;
                        p2 = y + j + k * c2;
                        
                        tmp1 = *p1 * *p2; p1++; p2+= c2;
                        tmp2 = *p1 * *p2; p1++; p2+= c2;
                        tmp3 = *p1 * *p2; p1++; p2+= c2;
                        tmp4 = *p1 * *p2;

                        sum += ((tmp1 + tmp2) + (tmp3 + tmp4));
                    }
                
            r[(j + i*c2) - sum >> q5];
        }
}
```
operations that can be executed in parallel.

- The variables are the same (c1, c2, x, y, i, j, k, sum). But there are different immediate operands: 1 (5 times for the LSL operations), 2, 4, 6 (for the computation of addresses for Load operations) and 4 (increment of $K$).

- As it is highlighted in red in Figure 4.30, some operations produce temporary data that are used multiple times by different other operations. It was not the case in the body loop in the first scenario. It implies that dummy nodes are used in the Petri Network to enable the distribution of these temp data amongst the various consumer places. However, to manage these data, two kinds of structure will be considered depending on the platform model used. Indeed, as shown in Figure 4.31, if explicit RF memorization is introduced in the model, the dummy node spread is done once the data has been stored into a register, what is modeled in the Petri Net by a pseudo operation called $RF$, following directly the producer place. In the other case, when the local registers are just transmission blocks, there is no need for explicit memorization of temp data that will be memorized inside the FU’s, and the duplication of the data is thus directly introduced downstream the producer place. Aside these temp data, an added variable, $c2$ is used two times, by two different operations, what was not the case in the basic loop. It also implies the addition of dummy nodes, two exactly. Concretely, it adds 20 dummy nodes to the Petri Network compared to the first scenario.

- In the model including the explicit memorization of temp data in local registers, 8 pseudo RF operations are added to the Petri Network.

- If we consider the loop represented as a dense model, aside the number of places added in the body as detailed above, the weight of the edges coming out of the starting transition is different as the number of iterations as changed. 32 and 31 are thus replaced by 8 and 7 respectively for incoming edges of places storing the variables (1, 1, 1, 1, 1, 2, 4, 6) and (4).

- If we consider the loop represented as a developed model, the added impact of the unrolling on the structure of the Petri Net appears in the number of dummy nodes used to distribute the variables of the starting transition amongst all the duplicated operations for each iteration. Indeed, the number of dummy nodes linked to these places is reduced to 8 per place, instead of 32.

Unrol 8 Similar observations can be done for the third scenario. We will thus just give the main information that characterize this loop compare to the other scenarios.

- The number of operations in the body has increased and more operations can be executed in parallel. There are 18 additions, 9 multiplications, 5 left shift and 14 loads what give altogether 46 operations.

- The immediate operands are: 1 (5 times for the LSL), 2, 4, 6, 8, 10, 12 (for the LD) and 8 (for the increment of $K$).
### 4.5. MODELING OF THE MATRIX MULTIPLICATION IN NESSIE

<table>
<thead>
<tr>
<th>O1 = i \times c1</th>
<th>O2 = \text{LSL}(k, 1)</th>
<th>O7 = k \times c2</th>
<th>O8 = \text{LSL}(y, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>O3 = \text{LSL}(O1, 1)</td>
<td>O4 = O2 + x</td>
<td>O9 = \text{LSL}(O7, 1)</td>
<td>O10 = O8 + j</td>
</tr>
<tr>
<td>O5 = \text{LD}(O3 + O4); O6 = O3 + O4</td>
<td>O11 = \text{LD}(O9 + O10); O12 = O9 + O10</td>
<td>O13 = \text{LSL}(c2, 1)</td>
<td></td>
</tr>
<tr>
<td>O14 = \text{LD}(O6 + 2); O15 = \text{LD}(O6 + 4); O16 = \text{LD}(O6 + 6); O17 = O5 \times O11</td>
<td>O18 = \text{LD}(O12 + O13); O19 = O12 + O13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O18 = O18 \times O14</td>
<td>O19 = O19 \times O16; O20 = O21 + O22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O25 = O20 \times O24</td>
<td>O26 = O21 \times O25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O27 = O23 \times O25</td>
<td>O28 = O26 \times O27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O29 = \text{sum} + O28</td>
<td>O30 = k + 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.30**: Pseudo representation of the operations parallelism in the inner loop unrolled 4 times - in red: illustration of the multiple use of data

![Diagram](a)

**Figure 4.31**: Illustration of the use of dummy nodes to manage temporary data - (a) when data are explicitly stored in RF; (b) when data are stored in FU
• The temporary data add 35 dummy nodes. The variable $c_2$ is also used twice in the body loop what implies the use of two additional dummy nodes.

• In the model including the explicit memorization of temp data in local registers, 12 pseudo RF operations are added to the Petri Network.

• In the dense model, the incoming edges of the places storing the variables (1, 1, 1, 1, 1, 2, 4, 6, 8, 10, 12) and (8) have respectively a weight equal to 4 and 3 as the number of iteration has decrease to 4.

• In the developed model, the number of dummy nodes enabling the distribution of the variables is reduced to 4 by variable.

All these considerations will be useful when discussing the results in Section 4.6. In the next part, we give the scenarios considered from the side of the architecture with some modeling considerations.

4.5.3 Architecture scenarios

Reflection on the platform modeling has already be done in Section 4.5.1. No particular remarks are added on the structure and the components of the netlists. We thus present directly all the scenarios we have chosen and the degrees of freedom that have been considered to sweep the design space.

We have decided to vary the structures based on the size of the coarse grain array (thus the number and topology of FU’s), the FU’s interconnection scheme, the local registers distribution, the interconnection between FU’s and RF’s and finally the instruction set distribution amongst the array. In particular, we have vary these DoF as followed:

• the size of the CGA : 2x2, 2x3, 3x3, 4x4$^{15}$ ;

• the FU’s interconnection : mesh, mesh plus ;

• the RF distribution : shared RF, fully distributed RF ;

• the FU’s - RF’s interconnection : con1, con2 (distributed RF), fully connected (shared RF)

• the instruction set distribution : two variants have been simulated.

1. all FUs are able to execute the operations LSL, ADD. In addition, the first row (shared by the VLIW) is able to execute the LD, MUL.

2. When there are more than two rows, the FU0 can execute all operations, the others of the first row can execute only the MUL and LD, the second LSL, the others ADD.

The different variants are summarized in the table 5.1.

$^{15}$The first number gives the number of rows of the FU’s array, the second one the number of columns.
### Table 4.5: CGA architecture variants defined for the exploration

<table>
<thead>
<tr>
<th>FUs</th>
<th>sol</th>
<th>shared</th>
<th>RF distributed</th>
<th>ISA 1</th>
<th>ISA 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>con1</td>
<td>con2</td>
<td></td>
</tr>
<tr>
<td>2x2</td>
<td>1.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2x3</td>
<td>4.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2x3+</td>
<td>7.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3x3</td>
<td>10.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>13.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3x3+</td>
<td>14.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>17.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4</td>
<td>18.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>19.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>21.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4+</td>
<td>22.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>23.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>24.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25.</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In the next section, we present the results obtained for all the variants presented above (3 applications structures and 33 architecture structure) considering 4 types of models for the platform, resulting in two models for the Petri Networks (with or without pseudo RF operation). Explanation and discussion of these results are performed. Then they are compared to results obtained with the DRESC compiler. We will also discuss timing issues in both cases, related to the modelisation of the inputs, the simulation time. Hence, we will show to which extend Nessie is able to guide designer to choose efficient solution based on multiple criteria.

4.6 Results Analysis

This section is divided in three parts.

The first one discusses the validity of the models chosen to represent the CGA and the inner loop. This discussion is based on the results produced by NESSIE for the more critical criterion, the cycles, and for 75 solutions.

In the second part, we compare these results with the ones estimated by DRESC for all the solutions to analyze further the modeling capability and the results estimation accuracy of NESSIE.

Finally, in the third part, we discuss the prediction capability of NESSIE based on the estimation of multiple criteria (cycles, area, ED, effIPC) for a set of preselected solutions identified by our tool, with those generated in the external flow.

4.6.1 Models validity

Regarding the different models we have proposed in Section 4.5.1, this first part of the results analysis is devoted to a discussion on the validity of these models based on the main criterion estimated, i.e. the number of cycles needed to execute the application on the architecture. The four architecture models are summarized in the table 4.6 and have been tried for a multiplication of 32x32 matrices.

Amongst these models only two have been finally kept to simulate the system. These are the Model1 and the Model2. Indeed, the Model3 and Model4 have been rejected for the following reason:

- the modeling of FUs as computational blocks without memorization capability leads to deadlocks which consequently give invalid solutions for most of the architecture variants. Indeed, such situation occurs because FUs are blocked when a data has been produced until this data is consumed by another operation.

The deadlock may occur in two cases: either all the FUs are allocated and need token from each other to execute the operation. Or no route exist between two FUs. This can rapidly occur as the FUs cannot be used for transmission until it is unblocked.

As NESSIE does not implement backtracking solutions, and as we cannot change the
allocation/routing policy (see explanations of the allocation/routing weight), these models have been invalidated. In this part, we thus exclusively present the results obtained for the two first models for which we remind the main characteristics:

1. in the model 1, we consider that FU have memorization and transmission capability and the RF are also used as explicit memories enabling data transmission.

2. in the model 2, the FU is represented in the same way, but here, RF are no more explicit memories, but just transmission link. It implies that this block is not able to store tokens.

<table>
<thead>
<tr>
<th></th>
<th>Fu (M-T)</th>
<th>Fu (T)</th>
<th>Buffer (T)</th>
<th>RF (M-T)</th>
<th>(T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Simulations models of the CGA architecture

Several observations have been made when analyzing the mapping performed on this case study. These observations are at the base of the discussion that will be done for the criteria results.

- **HW** blocks cannot memorize two tokens at the same time step. Indeed, the minimum memorization time interval is 1 cycle, even if the user gives a BW bigger than the data token size. It implies that the sending of two tokens from the same producer will always be sent successively at different time steps even if they are consumed by different blocks, with a minimum interval of 1 cycle\(^\text{16}\).

This is also the case when considering dummy nodes used to distribute a token to multiple blocks. In that case, the token is sent successively to each destination.

For this case study, we do not want to add extra cycles when using memorization capability in constant memories and in FU. This state is indeed used just to allow the component to execute several operations successively and to unblock the component. These extra cycles will add overhead on the effective number of cycles.

Depending on the unrolling factor, the overhead will be different. Indeed, when increasing the unrolling factor, the number of dummy nodes for the variables decreases, and the number of dummy nodes to manage the temp data varies (unrol1 : variables=480 D, no temp ; unrol4 : variables=160 D, temp=160 D (20/iteration) ; unrol8 : variables=92 D, temp=140 D (35/iteration)).

\(^{16}\)A BW+16 with data token size=32 implies an interval of two cycles between two sendings
• **Cycles** are computed at different events triggering: when a FU *executes* a task or *transmits* a data token, when a RF *memorizes* a temp data (only in model2) or *transmits* a data token, when a buffer *transmits* a data token. Consequently, FU cannot transmit and compute at the same time step. So it is for RF for memorization and transmitting states. The model1 has the advantage to relieve the FU that are not overdriven by the management of temp data. However, as RF cannot transmit during memorization of these temp data, overhead can be more significant for shared RF if a lot of communication is needed. It would be the case in particular when a lot of FUs share the same RF and if remote communication has to be performed.

In the model2, of course, RF can be used exclusively for transmission what could be more interesting for big arrays.

• There are three communication links to send data tokens from one FU to another: the FUs themselves, the RFs and the output buffers. Their use will depend on the relative location of the token producer and consumer and the availability of the communication links. Moreover, depending on the RF topology (shared, CON1, CON2) and the FU’s interconnection (mesh, mesh+), interconnection will be more or less increased.

![Diagram](image.png)

**Figure 4.32:** Number of cycles of 33 solutions simulated in NESSIE for the Model1

Figures 4.32 and 4.33 show the number of cycles estimated by NESSIE for all the architecture and application variants. On the X axis, the variants are ordered as follow:
4.6. RESULTS ANALYSIS

The 7 first variants refer to the architecture with shared register and the first instruction set distribution;

- The next four variants change the instruction set repartition (see section 4.5.1);

- The next group, with the suffix a, is composed of the architectures with the first instruction set distribution and with distributed local registers following a CON1 FU-RF interconnection;

- Finally the last group, with suffix b, gives the same variants than the previous group, but with a CON2 FU-RF interconnection.

The three curves correspond to the three application scenarios: the basic loop (unrolling factor 1) in blue, the loop unrolled 4 times in red and the loop unrolled 8 times in green. Global observations can be done.

1. We see that less solutions have been produced for model1 (fig. 4.32) than for model2 (fig. 4.33). Indeed, with model1, explicit memorization has to be included in Petri Networks. However, when several RF are available in the architecture, as it is the case in distributed RF CON1 and CON2 variants, we cannot predetermine where a temp data will be stored and cannot let the choice to NESSIE which could choose a wrong RF, i.e. a RF not directly linked to the FU producer of the temp data. We
have still produce results for the first model to make comparison and analysis based on the 11 first architecture variants.

2. If we look at the global curve given by Figure 4.33 for model2, we observe three peaks, related to the array variants 2x2.

Explanation : As illustrated on Figure 4.34 which gives the activity report of the variants 2x2 for the “unroll” loop scenario, most of the operations of the Petri Network are mapped on the first row of the array. This is due to the instruction set distribution in which all the operations (ld, mull, lsl and add) are in the compatibility list of the first row while the others FUs can only execute the “add” and “lsl”. When increasing the size of these rows, as it is the case with the other array variants (2x3, 3x3, 4x4), the architecture offers more parallelism to execute the operations, what explains the lower number of cycles for these solutions. Cycles differences are less significant between architectures 2x3→4x4 due to the size of the application which do not require as many parallelism as offered by the array.

3. If we look at Figure 4.33, we see that the distributed RF variants (a and b) are less efficient than the shared ones.

Explanation : indeed, the shared register creates full interconnection between FU’s. This is not the case with CON1 and CON2 where extra connections are settled unidirectionally between diagonal FUs.

However this result is not obvious to foresee as it depends on the mapping performed by NESSIE which conditions the routing choice. If the shared register enables full
interconnection, it cannot be used to transmit several data simultaneously. In addition, it is not obvious to predict if NESSIE will take advantage of the communication scheme as the task allocation performed on different rows by our tool is independent on the interconnection topology of the array.

To complete this model discussion, we compare the two models below. This is done for the shared architectural variants, i.e. the first 11 ones. Figure 4.35 shows this comparison for the three application scenarios. The blue and red curves are respectively given for the model1 and model 2. Moreover, the basic loop (unrol1) is marked with a square, the loop unrolled 4 times, by a cross, and the loop unrolled 8 times, by a circle. The curves give globally the same trend, but some points require the following remarks.

- in the first application scenario, there is no temp data to manage. The hardware components are thus used exactly the same way in both models. We expect no difference between the two models, as we can see on Figure.

- in the unrol4 scenario, the model 2 generates more cycles than the model1 but the trend is very close. In this scenario, we take advantage of the use of the RF to manage the temp data, due to the fact that the RF is not too much used for communication.

- in the unrol8 scenario, we observe a mixed behaviour. The curves of the model1 and 2 cross together. Indeed, there are more temp data to manage in each iteration (35/iteration compared to 20 for the unrol4) what overdrives more the FUs in the model2 and more the RF in the model1.

Moreover, as there are more operations that can be executed in parallel, there is more intercommunication between the FUs, and thus the RFs and FUs will be more employed for transmission. Depending on the array size and the way the operations are allocated to the resources, cycle overhead is added by the FU (particularly visible for variants 2X2, 2X3, 4X4meshA of model2) or by the RF (lightly visible for the variants 3X3mesh and 3X3mesh+).

N.B: the 4X4meshA variant is particularly worse in the model2 compared to the model1 in both scenario unrol4 and 8. This is due to the instruction set A which forces a more distributed allocation of the resources and thus creates more remote interconnections which require to solicit more the FU. This is particularly sensitive for the model2 where the FUs are already overdriven by the management of temp data.

To highlight these points and see the impact of both models on the results accuracy, we compare them with results estimated by DRESC in the next part of Section. This will confirm the previous observations and underline other behaviours of NESSIE.

### 4.6.2 Results reproducibility: NESSIE vs DRESC

In this part, we compare results obtained by NESSIE and DRESC to see how far our tool is able to reproduce the solutions trend for a given criterion. We still base this comparison...
Figure 4.35: Comparison of the Model1 and Model2 on the 33 shared RF CGA variants on the *cycles* execution.

Figure 4.36 gives the results estimated by DRESC for all the same 75 solutions simulated in NESSIE. We find thus the same axis and the same colors for the application scenarios.

Figure 4.36: Number of cycles of the 75 solutions compiled in DRESC

At first glance, we can make two observations. First, we see that the prediction accuracy of NESSIE is substantial regarding the relative
trend of the solutions compared with the DRESC curves. Second, the absolute values are far from the one given by our tool. This is explained by the extra cycles added by the tool each time a token is memorized and sent from a computing block to another (see section 4.6.1). Moreover, as numerous dummy nodes are added to manage the duplication of variables in the Petri Networks, this increases the number of extra time steps added by NESSIE. However, we have observed that, globally, the number of added time steps (and thus cycles) is equivalent for each loop scenario.

Having in mind this remark, we have normalized the NESSIE results to be able to make a workable comparison with the DRESC results. Based on Figure 4.36 and before making this comparison, we can already underline the main characteristics of the graph obtained with DRESC estimations, that will feed the following discussions.

- The scenario ”unrol1” gives a regular trend whatever the register topology and interconnection and for both instruction set distributions. In particular, changing the RF topology does not have impact on the execution time.

  **Explanation:** if we look at Figures 4.37, 4.38 and 4.39 that respectively show the schedule performed by DRESC\(^{17}\) for the variants 3X3mesh, 3X3mesh a and 3X3mesh b, we see that the allocation of the tasks has been performed almost exclusively on the first row of the array. This implies that the diagonal interconnections are not used what explain that no difference is observed between shared RF and CON1/2 configurations.

  Moreover, the architecture 4X4 offers a better parallelism than the architectures with only 2 or 3 FUs by row for the considered loop and reduces the number of cycles.

- In the scenarios ”unrol4” and unrol8, we see that the variants 2X2 differ between shared, con1 and con2. In these architectures, the mapping performed by DRESC has lead to more communication between the row 1 and the row 2 of the array, which is not favored with the con2 RF interconnection.

- This is the contrary in unrol4 for the 2X3 and 3X3 variants. These opposite behaviours show the difficulty to predict the final performance of the system which depend on the allocation and routing choices done by DRESC.

- We observe in the green curve that some points define a trend which differs from the two other scenarios. This is the case for variants 4, 9 and 11 (3X3mesh, 3X3mesh+A, 4X4mesh+A). We could expect that these solutions perform better than respectively 2x3mesh+, that offers less parallelism than 3x3 arrays, and 4x4meshA, that is composed of less interconnection than the 4X4mesh+A.

\(^{17}\)Extracted from the schedule viewer of DRESC
**Explanation:** this can be explained by the fact that for unrol 8, the number of operations in the loop body is sufficiently high to be difficult to find an efficient schedule on the CGA. Moreover, as we have just chosen default values for the DRESC parameters, we have limited the search algorithm possibilities for finding a good solution. Tuning these parameters (like random_seed, relax_factor) could certainly give better results for these cases. This will be confirmed in Section (design time) where we see that the compilation take much more time for unrol8 than the other scenarios as finding a good scheduling is time expensive in DRESC. Better solutions could be found if parameters were tuned.

- The curves ”unrol4” and ”unrol8” are closer together compared to the first scenario ”unrol1”.

**Explanation:** it is explained by the fact the the proportion of operations obtained between each application is different. Indeed, between unrol4 and unrol1, the ratio is 2.3 ; between the unrol8 and unrol4, the ratio is 1.5.

In the next part, we compare these results with NESSIE graphs to validate our models.
Figure 4.38: Schedule viewer of the 3X3 mesh with CON1 distributed RF variant for the unroll1 loop scenario

and discuss the estimation capability of our tool. Figures 4.40 and 4.41 present the graphs superimposed for respectively the model 1 and the model 2. In blue, we find the results estimated by NESSIE normalized to the value of the solution $2X3mesh-unroll1$\textsuperscript{18}. In red, we give the results obtained by DRESC.

We successively analyze these graphs for the Model 1, compared on the 11 first architectural variants, and the Model 2, on the 75 solutions.

Model 1

Figure 4.40 shows that the trends are close together. However, some points require a particular attention. The following remarks can be done:

Unroll 1 Cycles generated by NESSIE present more variability compared to DRESC and sometimes a different trend. More precisely, in the red curve, we see that mesh+ variants do not improve the mesh counterpart regarding the same array size. In NESSIE, on the

\textsuperscript{18}We have noticed previously that the values given for the 2X2 variant include significant overhead compared to other solutions an is thus less relevant.
contrary, improvement is achieved when using $mesh^+\,$ instead of $mesh$. Still, in the blue curve, the $3X3mesh$ and $4X4mesh$ do not improve or give worse result compare to respectively $2X3mesh^+$ and $3X3mesh^+$. On the contrary, DRESC take advantage of more hardware resources to parallelize the loop and reduce the number of cycles.

**Explanation:** the first observation is explained by the fact that NESSIE allocates the resources independently of the communication between the FU’s. Indeed, the token routing is performed after the allocation of the consumer. Two resources that have to exchange a token have thus sometimes to use ”indirect” links, what increases the total number of cycles.

On the contrary, DRESC allocates the tasks by optimizing the communication and thus does not take more advantage of $mesh^+$ compared to $mesh$.

For the same reasons, NESSIE does not take a lot of advantage of the increase of the array size when $mesh$ is used. Indeed, although more operations can be parallelized, it implies more remote communications. The parallelism gain is thus decreased by the communication overhead. This behaviour is more sensitive when the array growths. This can be taken into account when selecting solutions as we can expect more efficient solutions.
4.6. RESULTS ANALYSIS

Figure 4.40: Comparison of the cycles between the DRESC and NESSIE results for the Model1

Figure 4.41: Comparison of the cycles between the DRESC and NESSIE results for the Model2 for such architectures.
In DRESC, allocation of tasks is done such that resources communication is optimized.

Unrol4 We see that for unrol4, we are closer to the behaviour of DRESC. As the unrolling offers more parallelism, the mapping will induce more direct communications between tasks compared to the unrol1 scenario. The overhead in bigger mesh arrays is thus less sensitive for this loop structure.

Unrol8 It is still better for unrol8. However, we had identified that DRESC gave strange results for 3x3, 3x3+A, 4x4+A. On the contrary, Nessie gives logical results where the cited architectures improve the cycles compared to the 2x3+, the 3x3A and the 4x4A.

This observation is important. Indeed, it shows that Nessie releases some constraints that DRESC imposes via its metaheuristic parameters. In other words, as Nessie is able to identify the potential efficiency of architectures compared to others, solutions that would maybe have not been explored in DRESC can be considered and refined.

Model 2
The following remarks can be done for Figure 4.41:

Unrol1 For this scenario, the shared RF variants give the same results than the model1. The analysis made above is thus the same. Concerning the "distributed RF" variants, we see that NESSIE produces worse results compared to the shared variants, although DRESC gives identical results. As shown in Section 4.3.1, distributed RF offer less direct communications between FUs compared to shared RF. As NESSIE does not optimize the allocation of tasks regarding the routing capability, these variants add extra cycles. DRESC, on the contrary, find a schedule that enable a better use of the interconnection in the array.

Unrol4 In this scenario, three variants generated by NESSIE, 3X3mesh, 2X3mesh+ a and 4X4mesh b, present a different trend compared to DRESC. The 3X3mesh and the 4X4mesh b, the reasons have been given in the Model1 analysis and is related to the mapping policy of NESSIE.
Concerning the solution 2X3mesh+ a, the result given by DRESC is not logical as we would expect a better number of cycles for this architecture compared to the mesh one, as identified by NESSIE. Again, we explain that by the metaheuristic policy implemented in DRESC and the influence of its parameters.

Unrol8 As it was the case in Model1, NESSIE generates with the Model2 the same solutions that seem wrong in DRESC : 3X3mesh, 3X3mesh++A, 4X4mesh++A. Moreover, with the distributed RF topology, we find again the impact of the mapping policy in NESSIE for the solutions 4X4mesh a and 4X4mesh b.
4.6. RESULTS ANALYSIS

Conclusion

In this section, we have analyzed the modeling and estimation efficiency of NESSIE based on a comparison of the results generated by our tool with those estimated in DRESC, for a critical criterion, the cycles. This discussion has shown that the models chosen to represent the CGA are globally equivalent for this case study as the loop structure is composed of a reasonable amount of operations and temp data (responsible of overhead differences between the two models).

We have also highlighted that, for this case study, Nessie is able to estimate quite faithfully the behaviour of the ADRES processor executing an entire inner loop, on 75 solutions. However, for some solutions, NESSIE over-estimates the number of cycles, for others, DRESC produces non logical results contrarily to NESSIE.

These observations results of the intrinsic mapping policy of the tools:

- NESSIE allocates tasks on hardware blocks by selecting the first free HW candidate. As DRESC, the resource usage is mainly shared out the first row of the array. This leads to a similar mapping schema for both tools.

- However, as NESSIE does not consider the allocation and the routing jointly, but sequentially and independently, the inter FU’s communication is not optimized during the mapping performed by our tool. On the contrary, DRESC allocates tasks on FU’s in order to reduce the communication overhead. This explains that some solutions estimated by NESSIE are worse than those estimated by DRESC.

- Finally, the simulated annealing algorithm governing the mapping policy in DRESC is constrained by parameters that control the scheduling time and spread. The research of a valid schedule is conditioned by these parameters which stop the mapping even if a more optimal solution could be found. When increasing the complexity of the application, as it is the case for unrol 8 in which much more operations have to be scheduled on ADRES compared to unrol 1 and 4, we obtain sometimes underestimated solutions.

Along these observations and explanations, we have reasoned based on the architecture structure and the scheduling performed by the tools.

These reflections have shown that sometimes, solutions that are expected to be good, are not detected by DRESC.

Moreover, if one had to make these analysis by head, without generating the simulations, it would require a lot of effort. Indeed, the way registers are interconnected, the number of slots available for each kind of operation, the way the scheduler will use these resources in space and time, have an impact on the global latency (due to the temp data storage, use of MOV operations for FU transmission\textsuperscript{19}, interconnection latency, maximum degree of parallelism to execute operations) which is not obvious to predict. This is even more

\textsuperscript{19}MOV operations are used in ADRES to transmit a data from the input to the output of the FU. This add 1 cycle of latency. This is equivalent to the transmission state in NESSIE.
the case for complex applications in which other loop transformations are applied.

In the next section, we complete this study from a multicriteria perspective. Indeed, by adding dimensions to the performance estimation, we add complexity to the solution exploration and decision which becomes unworkable for the designer. It underlines the importance of a fast exploration tool like NESSIE and the necessity to trust in its estimations or at least, to complete the information with a degree of confidence.

This additional results will show the impact of the previous observations on the reliability of our tool. As no major differences have been observed between the two models and as more solutions have been generated with the model2, this last will be used in the further discussions.

4.6.3 Performance prediction based on multiple criteria

In this part of Section, we test the prediction capability of NESSIE based on multiple criteria. The experience consists in:

- generating in NESSIE the 75 solutions presented above for four criteria (cycles, area, effective IPC and effective density);
- amongst these results, selecting a set of three best solutions based on decision constraints. These solutions stands for the candidates that would be refined in the DRESC flow;
- comparing and discussing this set of solutions with the ones that would have been chosen in the external flow. Therefore, same criteria have been estimated in DRESC for all the solutions.

To analyze and compare the results, we present below 2D and 3D multi-criteria graphs in which "Pareto" points have been highlighted. From these graphs, we point out the trade-offs that must be made and select three solutions based on a "arbitrary" cycles constraint. Finally, the preselected solutions are compared with the ones generated in DRESC in a table for further discussion and validation of NESSIE.

NESSIE multicriteria estimation

For the concern of readability, we have produced different 2D graphs that show 2 by 2, how the criteria evolve depending on the architecture variants and the loop scenario. Figures 4.42, 4.43 and 4.44 present the trade-offs for the cycles with respect to respectively the area of the chip, the effective IPC and the effective density. Two other figures, the 4.45 and 4.46 give the results for the area in relation with the effective IPC and the effective density respectively.

For all the graphs, the pareto curve, which gives the best solutions for the considered criteria, is located in the left-bottom corner and is composed of circled points. For these
4.6. RESULTS ANALYSIS

<table>
<thead>
<tr>
<th>Variants</th>
<th>CYCLES</th>
<th>AREA</th>
</tr>
</thead>
<tbody>
<tr>
<td>sol1 u8-2X2 mesh</td>
<td>176.4</td>
<td>21.15</td>
</tr>
<tr>
<td>sol2 u8-2X2 conl</td>
<td>119.9</td>
<td>21.64</td>
</tr>
<tr>
<td>sol3 u8-2X3 mesh</td>
<td>94.76</td>
<td>31.5</td>
</tr>
<tr>
<td>sol4 u8-2X3 mesh+</td>
<td>86.77</td>
<td>31.72</td>
</tr>
<tr>
<td>sol5 u8-3X3 mesh+ A</td>
<td>83.34</td>
<td>33.7</td>
</tr>
<tr>
<td>sol6 u8-3X3 mesh</td>
<td>81.91</td>
<td>41.54</td>
</tr>
<tr>
<td>sol7 u8-3X3 mesh+</td>
<td>77</td>
<td>42</td>
</tr>
<tr>
<td>sol8 u8-4X4 mesh+ A</td>
<td>73.64</td>
<td>54.85</td>
</tr>
<tr>
<td>sol9 u8-4X4 mesh+</td>
<td>70.5</td>
<td>70.3</td>
</tr>
</tbody>
</table>

Table 4.7: Pareto solutions related to the cycles and the area

solutions, the corresponding CGA variant has been annotated on Figure. Blue, green and red colors correspond respectively to the unroll1, unrol4 and unrol8 scenarios of the loop.\(^{20}\)

We present the different figures below.

**Cycles vs Area** In Figure 4.42, we see that the best solutions are exclusively given for loops with unrolling factor 8. What means that for a same architecture area, these solutions give the best performances regarding the number of cycles. The values for these Pareto solutions are given in the table 4.7. Moreover, if small CGA are interesting to reduce the total area of the chip, they are not efficient to parallelize the loop and give bad results for the cycles. This is the contrary for bigger array sizes. For these criteria, the solutions that are in the middle target 2X3 mesh+ or 3X3 mesh+ with shared register files architectures.

**Cycles vs effective IPC** In Figure 4.43, we see that all unrolling factors (1, 4, 8) offer interesting solutions depending on the trade-off accepted by the designer. Indeed, if unrol8 gives the best number of cycles, however is is not efficient from the point of view of the IPC because the number of iterations in the loop is low and this increases the epilogue/prologue overhead compare to the kernel. For the unrol 1, it is obviously the contrary. In all cases, we see that the trade-off are obtained for 4x4 mesh+ with shared RF architectures. Indeed, these architectures are those that reduce the most the number of cycles. Moreover, they also enable the execution of more operations by cycle by increasing the potential parallelism. For these criteria, the application scenario "unrol4" is the best trade-off.

The results are summarized in the table 4.8

\(^{20}\)Take care that these colors are different than those used in the previous section.
Figure 4.42: Nessie results: cycles vs area (model2)

Cycles vs ED  In Figure 4.44, we have given the solutions for the effective density (ED). If this criterion does not give exactly the same information than the SD, it however shows how the resources are used on the CGA.

If we compare the dominant solutions of this figure with those of Figure 4.43, we see that, apart for the "unrol 8" scenario, the best architectural variants are not the same. Indeed, if increasing the number of FU increases the potential parallelism and thus reduce the number of cycles, it introduces more unused resources and reduces the effective density. For these criteria, a good trade-off is offered by the CGA variants 2X3 mesh+ with shared register, for unrol4 or 8.

The values for the Pareto solutions are summarized in the table 4.9.
4.6. RESULTS ANALYSIS

Area vs effective IPC  In Figure 4.45, we see a behaviour contrasting totally the one shown on figure 4.42. Indeed, here the unrolling factor 1 beats all the other solutions and exclusively compose the pareto curve for the criteria area and effective IPC. This is explained by the fact that for a same architecture, the effective IPC will always be better for unroll1 as the number of iterations is the bigger in this case. The circled points also show that there is really a trade-off between the cga size and the effective IPC, i.e. the software pipelining efficiency.

For these criteria, the arrays 3X3 and 2X3 mesh+ with shared register offer an interesting compromise. This is summarized in the table 4.10.

Area vs ED  In Figure 4.46, the best solutions are offered by the variants 2X2 mesh, whatever the RF interconnection topology and for the unroll1 scenario. However, the same
### Table 4.8: Pareto solutions related to the cycles and the effIPC

<table>
<thead>
<tr>
<th>Variants</th>
<th>CYCLES</th>
<th>effIPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>sol1 u1-4X4 mesh+</td>
<td>110.4</td>
<td>4.05</td>
</tr>
<tr>
<td>sol2 u4-4X4 mesh+</td>
<td>85</td>
<td>2.91</td>
</tr>
<tr>
<td>sol3 u8-4X4 mesh+</td>
<td>70.5</td>
<td>2.66</td>
</tr>
</tbody>
</table>

### Table 4.9: Pareto solutions related to the cycles and the ED

<table>
<thead>
<tr>
<th>Variants</th>
<th>CYCLES</th>
<th>ED</th>
</tr>
</thead>
<tbody>
<tr>
<td>sol1 u1-2X2 mesh con2</td>
<td>192.38</td>
<td>58.21</td>
</tr>
<tr>
<td>sol2 u1-2X3 mesh+</td>
<td>138.43</td>
<td>53.93</td>
</tr>
<tr>
<td>sol3 u4-2X3 mesh+</td>
<td>103.04</td>
<td>40.1</td>
</tr>
<tr>
<td>sol4 u8-2X3 mesh+</td>
<td>86.77</td>
<td>36.11</td>
</tr>
<tr>
<td>sol5 u8-3X3 mesh+</td>
<td>77</td>
<td>27</td>
</tr>
<tr>
<td>sol6 u8-4X4 mesh+</td>
<td>70.5</td>
<td>16.66</td>
</tr>
</tbody>
</table>

### Table 4.10: Pareto solutions related to the area and the effIPC

<table>
<thead>
<tr>
<th>Variants</th>
<th>AREA</th>
<th>effIPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>sol1 u1-4X4 mesh+</td>
<td>70.3</td>
<td>4.055</td>
</tr>
<tr>
<td>sol2 u1-4X4 mesh+ A</td>
<td>54.85</td>
<td>3.86</td>
</tr>
<tr>
<td>sol3 u1-3X3 mesh+</td>
<td>42</td>
<td>3.47</td>
</tr>
<tr>
<td>sol4 u1-3X3 mesh+ A</td>
<td>33.7</td>
<td>3.38</td>
</tr>
<tr>
<td>sol5 u1-2X3 mesh+</td>
<td>31.72</td>
<td>3.23</td>
</tr>
<tr>
<td>sol6 u1-2X3 mesh</td>
<td>31.52</td>
<td>3.089</td>
</tr>
<tr>
<td>sol7 u1-2X2 mesh con2</td>
<td>21.69</td>
<td>2.32</td>
</tr>
<tr>
<td>sol8 u1-2X2 mesh con1</td>
<td>21.64</td>
<td>2.21</td>
</tr>
<tr>
<td>sol9 u1-2X2 mesh</td>
<td>21.15</td>
<td>1.585</td>
</tr>
</tbody>
</table>
4.6. RESULTS ANALYSIS

Figure 4.44: Nessie results : cycles vs ED (model2)

solutions are very bad regarding the cycles and the effective IPC.

These different graphs clearly show the challenge consisting in finding the solution that will correspond to the design specifications and demonstrate the complexity of exploring solutions based on multiple criteria without an automatic tool such NESSIE. The values are given in the table 4.11.

Below, we make the experience of selecting a set of solutions based on a cycles constraint. Therefore, we have extracted the Pareto solutions based on the different criteria together. We present the results further and compare them with DRESC estimations subsequently.
Preselection of solutions

We illustrate in Figure 4.47 all the possible trade-offs that are possible for three criteria (cycles, area, effective IPC). The Pareto points are circled in magenta. A similar figure could be shown for others combinations of criteria.

As it is not possible to find solutions that outperform the other according to all the criteria, the designer has to give constraints, that are typically imposed by the specifications, and that restrict the number of solutions that will be selected for the lower design steps.

In our example, we have arbitrarily imposed a maximum number of cycles equal to 80 to converge to a smaller set of interesting solutions for the design of our case study. This results in three possible solutions for the design, that are given in the table 4.12 with
the corresponding values of the criteria\textsuperscript{21}. We see that NESSIE identifies three solutions resulting from the cycle constraint which are all related to the unrol8 loop scenario, all for the shared RF topology and for the bigger array size: \textit{3X3 mesh+}, \textit{4X4 mesh+} and \textit{4X4 mesh+A}.

\textit{Comparison with DRESC:} We give in the table 4.13 the values of the criteria estimated by DRESC for these three pre-selected solutions. We have add the \textit{scheduling IPC} and the \textit{scheduling density (SD)}. This table shows that, for these solutions, DRESC gives close values, apart for the variant \textit{u8-4X4mesh+A}. This variant had been discussed in Section 4.6.2 and is not identified by DRESC as a good solution due to the scheduling parameters.

If we look at the solutions DRESC would have selected amongst the 75 variants, without

\textsuperscript{21}The graphs including the constraint are given in the AppendixA for the interested reader.
Table 4.11: Pareto solutions related to the area and the ED
4.6. RESULTS ANALYSIS

<table>
<thead>
<tr>
<th>Variants</th>
<th>CYCLES</th>
<th>AREA</th>
<th>effIPC</th>
<th>ED</th>
</tr>
</thead>
<tbody>
<tr>
<td>u8-3X3 mesh+</td>
<td>77</td>
<td>42</td>
<td>2,439</td>
<td>27,1</td>
</tr>
<tr>
<td>u8-4X4 mesh+</td>
<td>70,5</td>
<td>70,3</td>
<td>2,66</td>
<td>16,66</td>
</tr>
<tr>
<td>u8-4X4 mesh+ A</td>
<td>73,54</td>
<td>54,85</td>
<td>2,55</td>
<td>15,95</td>
</tr>
</tbody>
</table>

Table 4.12: NESSIE preselected solutions for cycles constraint below 80

the use of NESSIE, we obtain the solutions presented in the table 6.1, for the same cycle constraint. We also see that two of the pre-selected solutions are identified by DRESC. For the three others, NESSIE under estimates the efficiency of the solution. However, if we look at Figure 4.48 giving the relative error between the solutions estimated by DRESC and NESSIE (in %) regarding the number of cycles, we see that for these solutions, the error does not exceed 20% what is very reasonable: u8-2X3 around 20%, u4-4X4+A around 15% and u8-4X4 below 10%.

<table>
<thead>
<tr>
<th>Variants</th>
<th>CYCLES</th>
<th>schIPC</th>
<th>effIPC</th>
<th>SD</th>
<th>ED</th>
</tr>
</thead>
<tbody>
<tr>
<td>u8-3X3 mesh+</td>
<td>74</td>
<td>4,27</td>
<td>2,54</td>
<td>47,47</td>
<td>28,23</td>
</tr>
<tr>
<td>u8-4X4 mesh+</td>
<td>66</td>
<td>5,875</td>
<td>2,85</td>
<td>36,72</td>
<td>17,8</td>
</tr>
<tr>
<td>u8-4X4 mesh+ A</td>
<td>88</td>
<td>5,22</td>
<td>2,72</td>
<td>36,72</td>
<td>17,8</td>
</tr>
</tbody>
</table>

Table 4.13: Estimated criteria for preselected solutions - DRESC results

<table>
<thead>
<tr>
<th>Variants</th>
<th>CYCLES</th>
<th>schIPC</th>
<th>effIPC</th>
<th>SD</th>
<th>ED</th>
</tr>
</thead>
<tbody>
<tr>
<td>unrol8 2x3</td>
<td>78</td>
<td>3,91</td>
<td>2,41</td>
<td>65,28</td>
<td>40,17</td>
</tr>
<tr>
<td>unrol8 3x3+</td>
<td>74</td>
<td>4,27</td>
<td>2,54</td>
<td>47,47</td>
<td>28,23</td>
</tr>
<tr>
<td>unrol8 4x4</td>
<td>72</td>
<td>4,7</td>
<td>2,61</td>
<td>29,37</td>
<td>16,31</td>
</tr>
<tr>
<td>unrol8 4x4+</td>
<td>66</td>
<td>5,875</td>
<td>2,85</td>
<td>36,72</td>
<td>17,8</td>
</tr>
<tr>
<td>unrol4 4X4+A</td>
<td>76</td>
<td>5,16</td>
<td>3,263</td>
<td>32,29</td>
<td>20,39</td>
</tr>
</tbody>
</table>

Table 4.14: Preselected solutions in DRESC flow for cycle constraint

**Example for an ED constraint:** If we had favoured the effective density, we would have completely different results that are given in the table 4.15 for the NESSIE pre-selection, for a minimum ED equal to 50%. We see in this case that the best solutions are those related to the unrol1 loop scenario, and for smaller CGA array sizes: 2X2mesh con1, 2X2 mesh con2, 2X3 mesh and 2X3 mesh+.

If we look at the solutions that DRESC would have chosen without the prediction of NESSIE, we obtain the result given in the table 4.16 for the same ED constraint. We see that only one solution has been selected, which is not found by NESSIE: u4-2X2 mesh. To weigh these results against, we show the relative error between DRESC and NESSIE regarding the ED\(^2^2\). In this figure, we see that:

\(^2^2\)For readability, the % is given relatively to the NESSIE values, contrarily to the relative error given
CHAPTER 4. CASE STUDY 1 : IMEC

Figure 4.48: Relative error (in %) between NESSIE and DRESC for the cycles

- for the variant $u4-2x2mesh$, we have a big error (80%) due to the significative overhead introduced by the model2 for this architecture (see section 4.6.2).

- however, for the other solutions, pre-selected by NESSIE, we see that the relative error is well below 10%, what means that if we relax a bit the constraint, DRESC should also have chosen these solutions.

This analysis underlines that, with a reasonable degree of accuracy, NESSIE is able to predict the performances of a low level design tool, based on multiple criteria.

In the next section, we complete this study by discussing the design time and modeling time gain using NESSIE upstream from the ADRES design flow, compared to the use of the classical tool chain alone.

for the cycles in Figure 4.48.
4.7 Tools integrations

In this section, we compare the two design flows to point out if NESSIE can reduce the design time and improve the quality of designed solutions, and identify what are the eventual limitations that could prevent our tool to improve the classical design flow. To enable this comparison, Figure 4.50 and 4.51 show respectively the critical parts that induce time spending in separated DRESC and NESSIE flows. Then, Figure 4.52 shows an integrated view of the flow where NESSIE pre-selects solutions for DRESC.
4.7.1 DRESC timing

In Figure 4.50, we identify three main delays: $\Delta T_1$, $\Delta T_2$ and $\Delta T_3$. The different steps of the flow are:

1. Having chosen a C algorithm and an ADRES template, perform the DRESC compilation. The time spent depends on the complexity of the algorithm and the size of the ADRES architecture and is given by $\Delta T_1$. In our case study, the compilation of the 75 solutions have taken around **14 hours**: 30 minutes for unroll1, 3 hours for unroll4 and 10.5 hours for unroll8.

2. Once the criteria (total cycles) and the metrics (II, length, schIPC, effIPC, SD) have been taken down from .DRE files, the designer can decide if the solution has to be improved or changed. In practice, the designer will not change both SW and HW but first consider the SW structure. Moreover, he will improve the structure of the algorithm based on the previous choice. The time spent to change/adapt the algorithm is given by $\Delta T_2$. For our case study, the algorithm being simple, the time spent to change the structure of the loop is few minutes.

3. Then the new solution is also compiled by DRESC according to a new time $\Delta T'_1$.

4. Depending on the specifications and the timing constraints, the designer chooses to go back to step 2, to try step 5 or to go to step 6.

5. When the designer is satisfied or not by the algorithm structure, he can also decide to change the HW. The effort to define a new instance of ADRES in the dedicated XML
6. Finally, the designer is satisfied by the solution, stop the exploration and go to the next design stage.

4.7.2 NESSIE timing

Now, if we look back to the design flow of NESSIE, shown in figure 4.51(a), we identify several steps that induce time spending. The delays are introduced by the modeling steps and by the simulation phase:

- To get a Petri Net representation of all desired application scenarios. The starting description we have is the C algorithm, which is the input of the DRESC compiler. We have explained in Section 4.4.2 that the transformation of this code to the Petri Net description in the NESSIE simulation file takes time represented by the $\Delta T_1$ in Figure 4.51 (a). This delay is the global delay needed to extract the Petri Net for all
Figure 4.52: Timings introduced in the flow integrating NESSIE with DRESC

the solutions, what corresponds to the transfo 1 in Figure. The steps of this transfo 1 are detailed at the top of Figure 4.51 (b) where we see that a ddg is extracted from the pre-cga step of the DRESC compiler ($\Delta T_1$) and a manual description of the Petri Net ($\Delta T_1b$).

The mean time spent to get one Petri Network description into NESSIE manually is approximated to 2 hours\textsuperscript{23}.

- To get a netlist description and structure of all desired architecture scenarios. Indeed, if the architectures are defined in XML files for the DRESC compiler, we need to describe models of these architecture in the format imposed by NESSIE. Thanks to the TCL/TK interface we have built, we have facilitated the getting of the main information to model the CGA. We have represented this step in the bottom of Figure 4.51(b) ($\Delta T_{2a}$). From the resulting table we can define an architecture model in the simulation file of NESSIE ($\Delta T_{2b}$).

The time spent today to model on architecture scenario is approximatively estimated to 1 hour\textsuperscript{24}.

- To extract a model for all criteria that should be evaluated by YETi. To feed the behaviour files of all the HW blocks in each possible states, we need to extract a model to evaluate all criteria.

\textsuperscript{23}This time would be highly decreased by using a GUI.
\textsuperscript{24}The transfo1 and transfo2 care much time consuming when having to do the work for the first time due to model analysis.
For this case study, this modeling phase was quite direct as simple analytical expressions are used to calculate the criteria.

- Aside these preparations steps, the simulation time is the third main delay, $\Delta T_3$, introduced by the tool. This delay is easily quantifiable if we include a timer in the simulator.

For the 75 variants considered in the case study, the simulations have taken approximately **1.5 hours to 2 hours**.

### 4.7.3 Integrated flow

Having in mind this information, we can estimate the design time that would be achieved when using NESSIE upstream from DRESC in order to refine a small set of solutions. Compared to the ADRES designed flow alone, the use of NESSIE aims at reducing the number of iterations, represented by the delay $\Delta T_5$ in Figure 4.52.

As a reference to this discussion, we suppose that NESSIE pre-selects 3 solutions. The table 4.17 summarizes the different times that compose the global design time for the classical design flow (named DRESC in the table) and the one integrated with NESSIE (called NESSIE+DRESC in the table). The first column of the table gives the concerned design steps: *precompilation* or *NESSIE* for respectively the first and the second flow - in each case, the modeling time and the simulation time; *compilation* step, performed by DRESC in both flows.

As the modeling stage is not optimized in NESSIE, we have computed the design time with and without the modeling time (in the rows 5 and 6 of the table).

<table>
<thead>
<tr>
<th></th>
<th>Precompiler/NESSIE</th>
<th>Modeling</th>
<th>Simulation</th>
<th>Compilation 75 it/ 3 it</th>
<th>Total time with Modeling</th>
<th>Total time without Modeling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DRESC</td>
<td>NESSIE+DRESC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precompiler</td>
<td>NESSIE</td>
<td>2 hours</td>
<td>30 hours</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modeling</td>
<td></td>
<td>14 hours</td>
<td>1 hour</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation</td>
<td></td>
<td>16 hours</td>
<td>33 hours</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compilation</td>
<td></td>
<td>14 hours</td>
<td>3 hours</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.17**: Timing values of the classical flow (DRESC) and the integrated flow (NESSIE+DRESC)

### Discussion

From this table, we can conclude that the modeling time in NESSIE is not negligible and that, for this case study, it takes off the usefulness of our tool. However, if we remedy this issue (as we will discuss further), we see in the last row of the table that the flow integrating NESSIE beats DRESC with a gain of 80%.
This conclusion must be seen in perspective of more complex case studies. Indeed, we show in Figure 4.53 and 4.54 the impact of respectively the body loop complexity and the loop size on the simulation time in NESSIE (in red) and in DRESC (in blue).

In Figure 4.53, we present the time (in minutes) for a given matrix size, 64x64, and for different unrolling factors. The time is the accumulated time for four different CGA architectures with shared register files: 2X2 mesh, 3X3 mesh, 4X4 mesh and 4X4 mesh+. As we show, when we make the loop structure more complex, the compilation time rapidly explodes in DRESC, when in NESSIE, we keep a constant simulation time.

In Figure 4.54, we give the simulation time (in minutes) for a given unrolling factor equal to 8 and a fixed CGA architecture (3X3 mesh with shared RF). We compare the time spent in both tools for different matrix sizes (16x16, 32x32, 64x64 and 128x128) which condition the number of iterations in the loop, keeping the same body loop structure. This figure shows that the compilation time is constant in DRESC and that the simulation time grows slowly in NESSIE when the matrix size increases. Indeed, DRESC performs the scheduling of the loop based on one iteration as the same scheduling scheme is applied for all iterations of the loop. As a consequence, increasing the number of iterations of the loop does not increase the compilation time. On the contrary, NESSIE considers the entire loop and the mapping time is thus linked to the loop size which grows with the matrix size. However, if the simulation time increases with the matrix size, we achieve a gain of more than 80% even for big matrices (128x128). Moreover, Figure illustrates this gain for a 3x3 CGA array which is not the most time expensive in DRESC.

These two figures highlight once more that the design time can be highly reduced by using NESSIE, but that we need to cope with the modeling time issue. The use of an automatic generator of scenario is a solution to this limitation that is presented in Chapter 6 presenting the future work.

### 4.8 Conclusions

The goal of this chapter was to confront NESSIE with a first external case study, representative of a topical design problem. In particular, we have targeted the design of a reconfigurable processor ADRES and its dedicated design flow, developed at IMEC, applied on a typical telecommunication kernel, the matrix multiplication.

In order to put our tool to the test, we have formalized the external design flow and study the feasibility to model the case study in NESSIE at a low level of abstraction. It has shown that NESSIE is sufficiently flexible to target this design problem.

We have defined 75 solutions constituting the design space and chosen four criteria for the simulation. The results that have been generated by NESSIE have been compared...
4.8. CONCLUSIONS

**Figure 4.53:** Comparison of the simulation time in NESSIE (red) and in DRESC (blue) for different unrolling factors, four CGA architectures with shared RF (2x2 mesh, 3x3 mesh, 4x4 mesh and 4x4 mesh+) and a fixed 64x64 matrix size.

**Figure 4.54:** Comparison of the simulation time in NESSIE (red) and in DRESC (blue) for different matrix sizes and a fixed unrolling factor (8) and CGA (3X3 mesh shared RF).
with the criteria estimated in the external flow by the DRESC compiler to evaluate the accuracy of the multicriteria estimation in our tool.

We have completed this discussion with the study of the prediction capability of NESSIE based on cycle and effective density constraints and compare the pre-selected solutions with those identified by DRESC.

Finally, we have discussed the design and modeling time in both tools to analyze the usefulness of NESSIE integrated in such external design flow and have seen that our tool is of main interest provided that a clever generation and exploration of scenarios is introduced in the framework.

More globally, from the formalization and the analysis done in this chapter, we can conclude that:

1. For the main case study, we have chosen to experiment our tool NESSIE on a sufficiently representative system but not to big to be able to make analysis, developments and discussions. Thanks to the available information we had from IMEC, we have been able to model a low level decision tool called DRESC. Via results comparisons and analysis, we have been able to conclude that NESSIE is able to give a close trend for the criteria estimations what has as main consequence that we are able to preselect interesting solutions based on the NESSIE simulations.

This answers the questions $Q_2$ and $Q_3$ asked in Chapter 1:

- *Is the mapping core sufficiently flexible?* yes.
- *Does it perform realistic simulations?* yes.

However, two major observations have also been done. Indeed, from the DRESC point of view, some non-logical points have been detected. They are a consequence
4.8. CONCLUSIONS

of the compiler parameters which can be changed by the designer. On the contrary, these points have been identified logically in Nessie. From the Nessie side, points have also been under-evaluated. This is due to overhead linked to the mapping policy in Nessie. We have pointed up the cases where this overhead cannot be neglected or when reliability of the solution is not guaranteed.

This answers the question Q7 asked in Chapter 1:

- *What are the limitations of the tool?* The mapping policy which is simple and based on independent routing and allocation.

2. As the modeling and estimation accuracy of low level systems and tools are more critical than high levels of abstraction, we can reasonably suppose that our tool would be efficient if applied on case studies modeled at a higher level. This is what we will illustrate in the next chapter.

3. We have formalized the DRESC flow and the integrated flow where NESSIE is used as a prediction tool for DRESC. This formalization and the timing analysis have shown the importance to have automatic tools enabling the interface between different design tools. The same way, we also see the importance of sharing standard description languages both for SW and HW. The problem is more obvious concerning the description of the Petri Nets for which a ddg of the application is needed. At low levels of abstraction, such a graph can be quickly too painful to describe if not disposing of an automatic tool which translates the application into the Petri Net or the ddg.

This answers the questions Q1, Q6, Q8 asked in Chapter 1:

- *How easy is it to model an application, a platform in NESSIE?* Low level application representation in Petri Networks is tedious but feasible. Platform modeling is more direct but the memorization management is a key point to take into account.

- *How the tool can be integrated into an existing toolchain or coupled with existing tools?* We need to create an interface with standard languages and an automatic translation between models.

- *Which modules or further development should be implemented to improve the tool?* An automatic generator of scenarios and a graphical interface to enter the application and platform models.

4. The mapping analysis of both tool has allowed us to understand why NESSIE is able to give a correct trend compared to DRESC results which is based on a Simulated annealing metaheuristic, a much more complex scheduling policy than Nessie. As Nessie, DRESC will map the operations such that the communication between FU is minimized. As we have seen with the ADRES viewers, it has implied that most of the operations are mapped by DRESC on the first FU, first rows of the CGA. This is also the case in Nessie which also chooses the first blocks as candidates, even
if the goal here is not fundamentally to reduce the communication. However, as the goal in DRESC is to find a mapping following the modulo scheduling principle, where the idea is to be able to have the same mapping pattern for each iteration of the loop, the scheduling algorithm is more complex and take more time than Nessie with timing increasing exponentially with the complexity of the solution while the simulation time in Nessie increases more linearly.

5. During the analysis of this integrated flow, we have discussed about the gain we can bring to the classical flow regarding the design time and the system performances (cycles, energy consumption, area, ...). If quantitative values are difficult to obtain, we have however made observations that give a good feeling on the utility of Nessie as a prediction tool to reduce the design time and explore wider the solutions space. However, this discussion has pointed out that a smart multicriteria exploration module should be added to our tool in order to improve the quality of the preselected solutions and prevent the user having to feed NESSIE with its own chosen scenarios. This is the purpose of automatic generators of scenarios for which we present the future developments in Chapter 6.

This answers the question Q4 and Q5 asked in Chapter 1:

- Is the exploration policy efficient? We could improve the exploration of solutions by integrating a smarter module.
- How fast are the simulations compared to other tools? Simulations in NESSIE are very fast compared to classical tool, but we need to reduce the modeling time.

In the next chapter, we illustrate the prediction capability of Nessie on a higher level case study which touch a design problem really representative of way new technologies and new design tools are considered in the industry. It concerns the 3D stacking technology applied on a video decoder application.

Bibliography


Chapter 5

Case study 2 : 3D Stacking Paradigm

Abstract

This chapter is dedicated to the validation of NESSIE on the second case study, defined by the 3D stacking design of a MPSoC platform applied on a video decoder application. The validation is based on a higher modeling level of the system compared to the case study of Chapter 4 and on the estimation of the power consumption of the wires constituting the NoC of the platform. First, a presentation and a formalization of the problem is done. Then, we show the feasibility and the limitations to model this case study in NESSIE. The modeling is followed by the presentation of 24 scenarios and three data resolutions that compose the exploration space that have been simulated in NESSIE and generated in the external design flow, used at the NTUA (National Technical University of Athens). The results obtained in both flows are compared to analyze the estimation and modeling capability of our tool. We have completed this analysis by a study of the prediction efficiency of NESSIE. Therefore, best identified solutions have been pre-selected and compared to the classical flow. The results have shown that NESSIE is able to predict with less than 30% of error the performances of the system. Finally, we show that NESSIE reduces significantly the design time compared to the other flow.

5.1 Introduction

In previous Chapter, we have put NESSIE to the test by modeling at a low level of abstraction the ADRES processor for a typical application kernel, a matrix multiplication. It has shown the modeling and prediction capability of our tool, but also its limitations. In present Chapter, we have chosen another case study which will be modeled at a higher
level of abstraction. Through this case study, we will show that NESSIE is sufficiently flexible to target two different design problems and face other modeling considerations.

The problem we have considered concerns the design of 3D stacked MPSoCs. The 3D stacking is an emergent technology that enables the three dimensional integration of layers by stacking vertically multiple dies, interconnected together via so-called through-silicon-vias (TSV). This technology enables the reduction of the total wire length and thus the wire delays and energy consumption which is critical in current embedded systems. This ability to stack layers in an unique chip offers new multiple degrees of freedom for the design of SoC compared to classical 2D integrated circuits: number of layers, floorplanning of components inside the layers, interconnection between the layers,... These degrees of freedom increase considerably the design space.

Currently, classical design flows are used to design 3D chips. However, they are not able to deal with so many DoF with respect to multiple constraints, what consequently leads to numerous iterations in the flow and costly design time. Regarding these considerations, such a design problem is a good candidate to test the potential of our prediction tool NESSIE.

In the previous work[1], a first study of a 3D MPSoC system, applied to a video coding application, has been done. This study consisted in exploring several solutions based on the estimation of the power dissipation of the network-on-chip interconnecting the components of the platform. This study showed the ability of our tool to model and quickly explore several scenarios of the system. However, the simulations were not based on realistic data and no comparisons with existing design flow were done.

In this work, we make a deeper analysis of the problem based on an external existing design flow[2] used at the National Technical University of Athens (NTUA). The validation will be done through the following steps:

- study the modeling capability of NESSIE for the 3D design problem. This is the purpose of Section 5.3 ;

- analyse and discuss the prediction ability of our tool by generating results in NESSIE and in the external flow on several variants of the system, and compare these results. This will be done in Section 5.4 ;

- quantify and discuss the design time when using NESSIE as a prediction tool upstream from the classical flow. This is presented in Section 5.5.

As an introduction to these different parts, we present in the next section, the external design flow and the 3D case study we have modeled and explored.
5.2 Description of the 3D design problem

This Section is divided in two parts. First, we present the external design flow. Second, we detail the platform and the application that form our case study.

5.2.1 The 3D design flow

The classical top-down flow used to design 3D stacked SoC is shown in Figure 5.1. It is composed in two main parts: the 3D stack generation and the system prototyping.

3D stack generation

The inputs of the tool chain are a communication graph describing the application and a netlist of the platform, both described in XML. First the netlist of the platform is partitioned to extract the different components that will be allocated on the dies. Generally, this partition is done such that the number of
connections between the architecture components is minimized. The number of dies is chosen at this level. The designer is assisted by the tool that checks the geometrical properties of the stacks (some layer shapes may create mechanical instabilities in the 3D stack). The allocation of the “partitions” is performed by the designer who assigns each part to the different dies and chooses the technology for each layer. A prototype of the stack is then built by ordering the dies, and the designer selects a 3D interconnection technology (TSV, wire bonding,...).

Once the stack generation has been performed, estimation of the performances of the system can be done. The tool is typically linked to an extended version of the WormSim NoC simulator which simulates the traffics of the interconnection network and generates the transfer traces between the various IP blocks and routers. A first evaluation of the chip area is also performed.

Analysis: this stage enables a first estimation of the performances of the system before its physical implementation. In this design step, the designer explores the way the architecture and the application will be distributed on the 3D layers. It is based on a previous designer’s decision who has a priori fixed the structure of the application and the architecture, based on its sole experience. Moreover, the choice of the 3D stacking is also performed by the designer, although it is constrained by the tool. Consequently, several iterations are needed to explore solutions which are time expensive, as the generation of one stacked chip can take several hours.

3D system prototyping

When this first estimation of metrics has been done and correspond to the specifications, the second design step performs a physical prototype of the 3D system by incorporating physical design information. In this stage, 2D physical tools are used to perform the floorplanning, the placement and routing of each die in a conventional planar fashion. Of course, constraints linked to the TSV interconnections are included during this process. The different steps (floorplanning, placement and routing) are performed by the Cadence SoC Encounter tool [3]. Based on the output of the tool, more accurate estimation of the power and the performance of the inter-IP communication is done. The wire-length is also evaluated precisely.

Analysis: this design step enables a more accurate estimation of the criteria, and an estimation of the energy consumption of the NoC. Although the flow is built to limit the number of iterations at this level, criteria like the energy consumption are evaluated late and could necessitate to change the choices made previously.

Regarding this analysis of the flow, design time could be reduced if a faster exploration of the solutions were performed, based on the critical criteria, to feed in a “right the first time” way the tool chain presented in Figure 5.1. Considering a higher decision step, this exploration should also include more degrees of freedom to converge to optimal solutions.
5.2. DESCRIPTION OF THE 3D DESIGN PROBLEM

(different type of application and architecture structures) which is not the case in this top-down flow.

This is the purpose of NESSIE and this is why we have chosen this case study as an additional validation problem.

In this work, we have limited the design space to the exploration of the floorplanning and layer distribution of a predefined MPSoC structure, based on the estimation of the power consumption of the NoC of the platform, as it is performed in the 3D stack generation design step of the external flow. From the side of the application, we have explored three scenarios, that we will detail in the next part of Section.

Another work, which is currently in process, is dedicated to the extension and management of a wider design space exploration based on more degrees of freedom. This includes the study and the integration of Multi-Criteria Decision Aids (MCDA) methods to NESSIE, which is not the purpose of this thesis.

We present in the following part the MPSoC architecture and the application that have been used for the validation of NESSIE.

5.2.2 Description of the case study

The case study has been taken from the paper [4] presenting the power dissipation of a network-on-chip interconnecting processors in a 2D MPSoC platform. In particular, the results are given for a video decoding application for three different data resolutions and three different application partitions. Based on the information given for this example, we have extended the problem to the 3D stacking to predict the impact of the 3D structure on the power dissipation of the NoC.

The platform

The 2D layout of the MPSoC architecture is shown in Figure 5.2. The platform is composed of several computational blocks and memories, interconnected together through an Arteris NoC [5].

At the left and right sides of the platform, there are six instances of the ADRES processor\(^1\). They are used for the execution of the data intensive part of the application. On the top of Figure, there are a FIFO that handles the input (compressed) and output (uncompressed) of data, two instruction memories (L2Is1 and L2Is2\(^2\)) and an ARM processor, used for the control and the audio processing.

On the bottom of Figure, there are two data memories (L2D1 and L2D2) and an EMIF (external memory interface).

All these hardware components are linked to a CA (communication assist) which control the data transfer from the nodes to the NoC. It can act as initiator or target of a data

\(^1\)We refer to Chapter 4 for details of this processor.

\(^2\)L2 indicates that these memories are use at the L2 cache level.
transfer through the *Network Interface Units* (NIU) that enable the protocol conversion between third party communication protocol like AHB, OCP, AXI and the NTTP (NoC Transaction and Transport Protocol). These NIUs are the small rectangles between the components and the NoC on Figure 5.2.

As shown in Figure, the NoC is composed of two paths: one for the request from an initiator to a target (plain line-white routers) and one for the response from the target to the initiator (dashed line-grey routers). In addition, the six ADRES nodes are connected through a separated instruction NoC to the instruction memories. The topology of the data NoC is a fully mesh composed of 2x2 switches (routers). This implies that the maximum distance to transfer data between two nodes is 2 HOPs (number of routers the packet goes through).

In this case study, the communication links are 32 bits wide and the operating frequency is 150 MHz. This offers maximum throughput of 600MB/s per link.

**The application**

The application we have considered is an AVC/H.264 video decoder. Figure 5.3 describes schematically its structure. In this figure, we find the different functions that compose the application (in white) and their inter-data dependency: the motion estimation (ME)
for full, half and quarter pixel, the motion compensation (MC), the intra prediction block (Intra), the error difference, the discrete cosine transform and quantization (DTC/Q), the inverse DCT and inverse quantization (IDCT/IQ), the deblocking filter and the entropy encoder. Buffering or memorization needs are also explicitly represented, in grey.

The numbers on the edges represent the throughput requirements between each functional block in bytes per macroblock [B/MBr], a macroblock being a 16x16 pixels data structure encoded with 384 bytes.

For this application, three data resolutions may be considered: CIF, 4CIF, HDTV. These impact the throughput requirement for the NoC data transmission (in MBytes/s).

The power dissipation model

As previously explained for this case study, we exclusively consider the power dissipation of the network-on-chip. Closed-form expressions are given in the paper [4] for each element constituting the communication NoC (switches, links, NIUs). For the switches and the NIUs, this power is composed of an idle and a dynamic term as expressed in the equation 5.1 where the $P_{idle}$ is a constant ($c_1$ or $c_3$ in the eq 5.2 and 5.3) and the $P_{dyn}$ depends on the activity $A$ of the component, i.e., the actual traffic percentage compared to the NoC bandwidth (600 MBytes/s for 150 MHz frequency); and the payload size, taken into account in the constants $c_2$ or $c_4$. 
\[ P_{tot} = P_{idle} + P_{dyn} \]  
\[ P_{\text{switch}} = c_1 + A \times c_2 \]  
\[ P_{NIU} = c_3 + A \times c_4 \]

The power dissipation of NoC’s links can be expressed with the equation 5.4 where \( l \) is the length of the link, \( \omega \) is the number of wires in a link, \( C \) is the total switching capacitance, \( f_{\text{NoC}} \) is the frequency of the NoC, \( V_{dd}^2 \) is the power supply voltage, \( A \) is the activity on the link and \( p_{\text{toggle}} \) is the bit toggle probability\(^3\).

\[ P_l = l \times P_s = l \times \omega \times C \times A \times f_{\text{NoC}} \times V_{dd}^2 \times p_{\text{toggle}} \]  

Regarding this information, we explain in the next section how we have modeled the case study and which solutions we have chosen to explore.

### 5.3 Modeling of the case study

In this Section, we explain what and how we have modeled the architecture of the MPSoC in 3D, the application, schematically presented in Figure 5.3, and the power consumption criterion. Then we present the degrees of freedom we have chosen and the solutions we have explored.

#### 5.3.1 The models

Regarding the information given in the previous section, we have decided to model the case study on two levels of abstraction in our tool, as shown in Figure 5.4.

**AL 0**: this first level is required in NESSIE and is composed of a unique primitive representing the "black box" of both the application and the platform. For this case study, the architecture primitive is a generic MPSoC. The application is a generic video decoder.

**AL 1**: the second level describes the inner structure of the black box. From the side of the MPSoC, the primitives are the components defined in Figure 5.2\(^4\): ADRES, ARM, EMIF,... routers, wires, ... From the side of the application, the primitives are the functions composing the decoder: motion estimation, error difference, DCT/IQ,...

We successively present the modeling of these primitives and the way the solutions are defined and explored based on the power criterion further in Section.

---

\(^3\)Indeed, when a bit does not change on a wire, no energy is consumed as there is no toggle

\(^4\)See section 5.2.2.
5.3. MODELING OF THE CASE STUDY

Figure 5.4: Levels of abstraction (AL) modeling the video decoder and the 3D MPSoC structures
CHAPTER 5. CASE STUDY 2 : 3D STACKING PARADIGM

The MPSoC platform

We have restricted the modeling of the architecture to the components implied in the computational-intensive part of the application: the ADRES, the EMIF, the FIFO and the memories. The control parts of the system, the ARM processor, has not been considered. Moreover, in this work, we are interested in the estimation of the power consumption of the wires. The components that have been considered in the NoC are thus: the wires and the routers. The NIU have not been used.

Below, we detail the different primitives and the way they have been modeled. They are represented in Figure 5.5 in a planar representation.

- ADRES processors: shown in orange in Figure 5.5, this component is defined by a primitive for each ADRES instance. Indeed, they differ by their compatibility list, ie the functions they will be able to execute. We will detail this in the application.

---

\(^5\)Values for these components had been evaluated in [4]. The interested reader can refer to these results for more details on this part.
part.

- FIFO and EMIF: they are both represented in red in Figure. The FIFO acts as a buffer and is used to memorize data and to interface the application with the inputs and outputs. The EMIF is the external memory interface used for the transaction with the FIFO. For both components, as NESSIE does not perform automatic memorization on external memory, we need to force the storage of data and include a computing state in the primitive. This implies the definition of pseudo-tasks in the Petri Network.

- Data and Instructions memories (L2D1, L2D2, L2Is1, L2Is2): shown in yellow, these blocks are memorization primitives as the FIFO/EMIF and will store different amounts of data depending on the application structure. It will be executed by the architecture.

- Switches: the green blocks in Figure are used to route data from one node to another. They can be modeled as a transmission block with several input/output ports. Several instances of this primitive will be defined in each platform structure (four in the data NoC and 1 in the instruction NoC).

- Wires: shown in blue (for wires connecting nodes to switches) and in green (for inter switches wires), they are obviously transmitting primitives. Moreover, contrarily to the other components, the behaviour associated to this component will contain the power consumption relation that will allow NESSIE to evaluate the criterion for the entire system. This criterion depends on several parameters, detailed in the equation 5.4. The wire length $l$ is a variable parameter which depends on the structure of the architecture that will be modeled and that must be defined for each wire of the NoC. We explain below how we explore the different possible wire length values in NESSIE.

**Wire length:** in NESSIE, degrees of freedom can be defined to explore parameters values during the simulation of a system. This is done by the use of a list of values or via a sweep from a starting value to an ending value with a user-defined step. When several parameters have been defined by multiple values, NESSIE simulates the scenarios and calculates the criteria for all the combinations of these parameters.

For our case study, the definition of a 3D structure is correlated with the wire length between each node of the MPSoC. For a given 3D structure, the values assigned to all the wires of the NoC must be coherent and are inter-dependent. We can thus not make use of the DoF capability of NESSIE for this parameter, otherwise, lot of results that will be produced will not make sense.

We had two possibilities to define different 3D structures and thus different wire length for each structure:

- first, create as many set of wire primitives (a set being composed of all the wires for a given MPSoC structure) as the number of architecture variants. It implies
to complete the simulation file of NESSIE with these new primitives for each new structure we want to explore and to add the length value for these primitives. This can be tedious. The advantage is that, once we have defined all the scenarios, we can launch all the simulations directly.

- second, create one structure representing the MPSoC with one set of wires composing the NoC. To explore several solution, the user must launch a new simulation for each scenario and replace the length parameters by new values. Less time is spent to define the scenarios, but we do not take advantage of the possibility of our tool to simulate in one shot multiple solutions.

In practice, we have used the second solution, but same results would be obtained in the other case.

Apart the wire length, other parameters must be defined to evaluate the power consumption of the wires. Moreover, rules must be given to NESSIE for the dynamic calculation of the relation during the mapping performed by our tool.

**The parameters:** in the equation 5.4, there are five constant parameters: the number of wires per link, the capacitance, the frequency, the supply voltage and the toggle probability. Their values are given in the table 5.1.

On the contrary, the activity $A$ depends on the number of data transmitted on the links which is related to the application scenario and the data resolution and the way NESSIE performs the mapping. It is equal to the amount of data transmitted on a link divided by the total bandwidth. This is equivalent to the latency resulting of the data transmission on the link. For example, if the BW=600MB/s and the amount of data to transmit=100MB, then the latency=1/6.

To dynamically calculate the activity, we can use an *integrate* time-dependent rule which will result in the multiplication of the equation 5.5 by the latency, equivalent to the activity. To deduce the global power consumption of the platform, an *additive* combination rule has been defined.

$$P = l \times \omega \times C \times f_{NoC} \times V_{dd}^2 \times p_{toggle}$$ (5.5)

In the next part, we explain how the application has been described in NESSIE.
5.3. MODELING OF THE CASE STUDY

The video decoder application

In the paper [4], three application scenarios are proposed. They differ by the way the functions and the data are split on the ADRES processors of the MPSoC platform. These scenarios implicitly fix the mapping as the designer foresees the execution of tasks on specific hardware components. To model the application represented in Figure 5.3 independently of the architecture, we have defined three different sets of software primitives for each scenario, varying by the type and the data size handled by the place. Each set of primitives is used in a distinct Petri Network.

From the side of the architecture, it results in a different compatibility list for each ADRES processor.

To model the memorization of data and their transfer to buffers (FIFO, EMIF) and dedicated memories (data or instruction), we include pseudo-primitives to enable the explicit mapping on these hardware primitives by NESSIE\(^6\).

In the next part of Section, we present the different scenarios that have been modeled and explored in NESSIE, for both the architecture and the application.

5.3.2 The scenarios

The design space that has been simulated is composed of 72 solutions:

- eight MPSoC structures, including the 2D architecture presented in the paper [4] and seven 3D variants.
- nine application structures describing three split variants, and three different data resolutions (CIF, 4CIF and HDTV).

These solutions are presented below, successively for the platform and the application.

The architecture variants

To have a significative observation of the impact of the 3D structures compared to 2D, we have consider solutions composed of 2 or 3 layers. In each layer, we have chosen to floorplan the components such that realist physical design could be performed. These choices have thus been done in collaboration with the NTUA. It has resulted in eight scenarios.

1. one layer: this is the 2D MPSoC structure taken from the paper ref.
2. two layers: (1) A1→A6, L2D1, L2D2, EMIF, FIFO - (2) L2Is1, L2Is2
3. three layers: (1) A1→A6, L2D1, L2D2, EMIF, FIFO - (2) L2Is1 - (3) L2Is2
4. two layers: (1) A1→A6, FIFO - (2) L2Is1, L2Is2, L2D1, L2D2, EMIF

\(^6\)We refer to Chapter 3 for the explanation.
5. three layers : (1) A4, A5, A6, EMIF, L2D1, L2D2 - (2) A1, A2, A3, FIFO - (3) L2Is1, L2Is2

6. three layers : (1) A1→A6, EMIF, FIFO - (2) L2Is1, L2D1 - (3) L2Is2, L2D2

7. three layers : (1) L2D1, L2D2, EMIF, FIFO - (2) L2Is1, A1, A2, A3 - (3) L2Is2, A4, A5, A6

8. three layers : (1) A2, A5, L2Is1, L2Is2, EMIF, FIFO - (2) A1, A4, L2D1 - (3) A3, A6, L2D2

The wire lengths that have been used to define these variants are given in the Appendix B. We have based these values on typical examples designed in the external flow presented in Section 5.2.1.

The application variants

The three scenarios that we consider are named data split, functional split and hybrid. Three Petri Networks have been described for each split to model three different data resolutions, which are the CIF, the 4CIF and the HDTV.

- In the Data split scenario, the input stream is divided into 6 equal substreams, each processed by a dedicated ADRES processor. All the functional blocks presented in Figure 5.3 are included in the compatibility list of each ADRES processor.

- In the Functional split scenario, the ME computation is distributed to three ADRES (A1→full, A2→1/2, A3→1/4). The Intra, error difference, DCT/Q, IDCT/IQ and MC are in the compatibility list of A4. The deblocking filter is executable by A5. And finally, the A6 can execute the entropy encoder function. In this scenario, the whole stream is treated in all functions.

- In the Hybrid scenario, the ME computation (the more computational intensive task) is split on three processors dealing with three substreams, as in the data split scenario. The other functions are computed as explained in the functional split.

The data sizes assigned to the primitives of these scenarios for each resolution are given in the Appendix B.

Having shown the ability to model our case study in NESSIE and having defined the solutions we will explore, we present in the next section the results obtained by our tool and discuss its prediction capability on such a design problem. This will be completed, in Section 5.5 by a discussion of the design time gain achieved when using NESSIE upstream from the classical 3D stacking design flow.
5.4 Results and discussion

To put NESSIE to the test, we have proceeded in two steps:

1. we have compared the results produced in the two tools to evaluate the model accuracy and the mapping capability of NESSIE. Therefore, we have computed the wire power consumption for the 72 variants in both NESSIE and the design flow used at the NTUA. The comparison has been made based on the general trend of graphs obtained for all the scenarios and the three different resolutions. This is presented in the part 5.4.1.

2. in order to discuss the prediction capability of NESSIE we have experimented the preselection of most interesting solutions identified by our tool and compared them to the solutions that would have been chosen without the use of NESSIE in the classical flow. This is the purpose of the part 5.4.2.

5.4.1 General trend

Figures 5.6, 5.7 and 5.8 present respectively the power consumption of the wires, in $\mu$W, for each data resolution. The variants are given on the $X$ axis : from left to right, we find the architecture variants for the data split solutions ($Dv_i$), then the functional split ($Fv_i$) and finally the hybrid scenario ($Hv_i$).

The green curve gives the results estimated by NESSIE. In orange, we find the results simulated in the external flow, with the Cadence SoC Encounter tool.

![Figure 5.6: Wires power consumption of eight MPSoC variants and three video decoder scenarios for the CIF resolution](image-url)
CHAPTER 5. CASE STUDY 2 : 3D STACKING PARADIGM

Figure 5.7: Wires power consumption of eight MPSoC variants and three video decoder scenarios for the 4CIF resolution

Figure 5.8: Wires power consumption of eight MPSoC variants and three video decoder scenarios for the HDTV resolution

Discussion: As we can see, the trends are really close for all resolutions. As we could expect, the 3D variants give better results regarding the wires power consumption than the 2D layout (variant v1). However, the curves do not exactly superimpose.
5.4. RESULTS AND DISCUSSION

Figure 5.9, 5.10 and 5.11 give the relative error for each point respectively for CIF, 4CIF and HDTV. The error goes until 30% for the HDTV in the hybrid split scenario what is reasonable.

This error is explained as follows. To feed the analytical expression used to calculate the wire power consumption, we have given Nessie several parameters. For some, we had exact values (e.g. $V_{dd}$, $f_{NoC}$) However, for the wires length and the data size, we did not have accurate values. Indeed, we have approximated these parameters based on partial information. The data amount exchanged between application places has been based on bus loads presented in the paper [4] which gives an idea of the data amount exchanged between switches and platform nodes.
The wire lengths have been deduced based on classical 2D layout and typical 3D inter-layer distance.
These errors show the importance of the model and the parameters accuracy and the necessity to give the designer a confidence interval with the estimated results.

However, NESSIE is overall dedicated to the fast prediction of performance to allow the designer to preselect a small set of interesting solutions. This is discussed in the next part.

5.4.2 Prediction

To discuss the prediction capability of Nessie, we have selected, for each data resolution, the best four solutions estimated by NESSIE\textsuperscript{7}.
These solutions are compared with the four best variants that have been estimated in the

\textsuperscript{7}Ordered bar graphs are given in the AppendixB for all the generated solutions.
CHAPTER 5. CASE STUDY 2: 3D STACKING PARADIGM

Figure 5.10: Relative error (in %) between NESSIE and NTUA results for the wires power consumption- 4CIF resolution

Figure 5.11: Relative error (in %) between NESSIE and NTUA results for the wires power consumption- HDTV resolution
5.4. RESULTS AND DISCUSSION

Table 5.2: Comparison of the preselected NESSIE solutions with the best solutions identified in the 3D external flow for the CIF resolution

<table>
<thead>
<tr>
<th></th>
<th>CIF</th>
</tr>
</thead>
<tbody>
<tr>
<td>NESSIE</td>
<td>Cadence</td>
</tr>
<tr>
<td>F v5</td>
<td>F v5</td>
</tr>
<tr>
<td>F v7</td>
<td>F v7</td>
</tr>
<tr>
<td>F v4</td>
<td>F v4</td>
</tr>
<tr>
<td>H v5</td>
<td>H v5</td>
</tr>
</tbody>
</table>

Classical flow by a more accurate tool.

The comparison is shown in the tables 5.2, 5.3 and 5.4 for respectively the CIF, 4CIF and HDTV. In each table, the solutions are classified from the best to the worst.

A first observation shows that for all resolutions, NESSIE identifies the same best solutions, that are the functional split scenario with respectively the 3D architecture variant 5 and 7. These corresponds to architectures with 3 layers with ADRES processors split into two groups on successive layers. These are reminded below.

- (1) A4, A5, A6, EMIF, L2D1, L2D2 - (2) A1, A2, A3, FIFO - (3) L2Is1, L2Is2
- (1) L2D1, L2D2, EMIF, FIFO - (2) L2Is1, A1, A2, A3 - (3) L2Is2, A4, A5, A6

In addition, other observations can be made:

For the CIF resolution (table 5.2): NESSIE predicts exactly the same solutions than the external flow.

For the 4CIF resolution (table 5.3): the preselection is globally efficient, however, we see some differences.

- Three solutions on the four pre-selected by NESSIE are also identified by the NTUA lower estimation tool. In particular, the two best are exactly the same.

- The fourth solution is different: F v6 in NESSIE compared to H v5. The corresponding relative error is shown in Figure 5.10: we see that the variant Fv6 present an error >25%, which explained that it is not detected in Cadence. For the variant Hv5, the error is around 15%, what is quite reasonable and shows that NESSIE is not far of detecting this solution.

They correspond to 3D variants with 3 layers. The first favors the wire length reduction between the computational nodes and the memories, that are allocated on different layers. The second favors the communication between the computation nodes by splitting them on two different layers, and reduces the links also between the three first ADRES with the data memories. They are given below.
Table 5.3: Comparison of the preselected NESSIE solutions with the best solutions identified in the 3D external flow for the 4CIF resolution

<table>
<thead>
<tr>
<th></th>
<th>NESSIE</th>
<th>Cadence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best solutions</td>
<td>F v5</td>
<td>F v5</td>
</tr>
<tr>
<td></td>
<td>F v7</td>
<td>F v7</td>
</tr>
<tr>
<td></td>
<td>F v4</td>
<td>H v5</td>
</tr>
<tr>
<td></td>
<td>F v6</td>
<td>F v4</td>
</tr>
</tbody>
</table>

Table 5.4: Comparison of the preselected NESSIE solutions with the best solutions identified in the 3D external flow for the HDTV resolution

<table>
<thead>
<tr>
<th></th>
<th>NESSIE</th>
<th>Cadence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best solutions</td>
<td>F v5</td>
<td>F v5</td>
</tr>
<tr>
<td></td>
<td>F v7</td>
<td>H v5</td>
</tr>
<tr>
<td></td>
<td>H v5</td>
<td>F v7</td>
</tr>
<tr>
<td></td>
<td>H v7</td>
<td>H v7</td>
</tr>
</tbody>
</table>

For the HDTV resolution (table 5.4): we see that the two tools predict exactly the same solutions for refinement. However, apart the best one (SOL1→F v5) and the worst (SOL4→H v7), they thus not classify them in the same order.

As a conclusion to these results, we have shown that NESSIE is able to predict solutions with a high degree of confidence compared to estimations done by a much lower simulation tool. The differences that have been identified have been explained and are due to the approximations included in the models.

In addition to this conclusion, we discuss in the next section of Chapter, the design time gain that we achieve by using NESSIE upstream from the external design flow, instead of the sole classical flow.

5.5 Design time vs performance estimation

The design time values have been quantified based on the simulation time spent to simulate 24 solutions in NESSIE, on an estimation of the time spent to design the variants in the external flow\(^8\) and on a quantification of the modeling time to define the simulations in

\(^8\)Approximation of this data based on NTUA information.
NESSIE.
The different timings are summarized in Figure 5.12. We have represented the integrated flow where NESSIE is used upstream from the lower 3D stacking tool chain. At the beginning of the flow, a set of solutions (application and architecture variants) are chosen and modeled in the input XML simulation file of our tool ($\Delta T_1$ and $\Delta T_2$). NESSIE generates results for all the solutions ($\Delta T_3$). A set of four solutions are pre-selected and injected in the lower tool chain for refinement and physical design ($4 \times \Delta T_4$). Finally, possible iterations are included and add extra timings ($\Delta T_5$).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig5_12.png}
\caption{Timings of the flow integrating NESSIE to the external 3D stacking tool chain}
\end{figure}

**Timing analysis** The estimations done to evaluate the design time gain give the following results:

- Contrarily to the main case study presented in Chapter 4, this one required less modeling effort as the information needed to describe the system was directly available and the simulations were done at a higher level of abstraction. We estimate this effort in a few **hours**.

- The simulations have globally taken **10 minutes** for all the variants. This time is very small and shows that the simple policies implemented in NESSIE are particularly interesting in such example. This is explained by the fact that we considered the system at a higher AL than the previous case study and that the Petri Networks are constituted of no more than 50 places and the platform is composed of maximum 40 blocks.
• The time spent in the lower design steps is much greater than NESSIE’s simulations. Typically, it can take \textbf{hours} to produce one physical prototype of the platform. This time must be multiplied by the number of solutions tested in the flow (four in our example).

We take advantage of the use of NESSIE by reducing the number of iterations and thus the number of solutions to refine, that can significantly increase the global design time. Comparatively, if NESSIE were not used, at least twice the number of solutions could be explored in the external tool chain, what implies an equivalent increase in the design time which can take several days.

This analysis shows that NESSIE is really useful to reduce design time of an existing classical flow and to predict the final performances of the system with a satisfying degree of confidence. However, it requires the designer to have sufficient information to model the design problem. In our case study, it was the case although the exploration could be extended to more degrees of freedom.

5.6 Conclusion

Through this chapter, we wanted to demonstrate that NESSIE is able to target different design problems and identify its prediction capability, but also its limitations when modeling a system at a higher level of abstraction. Therefore, we have formalized an external design flow, used at the NTUA, and a specific MPSoC platform applied to a video decoder application. We have studied the modeling capability of NESSIE for the 3D stacking problematic and produced results on a significant amount of solutions.

These solutions have been compared to results obtained in the lower tool chain to evaluate NESSIE.

The conclusions of this chapter are the followings:

1. This example has illustrated a typical trend in classical flows which consists in performing an \textit{a priori} mapping of the application onto a pre-defined platform. To model these two parts of the system \textit{independently}, we have thus artificially separated the two descriptions. This has implied to define different tasks and compatibility lists to represent the application scenarios. Such issue is typically encountered in designs where systems are represented at a high level of abstraction where computation nodes are able to execute application that can be considered at different levels (functions, tasks, operations,...).

This answers the question \textit{Q1} asked in Chapter 1:

\begin{itemize}
  \item \textbf{How easy is it to model an application, a platform in NESSIE?} NESSIE allows to model systems at a high AL. We have been able to model a different design problem.
\end{itemize}
2. Contrarily to the first case study, we have based the criteria estimation of the solutions on the evaluation of a *closed-formed model* based on *several parameters*, in particular for the wire power consumption.

This has shown that, in its current version, NESSIE is not totally adapted to the exploration of 3D stacking platforms as the definition of parameter DoF is not convenient to define realistic platform. However, this problem could be resolved by improving our framework by generating automatically the scenarios and facilitating the data input description for example.

In addition, in our study, we have restricted the simulation to the wires and to the floorplanning of a predefined MPSoC structure on 3D layers, although the criterion could be extended to the nodes of the platform and more degrees of freedom could be considered (e.g. types and number of components).

This highlights the importance of getting models to explore automatically solutions, which is of main concern in todays embedded system design. This bottleneck is particularly present when considering new technologies and design tools.

This answers the question *Q4* and *Q7* asked in Chapter 1:

- *Is the exploration policy efficient?* NESSIE does not allow to add constraints on the architecture, what limits the design space definition and thus its exploration.
- *What are the limitations of the tool?* This joins the previous answer.

3. We have seen that the modeling effort is reduced when representing systems at a higher level of abstraction what is of main importance to integrate our tool in an existing design flow.

Moreover, the design time gain is significative if we compare the design performed thanks to NESSIE prediction and the external tool chain alone (few hours compared to days).

This answers the question *Q5* and *Q6* asked in Chapter 1:

- *How fast are the simulations compared to other tools?* The simulations are faster when modeling at high AL. Moreover, the modeling time is also decreased.
- *How the tool can be integrated into an existing toolchain or coupled with existing tools?* The integration to existing tool chains would be easier in that case as we generally start from a less detailed graphical representation of the system.

4. Finally, this second case study has shown that NESSIE is flexible enough to target different design problems, at different levels of abstraction with a sensitive degree of accuracy and high design time reduction.

This answers the question *Q2* and *Q3* asked in Chapter 1:

- *Is the mapping core sufficiently flexible?* Yes.
- *Does it perform realistic simulations?* Yes.
Still, to overcome the limitations that have been identified and that could prevent efficient modeling, simulation and prediction in our tool NESSIE, different developments and improvements are required. The purpose of the next and last chapter is to present them in order to push NESSIE to a competitive framework. This will answer the Q8 of Chapter 1: Which modules or further development should be implemented to improve the tool?

Bibliography


Chapter 6

Conclusions and Future Work

Through the chapters 4 & 5, we have studied two different case studies that were good candidates to put NESSIE to the test. To validate our tool:

- we have formalized two external flows and two design problems at different levels of abstraction;
- we have studied the modeling capability of our tool and its exploration policy, applied on these different case studies;
- we have generated results and have compared them to references given by the external flows;
- we have compared the design time between the classical flow and a flow integrating NESSIE as a prediction front-end.

This study has led to the following conclusions for each case respectively:

The first case study has shown the ability of NESSIE to model a low level decision tool and to explore, with a high degree of accuracy, multiple critical criteria on a representative number of solutions.

- the results for the "cycles" estimation give a close trend between NESSIE and DRESC;
- NESSIE is able to detect interesting solutions that could not be identified by DRESC;
- the simulation time in NESSIE beats the compilation time of DRESC, which grows exponentially with the complexity of the algorithm, even on a simple application example such as the matrix multiplication.

The second case study has shown the ability of NESSIE to target a different design problem and to be able to model it.
contrarily to the first case study, we have introduced an analytical expression to estimate the critical criterion of the design problem. The flexibility offered by NESSIE has allowed us to model this expression to perform a dynamic estimation of the criterion during the mapping;

- the results obtained by our tool for a significant number of solutions have shown a close trend compared to results estimated by a lower level simulator (error under 30%);

- the simulation time in NESSIE beats the design time in the external flow of the NTUA : 1 minute compared to hours for one solution.

Through these two different examples, we have shown that NESSIE, although it has been built as a “generalist” tool, non-specialized to dedicated applications and platforms, is sufficiently flexible to model, explore and predict solutions with a satisfying degree of accuracy and at very different abstraction/decision levels.

However, we have also identified limitations in the current version of the tool that deserve some attention and motivate the future work around NESSIE.

6.1 Limitations of NESSIE

From the IMEC case study analysis: we have identified several limitations that have to be improved:

- modeling of pure memorization components is not supported. It implies to explicitly define memorizations steps in Petri Network. This makes the application structure dependent on the architecture structure what is conflicting with the principle of NESSIE.

Moreover, NESSIE is not able to cleverly manage the allocation of data in memories. Indeed, the location of the memorization blocks compared to the computation components is not considered and there is no maximum storage capacity associated to memories what enables the allocation of an unlimited amount of data.

- due to the simple mapping policy implemented in NESSIE, deadlocks often appear, in particular when computation components of the architecture are defined without memorizing state and for applications with a high degree of parallelism, as it is the case with the first study. If these deadlocks are detected, no solution is implemented to find an alternative mapping. For some design problems, it could prevent any simulation feasibility.

- this case study has shown the significant amount of time spent to define different Petri Networks in the XML input file of NESSIE. Indeed, from the data dependence
6.1. LIMITATIONS OF NESSIE

Graph of a loop to the generation of the Petri Network, we have to make a correspondence between operations names and numerical IDs and types of the places. Moreover, we need to deal with dummy nodes. This proprietary model definition has required an intermediate sheet representation which is error prone and highly tedious to manipulate (for the simple matrix multiplication, one iteration of the inner loop requires already around 70 places). This has obviously limited the number of solutions we have explored.

- In addition, to get a DDG representation of the application, starting from a C algorithm, we have used the sole automatic tool available which is strictly dependent on DRESC. This was the faster way to generate multiple loop solutions. The main drawback is that it has made us dependent on the external flow we have modeled, what is not the purpose of our methodology.

- Finally, regarding the available models to estimate the criteria (cycles, area, IPC...) and to describe the case study in NESSIE, we have explored solutions at the same level than DRESC. To reduce the simulation time and explore a much wider design space of solutions, it would be of main interest to model the system at a higher level of abstraction (we present concrete propositions in the section 6.3). This highlights the importance of getting costs models, which is one of the main bottleneck in design exploration.

From the 3D stacking case study analysis: we have encountered less limitations. However,

- We have seen that NESSIE is not so powerful to cleverly explore 3D architectures. Indeed, the simulation of solutions would include constraints that take into account realistic architectural considerations. However, this is not a major issue and we could resolve this problem in NESSIE.

- We have limited the study of the 3D stacking to the distribution on different layers of a predefined MPSoC platform and to the estimation of the power consumption of the sole wires of the NoC. Designers could better take advantage of the potential of NESSIE if the case study were extended to more degrees of freedom and multiple criteria (e.g. change the MPSoC platform). Therefore, we need to get models that allow the estimation of criteria for different hardware components and application. The same remark was done for the IMEC case study.

In addition, in both case study, we have not included the modeling of the control flow of the applications. This is not straightforward and has not been studied in this work. Moreover, nothing is foreseen to check the functionality correctness of the Petri Network.

All these remarks can be summarized as follows. In its current version,
NESSIE is keyed to

- quickly explore a reasonable amount of solutions, even at low levels of abstraction;
- estimate user-defined criteria faster than existing tools;
- offer flexibility to model different design problems;

NESSIE shows limitations for

- exploring a lot of solutions as it is tedious to feed the input file;
- checking the functionality/platform coherency based on realistic constraints;
- dealing with deadlocks, in particular for parallel intensive applications;
- managing memories;
- representing the control flow in applications.

Although these limitations have been identified, this work has shown, through the two case studies, that NESSIE has the potential to predict interesting solutions compared to the existing design flows for different targets. Moreover, both IMEC and the NTUA have shown a real interest for the use of NESSIE with their flow.

The figure 6.1 places NESSIE regarding the classical design tools. In green, the current version of the tool is represented, which improves the designed solution regarding the
6.2. COMPARISON WITH SOA TOOLS

performances. However, if the simulation time is small compared to classical tools, the
significant modeling time in the current version of NESSIE places the tool below existing
flows. In orange, we situate the future second version of our tool which should include
modules in order to cope with the identified limitations allowing to converge to better
final system’s performances and reduced design time.

Indeed, to be competitive, we need to add functionalities and develop further our tool.
Nevertheless, we had identified in the chapter 2 dedicated to the state-of-the-art, three
tools that deserve lot of attention regarding the current embedded system design. We have
chosen these tools as a reference to discuss the usefulness of NESSIE in order to decide
on the further works to perform, or not, on our original framework. The comparison of
NESSIE with Daedalus, GASPARD and SynDEEX is presented in the next section and is
based on the following criteria:

- the capability to explore automatically the design space (AE), based on the appli-
cation (SW) and/or the platform (HW)
- the multi-criteria simulation/exploration (MC Sim)
- the functional verification (FV)
- the synthesis capability (Synt)
- the multiple levels of abstraction (Multi AL)
- the ability to target several types of applications and architectures (SW/HW)
- the mapping policy efficiency and flexibility

6.2 Comparison with SoA tools

The table 6.1 summarizes the characteristics and the tools to which we compare NESSIE.
The symbols in the table have to be interpreted as :

++ : indicates that the tool implement this functionality which constitutes a main char-
acteristic of the tool.

+ : indicates that the tool is not powerful regarding this criterion, but that it implements
more or less efficiently the functionality.

- : indicates that the tool is not foreseen for this criterion or that it implements not
efficiently the related property.

From this table, we can conclude that:

- Concerning the exploration of solutions, we see in the three first columns that
NESSIE performs better than the reference tools. In fact, compared to these tools,
NESSIE allows to explore both the SW and HW parts of the system. However, as the scenarios generation is not automatic, this functionality can still be improved, what explains that it does not radically beat the other frameworks. From the multicriteria point of view, NESSIE enables the estimation of more criteria which are not hard-coded, contrarily to the other tools.

- If we look at the SW and HW columns (7, 8), we see that NESSIE performs better than the three reference tools, especially for the application modeling. Indeed, our generalist framework allows to model various kinds of applications and platforms while other tools rather target a specific type of system (multimedia applications and MPSoC platforms). However, SynDEx and GASPARD extend their framework to target also FPGA or ASIC architectures.

- Compared to the simple allocation and scheduling policy implemented in NESSIE, Daedalus and SynDEx use more complex policies (e.g. metaheuristics) that enable more accurate or efficient mapping, at the cost of time, obviously. But all these tools, including NESSIE, enable the description of the system at several levels of abstraction, even if NESSIE is probably more flexible and less constrained regarding the allowed levels.

- Finally, we clearly see that NESSIE is not dedicated to the functional verification and the synthesis, as it has not been built for this purpose. On the contrary, the three other tools include these functionalities. They enable automatic refinement to lower levels of abstraction, like the RTL, and automatic models transformations that enable functional verification.

These observations show that NESSIE is still in the race regarding the existing frameworks. Moreover, it highlights that our tool could be used as a complement to existing frameworks and that enabling interface with lower automatic refinement tools must be a research topic for the future work. Based on the previous comparison and the conclusions of our work, we believe that the further development of NESSIE makes sense.

This is why we present, in the next section, the other works that are currently in process.
6.3 Perspectives

The current and future works target three goals: the full automatization of NESSIE for a clever generation and exploration of solutions, the constitution of a wide models library as a support to the multicriteria exploration and the improvement of the mapping engine.

For the former, three works are in process:

1. the study and development of an automatic scenarios generator coupled with a clever multicriteria exploration of solutions including MCDA tools (Multi-Criteria Decision Aid) [1], [2]. The proposed framework is illustrated in the figure 6.2.

2. the interface with standard modeling languages to set the generation of NESSIE’s proprietary XML files aside and be able to automatically generate input files from other representations. A preliminary work [3] has shown the feasibility to automatically transform the MARTE UML profile into the XML inputs of NESSIE thanks to models transformations.

3. the development of a graphical user interface (GUI) that enables an intuitive description of the design problem, the automatic launch of NESSIE and an easy analysis of its outputs.

To deal with the other goals, three works are foreseen:

1. perform a wide simulation campaign to get cost models for typical design problems. In particular, apply this campaign on the exploration of FPGA and MPSoC platforms.

2. extend the study of the IMEC design problem presented in this work. In particular, model the application at a higher level of abstraction, what also required to get cost models to estimate criteria. Three levels are suggested depending on the kind of primitives considered (from the higher to the lower): standard kernels (FFT, cos, ...), main software pipelined loop parts (prologue, kernel, epilogue), code lines.

3. improve the mapping engine of NESSIE. This implies the study and adaptation of the allocation and routing policy, the management of memorization blocks, the resolution of deadlocks. This study includes simulation time considerations.

Moreover, a future version of NESSIE could implement a more suitable MoC regarding the current applications which are designed in the SoC (multimedia, telecommunication).
Figure 6.2: Framework integrating MCDA and automatic generation and exploration of scenarios to the NESSIE engine
Another MoC?

Regarding the case studies we have analyzed and modeled in NESSIE, we have seen that Petri Networks (PN) can be used to represent the functionality at different levels of abstraction. However, we have also seen that it has some weaknesses that could be improved by using other MoC’s or by completing the current definition of the PN MoC used inside NESSIE.

The properties of the PN implemented in NESSIE are:

- Thanks to the transitions and the tokens elements, control is represented in the MoC.
- Data flow concepts have also been integrated in the PN to represent the data flow between different operations and their data dependency.
- Hierarchy has also been implemented to represent the complexity of the application. However, it is not defined automatically.

The weaknesses are:

- Difficulty to represent complex systems, or distributed/repetitive systems. Even if we can model loops in a dense way, it is tedious to describe lot of places and transitions if we consider the application at a middle or low level of abstraction. The PN MoC is not very suitable for data-oriented applications.
- Real-time or reactive modeling is not supported.

The papers [4], [5], [6] give a classification of the different types of MoC’s that can be used today to represent the functionality to design embedded systems. In this work, we don’t have studied the MoC’s possibilities and we will not detail all the models that have been compared in these papers. However, we will advert the MoC’s that we suggest to explore in order to improve the weaknesses identified in the current Petri Network model.

**Complexity and data-oriented applications**

Current complex applications are typically based on intensive signal processing highly data-oriented (multimedia, telecommunication). If the concept of data flow has been integrated in the PN in NESSIE, the model is not very suitable to represent easily this kind of application. On the contrary, other MoC’s are dedicated to model data-oriented applications.

- **Kahn Process Network** : KPN [7] is a set of processes communicating (one reader, one writer) thanks to unidirectional unbounded FIFO (queue of not timed tokens). They allow for latency in the communication channels.
- **Data Flow Graphs** : DFG are typically used to represent this kind of applications and support hierarchy. They can be used as Synchronous (fixed number of tokens) or Asynchronous (variable number of consumed tokens) DF. It uses directed graphs also composed of nodes (actors : inputs, outputs and operations) and edges (datapath between nodes).
• Hierarchical Concurrent Finite State Machines (HCFSM) [8] are composed of several concurrency models and allow a hierarchical representation of the functionality, by using dataflow, discrete-events and synchronous/reactive MoC’s.

Moreover, regarding the complexity, it is important to be able to represent the system more or less accurately. So-called Hierarchical Petri Networks have been created to deal with such issue by integrating natively a hierarchical modeling of the functionality [9].

Real-time and reactive applications Models exist to capture real-time behaviour which must react to external and dynamic stimuli. This is the case of Synchronous Models and Reactive Models. A typical language that supports this MoC is Esterel. They are based on the synchrony hypothesis that outputs are produced instantly in reaction to inputs (with no observable delay in the outputs). However, the paper [10] presents the use of Petri Networks to also design Real-Time embedded systems and can be of interest to complete our model with suitable properties.

6.4 Conclusions

In this dissertation, we have put NESSIE to the test on two external case studies that illustrate typical current design problems in order to validate this original tool in the context of design space exploration and multi-criteria estimation.

The main contributions of our work are:

• A characterization of a substantial number of additional estimation and design tools. This review of the literature comes as a complement to the review presented in the previous work. It has shown that NESSIE is still in the race to cope with existing tools limitations.

• The study and the formalization of the IMEC design flow and the modeling of the ADRES processor. In this case study,

  – we have proposed different models regarding the limitations of the tools.
  – we have validated NESSIE regarding the prediction of multiple criteria.
  – we have shown that, according to future developments to improve the modeling time, NESSIE is competitive to reduce the design time.

• The study and the formalization of the 3D stacking problems and its modeling into NESSIE. In this case study,

  – we have shown that NESSIE is flexible enough to target a different design problem, at a higher level of abstraction.
  – we have validated NESSIE regarding the estimation of a criterion based on the evaluation of an analytical expression and its prediction capability.
– we have shown that in its current version, we take more advantage of NESSIE regarding the modeling and design time reduction for systems considered at a higher level of abstraction.

As a conclusion, if this validation work has highlighted the limitations of NESSIE, it has also shown the interest of a generic approach implementing flexibility, fast simulation and prediction capability. Moreover, along the work, designers have shown real interest in such a prediction tool and express the need to improve the classical design flows. However, to be competitive, several developments and improvements must be done. These have been identified and different research topics are already in process to complete NESSIE with a full automatic framework.

In more general terms, I think an important contribution of this thesis lies in the definition of the future ingredients that will build a complete framework around NESSIE in the perspective of an industrial and efficient use of the tool. This is the result of my validation work and the underlying external collaborations. On a personal level, this thesis has particularly given rise to analytical and formalization skills.

Bibliography


Appendix A

A complement to the IMEC case study

A.1 TCL/TK developments

A.1.1 The timeline

The figure A.1 shows the table representation of the Timeline.xml file. Each column corresponds to a time step. For each time step, all the triggered events are given. They are represented by the HW ID associated to the event, its current state, optionally, the SW ID which is computed or memorized, or the second HW ID associated to a sending or receiving state.

![TCL/TK interface of the Timeline - table representation](image)

**Figure A.1:** TCL/TK interface of the Timeline - table representation

A save button has been included in the interface to save the formatted content of the timeline into a text file that can be further imported into excel sheet.
The filter button opens an other window that presents the events in a pipelined view. A screenshot of this pipelined view is shown in the figure A.2. This simple representation sorts the events from the HW blocks viewpoints to visualize directly the successive states associated to a given HW ID along the time steps. The first column contains the time steps, in a croissant order. And the other columns are dedicated respectively to each HW blocks, in the order of the HW IDs (column 1 is dedicated to the events of block 0, column 2 to block1, etc...).

A save button has also been included to save the table in a .txt format.

**Figure A.2:** TCL/TK interface of the Timeline -pipelined view

### A.1.2 The data token routing

To manage and analyze the token routing on the platform along the mapping process, we have extracted from the NESSIE core all the ID’s of the token, with the HW producer and the associated tasks (place ID) that has generated the token. These informations are stored in a textual representation that can be used as input to an GUI module. A piece of such file is shown in the figure A.3.

In a ”debugging” and mapping ”analyzis” perspective, a simple interface has been built to visualize the entire route followed by a data token on the platform. This allows, e.g., to identify deadlocks. This is illustrated in the figure A.4. The first column gives the IDs of the token, in the order they are generated during the mapping. The second column gives the number of events (memorizing, sending, transmitting,...) associated to the token. The other columns represent each event applied to the data token (the third column, with number ”1”, correspond to the first event, etc...). In these columns, we find the events associated to each token of the corresponding the row, with the time step at which the event has been triggered.

### A.2 NESSIE Pareto graphs

Several graphs have been produced to identify the Pareto points associated to the estimated criteria. Here are given the 2D projections of the two 3D Pareto graph generated for the criteria cycles, area in relation with effIPC and ED respectively. The graphs are
A.3. DRESC Pareto Graphs

In this part, we give the Pareto graphs that have been generated based on the criteria estimation done in DRESC. We find successively the Pareto graphs for two criteria, then the results considering 3 criteria.

A.3.1 Cycles vs Area

In the figure A.8, we show the solutions regarding the number of cycles and the area.

A.3.2 Cycles vs effective IPC

In the figure A.9, we show the solutions regarding the number of cycles and the effective IPC.
A.3.3 Cycles vs ED

In the figure A.10, we show the solutions regarding the number of cycles and the effective density.

A.3.4 Area vs effective IPC

In the figure A.11, we show the solutions regarding the area and the effective IPC.

A.3.5 Area vs ED

In the figure A.12, we show the solutions regarding the area and the effective density.

A.3.6 Cycles and Area vs effIPC/ED

In the next figures, we give the graphs for the Pareto solutions obtained for three criteria, respectively cycles and are in relation with effIPC and ED.

The figure A.13 gives the 3D solutions for the effIPC.

The figure A.14 give the projection regarding the area and the cycles.

The figure A.15 give the projection regarding the IPC and the cycles.

The figure A.16 give the projection regarding the ED and the cycles.
Figure A.5: Nessie results: pareto of area vs cycles (3D)
Figure A.6: Nessie results: pareto of ipc vs cycles (3D)
A.3. DRESC PARETO GRAPHS

Figure A.7: Nessie results: pareto of ipc vs cycles (3D)
Figure A.8: DRESC results: pareto of area vs cycles (2D)
Figure A.9: DRESC results: pareto of effIPC vs cycles (2D)
Figure A.10: DRESC results: pareto of ED vs cycles (2D)
Figure A.11: DRESC results: pareto of area vs effIPC (2D)
Figure A.12: DRESC results: pareto of area vs ED (2D)
Figure A.13: DRESC results: pareto of area, effIPC vs cycles (3D)
Figure A.14: DRESC results: projection of the 3D pareto graph - area vs cycles
Figure A.15: DRESC results: projection of the 3D pareto graph - effIPC vs cycles
Figure A.16: DRESC results: projection of the 3D pareto graph - ED vs cycles
Appendix B

A complement to the 3D stacking

B.1 Data tables

B.1.1 Bus Load values

We give the bus load values from which we have extracted the data sizes associated to the software primitives for the petri network modeling of the application scenarios data split, functional split and hybrid split. These values have been obtained from the external tool chain.

The tables B.1, B.2 and B.3 give these bus loads (in bytes) for respectively the data, the functional and the hybrid splits. In each table, the three resolutions are presented (CIF, 4CIF, HDTV).

B.1.2 Data size values

The data sizes resulting from the bus loads values, and that have been used to feed the SW primitives of our petri nets, are given in the tables B.4, B.5 and B.6 for the three resolutions and for the scenarios data split, functional split and hybrid split respectively.

B.1.3 Wire Lengths

In this section, we give the wire lengths that have been estimated by the Cadence SoC Encounter tool, and the values that we have used in NESSIE to model the different 3D stacked architecture variants. The first column gives the components linked to the wire. The second column gives the wire ID. The other columns give the wire lengths for the Cadence tool and NESSIE respectively, in millimeters. The tables B.7, B.8 and B.9 give the values for the architecture variants v1 to v3, v4 to v6 and v5 to v8 respectively.
### Table B.1: Data split - bus load in bytes

<table>
<thead>
<tr>
<th>Links</th>
<th>CIF</th>
<th>4CIF</th>
<th>HDTV</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO ↔ sw11</td>
<td>6517</td>
<td>27132</td>
<td>62244</td>
</tr>
<tr>
<td>EMIF ↔ sw10</td>
<td>16891</td>
<td>185136</td>
<td>421344</td>
</tr>
<tr>
<td>AD1 ↔ sw00</td>
<td>5852</td>
<td>43092</td>
<td>98154</td>
</tr>
<tr>
<td>AD2 ↔ sw00</td>
<td>5852</td>
<td>43092</td>
<td>98154</td>
</tr>
<tr>
<td>AD3 ↔ sw00</td>
<td>5852</td>
<td>43092</td>
<td>98154</td>
</tr>
<tr>
<td>AD4 ↔ sw01</td>
<td>5852</td>
<td>43092</td>
<td>98154</td>
</tr>
<tr>
<td>AD5 ↔ sw01</td>
<td>5852</td>
<td>43092</td>
<td>98154</td>
</tr>
<tr>
<td>AD6 ↔ sw01</td>
<td>5852</td>
<td>43092</td>
<td>98154</td>
</tr>
<tr>
<td>L2D1 ↔ sw10</td>
<td>30851</td>
<td>123424</td>
<td>280763</td>
</tr>
<tr>
<td>L2D2 ↔ sw10</td>
<td>7714</td>
<td>30956</td>
<td>70091</td>
</tr>
<tr>
<td>sw00 ↔ sw11</td>
<td>399</td>
<td>1995</td>
<td>4788</td>
</tr>
<tr>
<td>sw11 ↔ sw01</td>
<td>399</td>
<td>1995</td>
<td>4788</td>
</tr>
<tr>
<td>sw01 ↔ sw10</td>
<td>17157</td>
<td>127281</td>
<td>289674</td>
</tr>
<tr>
<td>sw10 ↔ sw00</td>
<td>17157</td>
<td>127281</td>
<td>289674</td>
</tr>
<tr>
<td>sw00 ↔ sw01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw11 ↔ sw10</td>
<td>5719</td>
<td>23142</td>
<td>52668</td>
</tr>
<tr>
<td>ARM ↔ sw11</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table B.2: Functional split - bus load in bytes

<table>
<thead>
<tr>
<th>Links</th>
<th>CIF</th>
<th>4CIF</th>
<th>HDTV</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO ↔ sw11</td>
<td>6650</td>
<td>26733</td>
<td>60914</td>
</tr>
<tr>
<td>EMIF ↔ sw10</td>
<td>13433</td>
<td>53998</td>
<td>122892</td>
</tr>
<tr>
<td>AD1 ↔ sw00</td>
<td>57589</td>
<td>231420</td>
<td>526148</td>
</tr>
<tr>
<td>AD2 ↔ sw00</td>
<td>53732</td>
<td>215992</td>
<td>491036</td>
</tr>
<tr>
<td>AD3 ↔ sw00</td>
<td>34580</td>
<td>138852</td>
<td>315609</td>
</tr>
<tr>
<td>AD4 ↔ sw01</td>
<td>17290</td>
<td>69426</td>
<td>157871</td>
</tr>
<tr>
<td>AD5 ↔ sw01</td>
<td>11438</td>
<td>46284</td>
<td>105336</td>
</tr>
<tr>
<td>AD6 ↔ sw01</td>
<td>1862</td>
<td>7182</td>
<td>16492</td>
</tr>
<tr>
<td>L2D1 ↔ sw10</td>
<td>13433</td>
<td>53998</td>
<td>122579</td>
</tr>
<tr>
<td>L2D2 ↔ sw10</td>
<td>34580</td>
<td>138852</td>
<td>315609</td>
</tr>
<tr>
<td>sw00 ↔ sw11</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw11 ↔ sw01</td>
<td>931</td>
<td>3591</td>
<td>8246</td>
</tr>
<tr>
<td>sw01 ↔ sw10</td>
<td>9576</td>
<td>38570</td>
<td>87780</td>
</tr>
<tr>
<td>sw10 ↔ sw00</td>
<td>30723</td>
<td>123424</td>
<td>280630</td>
</tr>
<tr>
<td>sw00 ↔ sw01</td>
<td>7714</td>
<td>30856</td>
<td>70091</td>
</tr>
<tr>
<td>sw11 ↔ sw10</td>
<td>5719</td>
<td>23142</td>
<td>52668</td>
</tr>
<tr>
<td>ARM ↔ sw11</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
B.2 ORDERED BAR GRAPHS

The bar graphs that have been generated to pre-select best estimated solutions generated in NESSIE and in the external 3D stacking design flow are given below.

B.2.1 NESSIE results

The figures B.1, B.2 and B.3 give the ordered results for the 24 solutions, for the power consumption of the wires (in uW) estimated by NESSIE, respectively for the CIF, the 4CIF and the HDTV resolutions.

<table>
<thead>
<tr>
<th>Links</th>
<th>CIF</th>
<th>4CIF</th>
<th>HDTV</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO ↔ sw11</td>
<td>6783</td>
<td>27398</td>
<td>62244</td>
</tr>
<tr>
<td>EMIF ↔ sw10</td>
<td>74214</td>
<td>92568</td>
<td>610672</td>
</tr>
<tr>
<td>AD1 ↔ sw00</td>
<td>11438</td>
<td>46284</td>
<td>105203</td>
</tr>
<tr>
<td>AD2 ↔ sw00</td>
<td>11438</td>
<td>46284</td>
<td>105203</td>
</tr>
<tr>
<td>AD3 ↔ sw00</td>
<td>11438</td>
<td>46284</td>
<td>105203</td>
</tr>
<tr>
<td>AD4 ↔ sw01</td>
<td>23807</td>
<td>95893</td>
<td>219583</td>
</tr>
<tr>
<td>AD5 ↔ sw01</td>
<td>11438</td>
<td>46284</td>
<td>105336</td>
</tr>
<tr>
<td>AD6 ↔ sw01</td>
<td>7847</td>
<td>31388</td>
<td>71155</td>
</tr>
<tr>
<td>L2D1 ↔ sw10</td>
<td>7714</td>
<td>30856</td>
<td>70091</td>
</tr>
<tr>
<td>L2D2 ↔ sw10</td>
<td>30856</td>
<td>123424</td>
<td>280364</td>
</tr>
<tr>
<td>sw00 ↔ sw11</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw11 ↔ sw01</td>
<td>1064</td>
<td>4256</td>
<td>9576</td>
</tr>
<tr>
<td>sw01 ↔ sw10</td>
<td>11438</td>
<td>46284</td>
<td>105336</td>
</tr>
<tr>
<td>sw10 ↔ sw00</td>
<td>28728</td>
<td>115710</td>
<td>262941</td>
</tr>
<tr>
<td>sw00 ↔ sw01</td>
<td>5586</td>
<td>23142</td>
<td>52668</td>
</tr>
<tr>
<td>sw11 ↔ sw10</td>
<td>5719</td>
<td>23142</td>
<td>52668</td>
</tr>
<tr>
<td>ARM ↔ sw11</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.3: Hybrid - bus load in bytes

<table>
<thead>
<tr>
<th>Links</th>
<th>CIF</th>
<th>4CIF</th>
<th>HDTV</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO → EMIF</td>
<td>5719</td>
<td>23142</td>
<td>52668</td>
</tr>
<tr>
<td>EMIF → ADi</td>
<td>1862</td>
<td>26999</td>
<td>61446</td>
</tr>
<tr>
<td>ADi → L2D2</td>
<td>700</td>
<td>2900</td>
<td>6600</td>
</tr>
<tr>
<td>L2D2 → L2D1</td>
<td>3514</td>
<td>13456</td>
<td>30491</td>
</tr>
<tr>
<td>L2D1 → ADi</td>
<td>4557</td>
<td>18328</td>
<td>41712</td>
</tr>
<tr>
<td>ADi → FIFO</td>
<td>133</td>
<td>665</td>
<td>1596</td>
</tr>
</tbody>
</table>

Table B.4: Data split - nodes data exchange

B.2 Ordered bar graphs

The bar graphs that have been generated to pre-select best estimated solutions generated in NESSIE and in the external 3D stacking design flow are given below.
APPENDIX B. A COMPLEMENT TO THE 3D STACKING

<table>
<thead>
<tr>
<th>Links</th>
<th>CIF</th>
<th>4CIF</th>
<th>HDTV</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO → EMIF</td>
<td>5719</td>
<td>23142</td>
<td>52668</td>
</tr>
<tr>
<td>EMIF → ADj</td>
<td>1000</td>
<td>4285</td>
<td>10674</td>
</tr>
<tr>
<td>EMIF → AD4</td>
<td>5000</td>
<td>18000</td>
<td>50000</td>
</tr>
<tr>
<td>AD1 → AD2</td>
<td>24006.5</td>
<td>96158</td>
<td>218652</td>
</tr>
<tr>
<td>AD1 → L2D2</td>
<td>31654</td>
<td>127548</td>
<td>289940</td>
</tr>
<tr>
<td>L2D2 → L2D1</td>
<td>2926</td>
<td>11304</td>
<td>25669</td>
</tr>
<tr>
<td>AD2 → AD3</td>
<td>27797</td>
<td>112120</td>
<td>254828</td>
</tr>
<tr>
<td>AD3 → AD4</td>
<td>8000</td>
<td>25000</td>
<td>65000</td>
</tr>
<tr>
<td>AD4 → AD5</td>
<td>2000</td>
<td>7180</td>
<td>20000</td>
</tr>
<tr>
<td>AD4 → AD6</td>
<td>1000</td>
<td>5000</td>
<td>15000</td>
</tr>
<tr>
<td>AD5 → L2D1</td>
<td>10507</td>
<td>42694</td>
<td>90000</td>
</tr>
<tr>
<td>AD6 → FIFO</td>
<td>900</td>
<td>3000</td>
<td>7000</td>
</tr>
</tbody>
</table>

Table B.5: Functional split - nodes data exchange

<table>
<thead>
<tr>
<th>Links</th>
<th>CIF</th>
<th>4CIF</th>
<th>HDTV</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO → EMIF</td>
<td>5719</td>
<td>23142</td>
<td>52668</td>
</tr>
<tr>
<td>EMIF → ADj</td>
<td>3901.33</td>
<td>15538.8</td>
<td>35112</td>
</tr>
<tr>
<td>EMIF → AD4</td>
<td>5719</td>
<td>22809.5</td>
<td>52668</td>
</tr>
<tr>
<td>AD1,2,3 → AD4</td>
<td>1862</td>
<td>7714</td>
<td>17556</td>
</tr>
<tr>
<td>AD4 → AD5</td>
<td>5719</td>
<td>22809.5</td>
<td>52668</td>
</tr>
<tr>
<td>AD4 → AD6</td>
<td>6783</td>
<td>27132</td>
<td>61579</td>
</tr>
<tr>
<td>AD6 → FIFO</td>
<td>1064</td>
<td>4256</td>
<td>9576</td>
</tr>
<tr>
<td>AD5 → L2D1</td>
<td>5719</td>
<td>23474.5</td>
<td>52668</td>
</tr>
<tr>
<td>L2D1 → L2D2</td>
<td>1995</td>
<td>7381.5</td>
<td>17423</td>
</tr>
<tr>
<td>L2D2 → ADj</td>
<td>9620.3</td>
<td>38680.8</td>
<td>87647</td>
</tr>
</tbody>
</table>

Table B.6: Hybrid - nodes data exchange

B.2.2 NTUA results

The figures B.4, B.5 and B.6 give the ordered results for the 24 solutions, for the power consumption of the wires (in uW) estimated by the external flow (used at NTUA), respectively for the CIF, the 4CIF and the HDTV resolutions.
### B.2. ORDERED BAR GRAPHS

Table B.7: Length of the NoC wires estimated by Cadence SoC Encounter and used in NESSIE - architecture variants V1 to V3

<table>
<thead>
<tr>
<th>Blocks</th>
<th>ID</th>
<th>V1 Cadence</th>
<th>V1 NESSIE</th>
<th>V2 Cadence</th>
<th>V2 NESSIE</th>
<th>V3 Cadence</th>
<th>V3 NESSIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>wire1</td>
<td>0.915</td>
<td>1</td>
<td>2.303</td>
<td>2.3</td>
<td>0.917</td>
<td>1</td>
</tr>
<tr>
<td>A2</td>
<td>wire2</td>
<td>2.496</td>
<td>2.5</td>
<td>0.967</td>
<td>1</td>
<td>0.848</td>
<td>1</td>
</tr>
<tr>
<td>A3</td>
<td>wire3</td>
<td>0.733</td>
<td>0.5</td>
<td>1.084</td>
<td>1</td>
<td>2.261</td>
<td>2.3</td>
</tr>
<tr>
<td>A4</td>
<td>wire4</td>
<td>1.977</td>
<td>2</td>
<td>2.953</td>
<td>3</td>
<td>0.843</td>
<td>1</td>
</tr>
<tr>
<td>A5</td>
<td>wire5</td>
<td>1.083</td>
<td>1</td>
<td>1.473</td>
<td>1.5</td>
<td>0.757</td>
<td>0.7</td>
</tr>
<tr>
<td>A6</td>
<td>wire6</td>
<td>2.764</td>
<td>2.5</td>
<td>1.29</td>
<td>1.3</td>
<td>1.836</td>
<td>1.8</td>
</tr>
<tr>
<td>L2D1</td>
<td>wire7</td>
<td>1.552</td>
<td>1.5</td>
<td>2.352</td>
<td>2.5</td>
<td>1.79</td>
<td>1.8</td>
</tr>
<tr>
<td>L2D2</td>
<td>wire8</td>
<td>2.065</td>
<td>2</td>
<td>2.3</td>
<td>2.5</td>
<td>1.865</td>
<td>1.8</td>
</tr>
<tr>
<td>FIFO</td>
<td>wire9</td>
<td>0.375</td>
<td>0.4</td>
<td>0.331</td>
<td>0.3</td>
<td>0.899</td>
<td>1</td>
</tr>
<tr>
<td>EMIF</td>
<td>wire10</td>
<td>1.681</td>
<td>1.5</td>
<td>2.017</td>
<td>2</td>
<td>2.983</td>
<td>3</td>
</tr>
<tr>
<td>sw00-sw10</td>
<td>wire11</td>
<td>2.342</td>
<td>2.5</td>
<td>0.598</td>
<td>0.5</td>
<td>0.886</td>
<td>1</td>
</tr>
<tr>
<td>sw00-sw01</td>
<td>wire12</td>
<td>0.445</td>
<td>0.5</td>
<td>0.4</td>
<td>0.5</td>
<td>0.209</td>
<td>0.2</td>
</tr>
<tr>
<td>sw00-sw11</td>
<td>wire13</td>
<td>1.403</td>
<td>1.4</td>
<td>2.735</td>
<td>2.7</td>
<td>0.528</td>
<td>0.5</td>
</tr>
<tr>
<td>sw11-sw10</td>
<td>wire14</td>
<td>1.209</td>
<td>1</td>
<td>2.136</td>
<td>2</td>
<td>0.836</td>
<td>1</td>
</tr>
<tr>
<td>sw01-sw10</td>
<td>wire15</td>
<td>2.654</td>
<td>2.5</td>
<td>0.3</td>
<td>0.3</td>
<td>1.084</td>
<td>1</td>
</tr>
<tr>
<td>sw11-sw01</td>
<td>wire16</td>
<td>1.458</td>
<td>1.5</td>
<td>2.38</td>
<td>2.5</td>
<td>0.75</td>
<td>0.7</td>
</tr>
<tr>
<td>L2I</td>
<td>wire17</td>
<td>3.060</td>
<td>3</td>
<td>3.264</td>
<td>3.3</td>
<td>1.375</td>
<td>1.4</td>
</tr>
<tr>
<td>L2I2</td>
<td>wire18</td>
<td>3.060</td>
<td>3</td>
<td>3.264</td>
<td>3.3</td>
<td>1.375</td>
<td>1.4</td>
</tr>
<tr>
<td>A1-swI</td>
<td>wire19</td>
<td>3.6</td>
<td>3.5</td>
<td>3.816</td>
<td>4</td>
<td>3.2</td>
<td>3</td>
</tr>
<tr>
<td>A2-swI</td>
<td>wire20</td>
<td>3.6</td>
<td>3.5</td>
<td>3.816</td>
<td>4</td>
<td>3.2</td>
<td>3</td>
</tr>
<tr>
<td>A3-swI</td>
<td>wire21</td>
<td>3.6</td>
<td>3.5</td>
<td>3.816</td>
<td>4</td>
<td>3.2</td>
<td>3</td>
</tr>
<tr>
<td>A4-swI</td>
<td>wire22</td>
<td>4.951</td>
<td>5</td>
<td>4.048</td>
<td>4</td>
<td>2.975</td>
<td>3</td>
</tr>
<tr>
<td>A5-swI</td>
<td>wire23</td>
<td>4.951</td>
<td>5</td>
<td>4.048</td>
<td>4</td>
<td>2.975</td>
<td>3</td>
</tr>
<tr>
<td>A6-swI</td>
<td>wire24</td>
<td>4.951</td>
<td>5</td>
<td>4.048</td>
<td>4</td>
<td>2.975</td>
<td>3</td>
</tr>
</tbody>
</table>
### Table B.8: Length of the NoC wires estimated by Cadence SoC Encounter and used in NESSIE - architecture variants V4 to V6

<table>
<thead>
<tr>
<th>Blocks</th>
<th>ID</th>
<th>V4 Cadence</th>
<th>V4 NESSIE</th>
<th>V5 Cadence</th>
<th>V5 NESSIE</th>
<th>V6 Cadence</th>
<th>V6 NESSIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>wire1</td>
<td>0.433</td>
<td>0.4</td>
<td>0.001</td>
<td>0.564</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>wire2</td>
<td>0.372</td>
<td>0.4</td>
<td>0.001</td>
<td>0.5</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>wire3</td>
<td>0.449</td>
<td>0.4</td>
<td>0.001</td>
<td>1.914</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>wire4</td>
<td>0.629</td>
<td>0.5</td>
<td>0.984</td>
<td>1.398</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>wire5</td>
<td>1.317</td>
<td>1.3</td>
<td>1.0425</td>
<td>0.494</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>wire6</td>
<td>1.13</td>
<td>1</td>
<td>0.858</td>
<td>1.938</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>L2D1</td>
<td>wire7</td>
<td>0.681</td>
<td>0.5</td>
<td>0.72</td>
<td>1.311</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>L2D2</td>
<td>wire8</td>
<td>0.725</td>
<td>0.7</td>
<td>0.788</td>
<td>1.318</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>FIFO</td>
<td>wire9</td>
<td>0.747</td>
<td>0.7</td>
<td>1.024</td>
<td>1.2185</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>EMIF</td>
<td>wire10</td>
<td>0.429</td>
<td>0.5</td>
<td>1.782</td>
<td>1.731</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td>sw00-sw10</td>
<td>wire11</td>
<td>1.485</td>
<td>1.5</td>
<td>0.643</td>
<td>1.973</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>sw00-sw01</td>
<td>wire12</td>
<td>1.163</td>
<td>1</td>
<td>0.281</td>
<td>0.89</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sw00-sw11</td>
<td>wire13</td>
<td>1.493</td>
<td>1.5</td>
<td>0.454</td>
<td>0.331</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>sw11-sw10</td>
<td>wire14</td>
<td>0.843</td>
<td>1</td>
<td>1.04</td>
<td>1.609</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>sw01-sw10</td>
<td>wire15</td>
<td>0.681</td>
<td>0.6</td>
<td>0.82</td>
<td>1.109</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sw11-sw01</td>
<td>wire16</td>
<td>0.671</td>
<td>0.6</td>
<td>0.3</td>
<td>0.553</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>L2I</td>
<td>wire17</td>
<td>0.956</td>
<td>1</td>
<td>0.947</td>
<td>1.767</td>
<td>1.8</td>
<td></td>
</tr>
<tr>
<td>L2I</td>
<td>wire18</td>
<td>0.956</td>
<td>1</td>
<td>0.947</td>
<td>1.767</td>
<td>1.8</td>
<td></td>
</tr>
<tr>
<td>A1-sw1</td>
<td>wire19</td>
<td>2.535</td>
<td>2.5</td>
<td>0.37</td>
<td>3.575</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td>A2-sw1</td>
<td>wire20</td>
<td>2.535</td>
<td>2.5</td>
<td>0.37</td>
<td>3.575</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td>A3-sw1</td>
<td>wire21</td>
<td>2.535</td>
<td>2.5</td>
<td>0.37</td>
<td>3.575</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td>A4-sw1</td>
<td>wire22</td>
<td>1.416</td>
<td>1.5</td>
<td>2.0798</td>
<td>2.21</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>A5-sw1</td>
<td>wire23</td>
<td>1.416</td>
<td>1.5</td>
<td>2.0798</td>
<td>2.21</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>A6-sw1</td>
<td>wire24</td>
<td>1.416</td>
<td>1.5</td>
<td>2.0798</td>
<td>2.21</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
### Table B.9: Length of the NoC wires estimated by Cadence SoC Encounter and used in NESSIE - architecture variants V7 and V8

<table>
<thead>
<tr>
<th>Blocks</th>
<th>ID</th>
<th>V7 Cadence</th>
<th>NESSIE</th>
<th>V8 Cadence</th>
<th>NESSIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>wire1</td>
<td>0.64</td>
<td>0.6</td>
<td>0.858</td>
<td>0.8</td>
</tr>
<tr>
<td>A2</td>
<td>wire2</td>
<td>0.338</td>
<td>0.3</td>
<td>1.529</td>
<td>1.5</td>
</tr>
<tr>
<td>A3</td>
<td>wire3</td>
<td>0.251</td>
<td>0.3</td>
<td>2.198</td>
<td>2</td>
</tr>
<tr>
<td>A4</td>
<td>wire4</td>
<td>0.497</td>
<td>0.5</td>
<td>1.735</td>
<td>1.7</td>
</tr>
<tr>
<td>A5</td>
<td>wire5</td>
<td>0.184</td>
<td>0.2</td>
<td>1.168</td>
<td>1.2</td>
</tr>
<tr>
<td>A6</td>
<td>wire6</td>
<td>0.307</td>
<td>0.3</td>
<td>2.348</td>
<td>2.3</td>
</tr>
<tr>
<td>L2D1</td>
<td>wire7</td>
<td>0.623</td>
<td>0.5</td>
<td>2.395</td>
<td>2.3</td>
</tr>
<tr>
<td>L2D2</td>
<td>wire8</td>
<td>0.262</td>
<td>0.3</td>
<td>2.185</td>
<td>2.2</td>
</tr>
<tr>
<td>FIFO</td>
<td>wire9</td>
<td>0.403</td>
<td>0.4</td>
<td>0.83</td>
<td>0.8</td>
</tr>
<tr>
<td>EMIF</td>
<td>wire10</td>
<td>1.579</td>
<td>1.5</td>
<td>0.452</td>
<td>0.5</td>
</tr>
<tr>
<td>sw00-sw10</td>
<td>wire11</td>
<td>0.919</td>
<td>1</td>
<td>0.496</td>
<td>0.5</td>
</tr>
<tr>
<td>sw00-sw01</td>
<td>wire12</td>
<td>1.658</td>
<td>1.6</td>
<td>0.584</td>
<td>0.5</td>
</tr>
<tr>
<td>sw00-sw11</td>
<td>wire13</td>
<td>0.847</td>
<td>0.8</td>
<td>0.695</td>
<td>0.7</td>
</tr>
<tr>
<td>sw11-sw10</td>
<td>wire14</td>
<td>0.773</td>
<td>0.8</td>
<td>0.55</td>
<td>0.5</td>
</tr>
<tr>
<td>sw01-sw10</td>
<td>wire15</td>
<td>1.646</td>
<td>1.5</td>
<td>0.36</td>
<td>0.4</td>
</tr>
<tr>
<td>sw11-sw01</td>
<td>wire16</td>
<td>1.284</td>
<td>1.3</td>
<td>0.214</td>
<td>0.2</td>
</tr>
<tr>
<td>L2D1</td>
<td>wire17</td>
<td>1.339</td>
<td>1.3</td>
<td>1.522</td>
<td>1.5</td>
</tr>
<tr>
<td>L2D2</td>
<td>wire18</td>
<td>1.339</td>
<td>1.3</td>
<td>1.522</td>
<td>1.5</td>
</tr>
<tr>
<td>A1-swI</td>
<td>wire19</td>
<td>0.846</td>
<td>0.8</td>
<td>4.024</td>
<td>4</td>
</tr>
<tr>
<td>A2-swI</td>
<td>wire20</td>
<td>0.846</td>
<td>0.8</td>
<td>4.024</td>
<td>4</td>
</tr>
<tr>
<td>A3-swI</td>
<td>wire21</td>
<td>0.846</td>
<td>0.8</td>
<td>4.024</td>
<td>4</td>
</tr>
<tr>
<td>A4-swI</td>
<td>wire22</td>
<td>1.719</td>
<td>1.7</td>
<td>4.234</td>
<td>4</td>
</tr>
<tr>
<td>A5-swI</td>
<td>wire23</td>
<td>1.719</td>
<td>1.7</td>
<td>4.234</td>
<td>4</td>
</tr>
<tr>
<td>A6-swI</td>
<td>wire24</td>
<td>1.719</td>
<td>1.7</td>
<td>4.234</td>
<td>4</td>
</tr>
</tbody>
</table>
Figure B.1: Ordered NESSIE results of the wires power consumption (in uW) for CIF data resolution

Figure B.2: Ordered NESSIE results of the wires power consumption (in uW) for 4CIF data resolution
B.2. ORDERED BAR GRAPHS

**Figure B.3:** Ordered NESSIE results of the wires power consumption (in uW) for HDTV data resolution

**Figure B.4:** Ordered NTUA results of the wires power consumption (in uW) for CIF data resolution
Figure B.5: Ordered NTUA results of the wires power consumption (in uW) for 4CIF data resolution

Figure B.6: Ordered NTUA results of the wires power consumption (in uW) for HDTV data resolution
Appendix C

A complement to the Future Work

As perspective for the use of our tool in industrial design, we have thus begun to start works on three major axis that are the development of a multiobjective exploration module, a graphical interface to accelerate the definition of the inputs in Nessie and the study of the potential interface of Nessie with standard languages used in industrial tools.

C.1 Multiobjective exploration module

Through the case studies analyzed in the chapters 4 and 5, we have pointed up the limitations of our tool to explore a wide number of solutions as they have to be defined by the user himself: it may take unwanted time, the design space exploration is limited. In the future, we would like to integrate in Nessie a more clever multicriteria exploration of solutions. Moreover, a module could be developed to generate automatically new scenarios based on results obtained from a basic set of solutions.

Several aspects have to be considered:

Fast exploration, performance estimation and optimization amongst all the solutions composing the huge design space, we need algorithms that identify the set of interesting solutions. If exact methods exist, we will preferably choose approximate methods that enable a quick exploration of the solution space and approach the pareto frontier. Metaheuristics are typically used for such multi-objective optimization problem.

Multi-criteria decision aid the MCDA is a discipline that aims at supporting a decision maker facing conflicting solutions, which is typically the case when designing embedded SoC. The MCDA helps to take decision by highlighting compromises to select the most-suitable solutions. To decide, the tool make use of objectives (maximization of the performances, minimization of the costs,...) and preferences models (aggregation, interactive or outranking methods) which are typically interested in the answer of the following problematics: choice of a small number of good solutions, sorting to assign each solution to a predefined category, ranking to pre-order completely or partially all the solutions.
APPENDIX C. A COMPLEMENT TO THE FUTURE WORK

Figure C.1: Framework integrating MCDA and automatic generation and exploration of scenarios to the NESSIE engine

We have seen in the state-of-the-art (see chapter 2) that few tools include an overall design space exploration. We have cited MILAN framework (2002), the SoC architecture Explorer tool (2005) and more recently the MULTICUBE project (2008). A deeper analyse of these tools can be performed to identify how NESSIE could be coupled with a third party framework or take most relevant ingredients that should be integrated directly in our tool. We have already observed, however, that these tools don’t allow the designer to make choices in a transparent process in which preferences could be clearly specified.

A first study has been initiated in particular for the design of 3D Integrated Circuits that offer a huge design space with a wide set of degrees of freedom and multiple objectives to optimize as explained in the chapter 5. The study proposes a model and design method based on the PROMETHEE-GAIA method, developed at the ULB, which is an outranking method used for the problematic of choice and ranking of solutions.

A typical integrated framework could be represented by the figure C.1 in which we have distinguish two parts: the design space exploration and the multi-criteria decision aid parts. In the former, an external module generates several scenarios (applications and/or architectures) based on defined rules, parameters, constraints. This set of scenarios is given as inputs to Nessie which performs the fast simulations and gives the estimated
C.2. GRAPHICAL USER INTERFACE

These results can be fed back to the scenarios generator that can adapt the solutions automatically and give new and better inputs to Nessie. Once a set of satisfying solutions have been generated and estimated to constitute a pareto curve, a decision module is used. In this latter part, methods select the most suitable solution amongst all the solutions constituting the pareto frontier and give this information to the designer for physical design.

C.2 Graphical user interface

In the chapter 4 and 5, we have stressed the non intuitive process and the time spent when filling the input files of NESSIE. Indeed, even if there is autocompletion to build the .XML files and error detection, one needs to create petri networks and hardware structures by using numbers ID or types which are not explicit regarding tasks and components they represent. Moreover, if small structures could be directly defined into the file, for bigger ones, the user can not reasonably have a right global view of the petri networks or of the architectures he has to define.

To be able to manipulate the primitives and the multiple structures, there is a need for a graphical definition of the inputs of the tool.

Remarks have also be done for the interpretation of the output files. Indeed, the more complex and plodding to comprehend is the timeline.xml file which contains all the events composing the mapping with all the timestep. The file rapidly explodes as we have shown in the chapter 4 in particular.

From 2008, we have started to develop a graphical user interface to handle these problems. Several functionalities should be implemented in this GUI.

C.2.1 Generation of inputs

- edit and build HW structures : manually create blocks, allow copy/paste, enable automatic wires routing, automatically generate predefined patterns (mesh, rings, star topologies, ...),....
- edit and build SW structures : manually create blocks, allow copy/paste, connect places, generate automatically nets based on metrics defining the parallelism, the operations set,....
- describe HW/SW primitives : enable the modification of the defined instances
- edit, display and compose Yeti models : define parameters, operation set, orientation

C.2.2 Processing of outputs

- display of the activity report :
- execute jointly the mapping evolution on the petri network from one side and on the material structure on the other side : allow step by step running, event filtering, adapted colors, token routing,....
C.2.3 Other functionalities

- enable the validation of the specified files: errors feedback and parsing
- use of scripts for the different modules launch
- enable projects loading, saving, modifying.
**C.2. GRAPHICAL USER INTERFACE**

![Graphical User Interface](image)

**Figure C.3:** Post-processing graphical interface - Activity Report
Appendix D

Miscellaneous

As it is the case for all young tools, NESSIE suffers of dysfunctions, problems or weaknesses that can be identified only when practicing the tool. This has been the case during this work where lot of modeling have been performed and for which different functionalities were tested. We have thus dedicated a special Appendix to list different points that have to be taken into account for future purpose and development of the tool.

1 Allocation and routing

The allocation and the routing can be adapted by the user through the AllocationWeight.xml and RoutingWeight.xml files. We wondered if it is possible to force the allocation on memory block thanks to an adequate law in the allocationWeight.xml file. We have thought about it and the conclusion is that it is not possible in the current state of the tool to adapt the allocation policy easily to force such mapping. The way to particularize the mapping in Nessie is to express a function cost to evaluate the weight for the HW nodes. This function is the same for all the nodes and all the abstraction levels. This weight can be a function of the input parameters and the output criteria (cost, energy, area,...). We have thought on a possibility to give an expression where we could give a value to a global parameter dynamically and use it into an other local parameter. But it is not possible to change dynamically the parameters. In addition, a limitation could appear when considering more than 2 AL. Indeed, the allocation and routing weight are defined commonly for all abstraction levels what could be a drawback for some design problems.

2 Functional verification and coherency

Contrarily to other existing tools, the user can not check the correctness of the functionality when defining its structures. We have also observed that the consistency between two abstraction levels is not checked before the simulation. Only simulation errors tell the user that incoherencies have been defined in the simulation file what is not always obvious to detect.

1Some have already be highlighted in other chapters. However, we summarize them in this section for a global overview.
States  We notice that into Nessie, the core states and the port states have been numbered. The correspondence between the states and their ID is summarized in the table D.1.

<table>
<thead>
<tr>
<th>Core states</th>
<th>Port states</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: idle</td>
<td>0: inactive</td>
</tr>
<tr>
<td>1: sleeping</td>
<td>1: receiving</td>
</tr>
<tr>
<td>2: transmitting</td>
<td>2: sending</td>
</tr>
<tr>
<td>3: memorizing</td>
<td></td>
</tr>
<tr>
<td>4 – N: computing</td>
<td></td>
</tr>
</tbody>
</table>

Table D.1: Correspondence between states names and IDs in NESSIE

Transition time table  Transition states have been foreseen. Indeed, to express latency between two states, typically the sleeping mode and the idle mode, the user would be able to specify a time. This entry has been added in the simulation file. However, the functionality itself is not yet managed by Nessie.