A System-Level Framework for Energy and Performance Estimation of System-on-Chip Architectures

SANDRO PENOLAZZI

Doctoral Thesis in Electronic and Computer Systems
Stockholm, Sweden 2011
Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen onsdagen den 6 april 2011 klockan 13.00 i Sal C1, Electrum, Isafjordsgatan 26, Kista.

© Sandro Penolazzi, April 2011

Tryck: Universitetsservice US AB
Abstract

Shifting the design entry point up to the system level is the most important countermeasure adopted to manage the increasing complexity of SoCs. The reason is that decisions taken at this level, early in the design cycle, have the greatest impact on the final design in terms of performance, energy efficiency and silicon area occupation. However, taking decisions at this level is very difficult, since the design space is extremely wide, and it has so far been mostly a manual activity. Efficient system-level estimation tools are therefore necessary to enable proper design-space exploration and the development of system-level synthesis tools.

Proposing an efficient approach to system-level estimation is the main contribution of this thesis.

The approach consists of three layers. The bottom layer relies on building a library of IP energy and performance models, where each IP functionality is pre-characterized. Characterization is done only once at the gate level, which gives high accuracy to the approach. The implementation of an energy and performance model for a Leon3 processor is reported as an example. The impact that the IP-to-IP communication infrastructure has over individual IP properties is also taken into account, for bus-based and NoC-based architectures.

The intermediate layer is where the actual estimation takes place. At this level, applications are run and profiled on a development host (a common PC). This allows us to create a trace of the executed source code, which is then mapped to the assembly code of the target architecture. This operation allows a trace of target instructions to be indirectly built and confers high speed on the whole methodology. Once the target trace is inferred, energy and performance figures can be extracted by using the IP models from the bottom layer. To make the whole process possible, changes are made to the GNU GCC compiler. Estimation is shown for a few common image/video codec applications.

The top layer is a refinement layer that accounts for the presence of caches and for the fact that multiple applications normally run concurrently, share the same resources and are controlled by an operating system. Statistical models are built to account for the impact of each of these components. An MPSoC hosting up to 15 processors and using both fixed-priority and round robin bus arbitration is used for modeling bus contention. The RTEMS operating system is taken as a reference to model the OS impact.

Validation for each layer is also carried out. The results show that the approach is within 15% of gate-level accuracy and exhibits an average speed-up of ∼30X compared to transaction-level modeling (TLM).
Acknowledgments

Stockholm, February 2011

I would like first of all to thank my supervisor, Professor Ahmed Hemani. He has always been very present and supported me with his valuable advice and knowledge, especially at the beginning of my PhD, when the path to take still looked uncertain. A great thank you also goes to Ingo Sander, who accepted to be my co-supervisor during the second half of my PhD and who has always given me very useful guidance.

I am very thankful to all my colleagues and friends that shared part of the last five years with me at the department. A warm thanks to Haider Abbas, Davide Calusini, Dr. Zhonghai Lu, Omer Malik, Ali Shami, Adeel Tajammul and Chen Zhipeng for the nice conversations we had, always a valid excuse to take a break. I am particularly grateful to Mohammad Badawi, Luca Bolognino and Bin Liu: each of them has helped me in a different phase of my research and it is also in part thanks to them if the accomplishment of this PhD thesis has been possible. I also want to thank Jean-Michel Chabloz and Shohreh Sharif Mansouri, two dear friends with whom I have shared my thoughts, worries and moments of joy. Thank you also to Jun Zhu for helping me with the practicalities related to the process of handing in the thesis.

I am very grateful to my parents whom, despite the distance, I have always felt near during these long five years of doctoral studies. It is thanks to their support and encouragement if I did not lose my motivation and if I was able to get over the many challenges that appeared on my way. Finally, I want to express my sincere gratitude to a person that has a special place in my life: thank you, Evanny, for always staying close to me and understanding me. Your presence and your words have given me the energy and the optimism I needed to complete this long work.

Sandro Penolazzi
Contents

List of Tables xi
List of Figures xiii

1 Introduction 1

1.1 Background: the Increasing Complexity of Electronic Devices . . . . . . . . 1
  1.1.1 Progress in the semiconductor industry ............................. 2
  1.1.2 Increased applications demand .................................... 2
1.2 System Partitioning ...................................................... 3
1.3 Abstraction: from Physical to System Level ............................ 4
1.4 System-Level Estimation ................................................ 5
1.5 Thesis Contribution ...................................................... 8
1.6 Thesis Layout ............................................................. 11

2 Common Approaches to System-Level Estimation and Design Today 13

2.1 Introduction .............................................................. 13
2.2 Simulation-Based Estimation Tools ...................................... 14
  2.2.1 An overview of SystemC and Transaction Level Modeling (TLM) .... 21
  2.2.2 Time representation in TLM ........................................ 23
  2.2.3 TLM advantages .................................................... 24
  2.2.4 TLM drawbacks .................................................... 25
  2.2.5 Examples of SystemC/TLM usage in RTOS and MPSoCs modeling .................................................... 26
2.3 Analytical Estimation Tools .............................................. 27
2.4 Estimation Tools Based on the Combination of Multiple Approaches .... 27
2.5 Which Category Does Funtime Belong to? ............................ 28

3 An Overview of the Funtime Estimation Framework 31

3.1 Introduction .............................................................. 31
| 3.2 | Funtime User Interface | 32 |
| 3.3 | Level A – The IP Level | 33 |
| 3.3.1 | Level A1 – Individual IPs | 33 |
| 3.3.2 | Level A2 – Architecture of IPs | 34 |
| 3.4 | Level B – The Algorithmic Level | 35 |
| 3.4.1 | Instrumentation and trace generation for applications mapped to software | 37 |
| 3.4.1.1 | Advantages and challenges of the estimation approach for applications mapped to software | 39 |
| 3.4.1.2 | Enhancing the GNU GCC compiler | 40 |
| 3.4.2 | Estimation of applications mapped to hardware | 41 |
| 3.5 | Level C – The Refinement Level | 44 |
| 3.6 | From Single Transactions to a Final Trace | 47 |
| 3.7 | Time Representation in Funtime | 47 |
| 3.8 | Engineering Effort in Funtime | 50 |
| 3.9 | Refinements Order and Interaction | 50 |
| 4.1 | Introduction | 53 |
| 4.2 | Related Work | 54 |
| 4.3 | Configuration and Synthesis | 55 |
| 4.4 | Energy Characterization | 55 |
| 4.4.1 | 1-instruction-based model | 56 |
| 4.4.2 | 2-instruction-based model | 58 |
| 4.5 | Data Switching Activity | 59 |
| 4.6 | Registers Correlation | 60 |
| 4.7 | Validating the Leon3 Model Estimation Accuracy | 61 |
| 4.8 | Summary | 65 |
| 5 | Level A2 – Modeling IPs Composite Transactions | 67 |
| 5.1 | Introduction | 67 |
| 5.2 | Bus-based and NoC-based SoC: Architectural Similarities and Differences | 68 |
| 5.3 | Characterizing Composite Transactions | 69 |
| 5.4 | Validating the Accuracy of the Extended Model for Leon3 | 74 |
| 5.5 | Summary | 76 |
| 6 | Level B – Inferring Primary Transactions and Refinements for Transactions Interdependency and Caches | 77 |
| 6.1 | Introduction | 77 |
| 6.2 | Transactions Interdependency Refinement | 78 |
| 6.3 | Validating the Algorithmic Level and the Accuracy of the Transactions Interdependency Refinement | 80 |
6.4 Cache Effect Refinement ........................................ 82
6.5 Comparing Funtime and Untimed TLM Estimation Speed .... 83

7 Level C – Refining Bus Contention Effects on Energy and Performance .................................................. 85
  7.1 Introduction .......................................................... 85
  7.2 Characterizing Bus Contention .................................... 87
    7.2.1 Factor 1: number of processors ................................ 89
    7.2.2 Factor 2: number of w/r accesses to memory ............ 92
    7.2.3 Summary ....................................................... 93
  7.3 Predicting Bus Contention ......................................... 94
    7.3.1 Case study ..................................................... 94
  7.4 Validating the Accuracy of Bus Contention Prediction ....... 96
  7.5 Summary ............................................................ 97

8 Level C – Refining the Operating System Overhead on Energy and Performance ........................................ 99
  8.1 Introduction .......................................................... 99
  8.2 RTOS Characterization ............................................. 101
    8.2.1 Characterizing a group of RTOS routines ................. 101
  8.3 RTOS Activity Prediction .......................................... 103
    8.3.1 Round Robin scheduler activity prediction ............... 105
      8.3.1.1 Case study 1 ............................................... 105
      8.3.1.2 Case study 2 ............................................... 105
      8.3.1.3 Case study 3 ............................................... 107
    8.3.2 Prediction of Priority-Driven Scheduler Activity ....... 107
      8.3.2.1 Case study 1 ............................................... 108
      8.3.2.2 Case study 2 ............................................... 110
  8.4 Validation of Refinement Accuracy ............................... 113
    8.4.1 Round Robin scheduler ....................................... 113
    8.4.2 Priority-driven scheduler .................................... 114
  8.5 Summary ............................................................ 116

9 Conclusions and Future Work ...................................... 117
  9.1 Conclusions ........................................................ 117
  9.2 Future Work ....................................................... 119

Bibliography .......................................................... 121
List of Tables

4.1 Leon3 area .................................................. 55
4.2 Leon3 LUT for energy and performance ......................... 57
4.3 Energy models: 1-instr. vs. 2-instr. .......................... 58
4.4 Analysis of data switching activity .......................... 59
4.5 Registers correlation analysis ................................ 61
4.6 Leon3 energy models (1-instr-based with/without data dependency) vs. gate-level energy measurement (no cache) ......................... 61
4.7 Leon3 energy models (1-instr-based with/without data dependency) vs. gate-level energy measurement (4kB I/D-cache) ......................... 62
4.8 Leon3 energy models (1-instr-based/2-instr-based with data dependency) vs. gate-level energy measurement (no cache) ......................... 63
4.9 Leon3 energy models (1-instr-based/2-instr-based with data dependency) vs. gate-level energy measurement (4kB I/D-cache) ......................... 64
5.1 Summary of the characterization results for simple and composite transactions in Leon3 (no cache) .................................................. 72
5.2 Leon3 model validation versus gate-level energy measurement .......................... 75
6.1 Comparison between ideal and real average cycles per instruction ......................... 81
6.2 2 sources of inaccuracy: Leon3 energy & performance model + estimated number of cycles (stalls) .......................... 81
6.3 3 sources of inaccuracy: Leon3 energy & performance model + estimated number of cycles (stalls) + estimated number of instructions .......................... 82
6.4 Timing comparison between Funtime and untimed TLM .......................... 84
7.1 Percentage of cycles difference $C_{N,1}^\%$ between N and 1-processor case ......................... 89
7.2 Percentage of cycles difference $C_{N,N-1}^\%$ .......................... 90
7.3 Value of $E_{stall\;cycle} [pJ]$ calculated by Equation 7.2 for $2 \leq N \leq 7$ ......................... 92
7.4 Percentage of instruction-level cycles difference $C_{7,1}^\%$ for the FFT .......................... 92
7.5 Average $\hat{C}_{wr,n}^\%$ and $\hat{C}_{wr,n+1}^\%$ calculated on the FFT, Fibonacci and Viterbi benchmarks .......................... 95
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.6</td>
<td>Measured vs. predicted execution time $ex_i[ms]$ and $E_i[\mu J]$, 3 different applications on a 3-processor SoC.</td>
<td>97</td>
</tr>
<tr>
<td>8.1</td>
<td>Energy and performance characterization for RTEMS on Leon3 (no cache)</td>
<td>103</td>
</tr>
<tr>
<td>8.2</td>
<td>Validating Funtime vs. gate level for energy estimation: 2 RR-scheduled tasks of different length</td>
<td>114</td>
</tr>
<tr>
<td>8.3</td>
<td>Validating Funtime vs. gate level for energy estimation: priority-driven scheduling</td>
<td>115</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>CPU transistors count from 1971 to 2008</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Complexity increase for newer Windows versions [44]</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Gajski-Kuhn Y-chart</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>System-level design challenge: the mapping phase</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>Design space width versus abstraction level</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>Ptolemy hierarchical model using two different domains. Source: [19]</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>SpecC design methodology. Source: [18]</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Mapping network in Metropolis. The functional network on the left side and the architectural network on the right side. Source: [8].</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>Mapping between the application and the architecture model in SPADE. Source: [41].</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>Left: mapping between the application and the architecture model in Artemis. Right: integration between DSE activity and use of reconfigurable architectures. Source: [64].</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>MILAN design, DSE and estimation framework. Source: [49].</td>
<td>19</td>
</tr>
<tr>
<td>2.7</td>
<td>TAPES trace example. Source: [76].</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>C++, SystemC and TLM relation.</td>
<td>22</td>
</tr>
<tr>
<td>2.9</td>
<td>Time abstraction levels. Source: [9].</td>
<td>23</td>
</tr>
<tr>
<td>2.10</td>
<td>TLM-based design flow. Figure taken from [26].</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>Funtime 3-layer estimation framework</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Example: database of IP models A</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Example: architecture composition of IPs</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Instrumentation of applications mapped to software</td>
<td>38</td>
</tr>
<tr>
<td>3.5</td>
<td>The GNU GCC execution flow</td>
<td>41</td>
</tr>
<tr>
<td>3.6</td>
<td>Example of source execution trace</td>
<td>41</td>
</tr>
<tr>
<td>3.7</td>
<td>Example: Applications mapped to software and hardware</td>
<td>43</td>
</tr>
<tr>
<td>3.8</td>
<td>Example: Real scenario including RTOS, concurrent execution and competition for shared resources (bus)</td>
<td>46</td>
</tr>
<tr>
<td>3.9</td>
<td>Visual representation of a single transaction</td>
<td>47</td>
</tr>
</tbody>
</table>
3.10 From single transactions to a full trace of primary and secondary trans-
actions .......................................................... 48
3.11 Comparison between the time representation in TLM and Funtime .... 48
3.12 First refinement: transaction interdependency .......................... 51

4.1 The IP Level – Individual IPs (Level A1) ............................... 54
4.2 $\text{En}_{\text{IUT}}^1 + \text{En}_{\text{IUT}}^2$ versus $\text{En}_{\text{IUT}}^1 \cdot \text{IUT}_2$ ................. 58
4.3 Energy change vs. data switching activity ............................... 59
4.4 Distribution of simultaneously switching bits per register ............. 60

5.1 The IP Level – Architecture of IPs (Level A2) .......................... 68
5.2 Leon3-based SoC Architecture .......................................... 68
5.3 Leon3-based McNoC Architecture ....................................... 69
5.4 Different cases considered for characterization of composite transactions 70
5.5 Cycles and energy statistics for the $\text{ld}$ instruction .................... 74

6.1 The Algorithmic Level (Level B) ....................................... 78
6.2 Validating the cache refinement .......................................... 83
6.3 Cycles variation as a function of the cache miss ratio .................. 84

7.1 The Refinement Level (Level C) - Refining bus contention .......... 86
7.2 Reference SoC ....................................................... 88
7.3 Execution time $e_{\text{x}_i}$ (in number of cycles) for FFT, Fibonacci and Viterbi 89
7.4 Execution time $e_{\text{x}_i}$ (in cycles) for FFT, $1 \leq N \leq 15$ ............... 90
7.5 MPSoC execution time for $N$ processors and $N$ different applications 95

8.1 The Refinement Level (Level C) – Refining the OS overhead ......... 100
8.2 Leon3-based platform .................................................. 101
8.3 Example of an objdump output ........................................ 102
8.4 RTOS Modeler ......................................................... 102
8.5 RR scheduling: $\forall T_i, t_{\text{rel},i} = 0$ and $e_{\text{x}_i} = EX$ ................ 105
8.6 Round Robin scheduling: $\forall T_i, t_{\text{rel},i} = 0, e_{\text{x}_i} \neq EX$ and $e_{\text{x}_i} \neq e_{\text{x}_j}$ 106
8.7 Round Robin scheduling: $\forall T_i, t_{\text{rel},i} \neq 0, e_{\text{x}_i} \neq EX, e_{\text{x}_i} \neq e_{\text{x}_j}, e_{\text{x}_i}(\text{rdy}(t))$ and $e_{\text{x}_i}(\text{rdy}(t))$ 107
8.8 3 independent tasks $T_1, T_2$ and $T_3$. $t_{\text{rel},i} \neq t_{\text{rel},j}$ .......... 109
8.9 3 inter-dependent tasks $T_1, T_2, T_3$ and $T_4$. $t_{\text{rel},i} \neq t_{\text{rel},j}$ 111
8.10 RTEMS-based signal-processing software ................................ 115
List of Publications


List of Acronyms

ABS  Anti-lock Braking System
API  Application Programming Interface
AS  Architecture Specification
CAD  Computer Aided Design
CLT  Central Limit Theorem
CPU  Central Processing Unit
DSE  Design Space Exploration
DSP  Digital Signal Processor
EDA  Electronic Design Automation
HDL  Hardware Description Language
HLL  High-Level Language
HLS  High-Level Synthesis
IP  Intellectual Property
IPC  Inter-Process Communication
ISS  Instruction-Set Simulation
ITRS  International Technology Roadmap for Semiconductors
MoC  Model of Computation
MPSoc  Multi-Processor SoC
NoC  Network on Chip
OS  Operating System
OSCI  Open SystemC Initiative
PBD  Platform-Based Design
RTL  Register Transfer Level
RTOS  Real-Time OS
SLD  System-Level Design
SLE  System-Level Estimation
SLOC  Source Lines of Code
SoC  System on Chip
TLM  Transaction Level Modeling
UCS  Use Case Specification
VHDL  Very high speed HDL
WCET  Worst-Case Execution Time
Chapter 1

Introduction

This chapter describes the motivation and the contribution of the present work. The main motivation lies in the lack of a quick and accurate estimation tool at the system abstraction level. Without such a tool, efficient design-space exploration at system level becomes impossible, due to the broadness of the design space at this level. As a consequence, taking the best architectural decisions becomes very challenging for system designers or for any system-level synthesis tool. However, since decisions taken at the system level are the most relevant in affecting the quality of the final design, it is very important to take them right from the beginning, in order to avoid costly and time-consuming reiterations. The contribution of this work is therefore the implementation of a fast and accurate system-level estimation methodology, which can really help system designers take the best architectural decisions early in the design cycle. The methodology operates at a very high abstraction level, namely the functional untimed level. For this reason it has been called Funtime.

1.1 Background: the Increasing Complexity of Electronic Devices

In the last decade, there has been a massive spreading of the electronic market. Devices like personal computers, smart phones, mp3 players, portable GPS systems, digital cameras, game consoles, plasma/LCD TV sets, digital decoders and a lot more have become part of most people’s daily life.

While users can take advantage of a steadily growing set of functionalities, performance and easiness of use, system designers are challenged by the growing design complexity. A path towards complexity is traced by the following two factors: progress in the semiconductor industry and increased applications demand.
1.1.1 Progress in the semiconductor industry

The progress in the semiconductor industry continues at the speed predicted by Moore’s Law \[50\], which means that the number of transistors per unit of area doubles every 18 months. This allows in turn a higher number of functionalities within a single hardware block, which directly leads to higher complexity. Figure 1.1 shows the increment of transistors count for Central Processing Unit (CPU)s from 1971 to 2008, in line with Moore’s prediction.

![Figure 1.1: CPU transistors count from 1971 to 2008](image)

1.1.2 Increased applications demand

The higher number of transistors per unit of area achieved on the hardware side provides enough computational power for also hosting more complex software. As a consequence, the Source Lines of Code (SLOC) keeps increasing, as well as the number of applications mapped to the same architecture. Figure 1.2 shows how the number of source code lines has increased with time in the different versions of the Windows operating system \[44\]: starting with \(~5M\) lines for Windows 3.1 in 1993, it reaches up to \(~50M\) lines with Windows Server in 2003. Interpolating the values reported in the table on the left of the graph results in a quadratic growth.
1.2. System Partitioning

Design partitioning is one of the two main countermeasures taken to manage the growing complexity. In general, this concept relies on the idea that one problem can be simplified by decomposing it into several smaller sub-problems. When applied to VLSI design, this means that a design is no longer seen as a whole flat sea of transistors; instead, it is considered as an aggregate of smaller and hierarchical sub-designs connected together. Multiple layers of hierarchy may coexist and each sub-design is normally representative of a certain functionality. In other words, hierarchy helps reduce complexity by hiding it within each sub-system block/module. Note that the partitioning process is a manual activity.

As complexity increases, so does the granularity of the basic building blocks. In addition, the trend goes towards a standardization of very commonly used blocks, which are usually known as Intellectual Property (IP)s. Examples of IPs are micro-processors, DSP cores, codecs, modems, etc. Such IPs are taken from a library and used as black boxes to build new system architectures, thus being a fundamental help in reducing design complexity and improving the time to market. Building new components from scratch is in fact very time consuming and expensive.

A further increase in the partitioning granularity has led to an extension of the IP concept and has led to the introduction of the Platform-Based Design (PBD) term. PBD goes beyond the standardization of individual IPs and proposes instead the standardization of entire platforms. Although different interpretations of PBD exist, the following definition given by Alberto Sangiovanni-Vincentelli can be taken as an example: a platform is “a layer of abstraction with two views: the upper view is the abstraction of the design below so that an application could be developed on the abstraction without referring to the lower levels of abstraction. The lower view is the set of rules that integrates components as part of the platform.” This definition suggests that a platform can be seen as an Application Programming Interface (API), thus allowing applications to be written without caring about the actual underlying architecture, as long as the interface rules are followed.
1.3 Abstraction: from Physical to System Level

The second countermeasure taken to manage complexity is raising the design abstraction level. While design partitioning reduces complexity by basically hiding it in each subsystem block, raising the design abstraction eliminates complexity, in the sense that automatic tools, known as Electronic Design Automation (EDA) tools, are normally developed to automatically deal with it through a process called synthesis. In other words, synthesis automatically accounts for complexity by translating a description made by the designer at a certain abstraction level into a lower level. Note that the increase of granularity proposed by system partitioning and the increase of abstraction usually go in parallel.

In 1983, Gajski and Kuhn derived what is today known as a Y-chart, which is a representation of the different abstraction levels at which a system can be modeled and estimated [25]. This is shown in Figure 1.3. Gajski and Kuhn distinguish five different abstraction levels, represented by concentric circles, each of which is classified according to three different domains: behavioral, structural, and physical. The innermost circle corresponds to the lowest abstraction level, while the outermost one to the highest.

![Figure 1.3: Gajski-Kuhn Y-chart](image-url)

Until a few decades ago, it was still possible to manually describe a system directly at the physical level, since the amount of complexity was very limited. When complexity grew, manual description at physical level became impossible and the starting point of the design flow was thus raised to a higher abstraction level,
1.4. SYSTEM-LEVEL ESTIMATION

i.e. the gate level. Place&Route tools were instead developed to automatically translate (synthesize) a gate-level description into a layout design. Similarly, as complexity kept growing, the starting point of the design flow was further raised up to the RT level and RTL synthesis was introduced to automatically translate RTL into gate level. Two very well-known examples of languages used for RTL description are Very high speed HDL (VHDL) and Verilog. Along with the increase of complexity, today’s trend is to further shift the entry level for automatic synthesis up to the system level. The idea with System-Level Design (SLD) is to abstract away even the register-transfer-level details and, instead, focus on describing the system functionality and how this has to be mapped to the underlying architecture.

1.4 System-Level Estimation

Shifting towards higher levels of abstraction has proved to be a winning strategy for dealing with increasing complexity. Indeed, by abstracting away the lower-level details, implementation is faster, which means lower engineering effort, lower cost and lower time to market, as well as higher productivity. Decisions made at the system level have a very strong impact on the quality of the final product, since the degree of achievable optimization is normally proportional to the abstraction level and, indirectly, to the point in the design flow where decisions are taken: the earlier the better. At the system level, the question that system architects have to answer to is the following: given a set of applications and a set of possible architectures, what is the best architecture on which to map this set of applications (also refer to Figure 1.4)? The expression best architecture refers to the properties of an architecture in terms of metrics like performance, power consumption and silicon area, for a given set of applications. For example, what is the power and performance impact of using a voltage-frequency scaling scheme rather than a fixed frequency? What is the power and performance impact of varying the number of levels in the memory hierarchy? What is the best interconnect to use: a bus or a NoC? What is the advantage/disadvantage of implementing part or the whole set of applications in hardware rather than software? These are just examples of the hard choices a designer has to make. Since they are so important, taking the right system-level decisions from the beginning is crucial, especially when complexity grows: any error at this early stage would lead to annoying design reiterations, as shown in Figure 1.4 with a consequent high loss of time, money and, probably, a sub-optimal final implementation.

However, although very important, decisions at system level are very hard to take and this is for two main reasons: the first is that, at the system level, the design space to consider is extremely broad as a consequence of the limited amount of implementation details available. Figure 1.5 shows the relation between design space width and abstraction level. The second reason is that the impact of the decisions taken at system level is not known until a very late stage of the design process, which can take months of work.
From the second reason mentioned above, it can be concluded that the lack of a quick and accurate System-Level Estimation approach is one of the main obstacles to successful system-level design today. In fact, if an efficient system-level methodology for energy and performance estimation was available, it would be possible to carry out a reasonably comprehensive Design Space Exploration (DSE), and thus judge from the beginning of the design flow which architecture is the most suitable for a certain applications domain, in terms of performance and power consumption. In addition, estimation at any abstraction level is a requirement for the implementation of automatic synthesis tools, since it is only after estimation that the tool can judge what the best solution is.

Efficient estimation at lower abstraction levels has allowed us to have quite mature automatic synthesis tools today. Estimation at the physical level requires accounting for the individual capacitance and resistance contributions coming from each transistor and interconnecting wire. Estimation at this level is extremely accurate, but also very slow. Simulation at physical level is also very slow and is thus feasible for only very small designs and for a very short design execution time.
1.4. SYSTEM-LEVEL ESTIMATION

At the gate level, estimation is simplified by the fact that standard cells are used, whose physical properties are pre-characterized. Only the impact of cell-to-cell connecting wires has to be estimated separately, which is done using so called wire load models. Estimation at this level is less accurate, although faster, and bigger design sizes can be simulated. At the RT level, Hardware Description Language (HDL) languages are used to describe in words what RTL synthesis translates into logic gates. Simulation is very common at RT level and reasonably fast for medium-size designs running very short chunks of application. However, estimation made at this level loses accuracy due to the lack of enough physical details. In general, the increase of the abstraction level is directly proportional to an increase of the estimation speed and inversely proportional to the estimation accuracy.

When it comes to system level, the lack of an efficient estimation methodology has been an obstacle to having mature automatic system-level synthesis tools available today. In fact, the operation of mapping the system-level functional description to the actual architecture is still largely done manually. The decision making approach used by system designers has been mostly relying on their acquired experience, on the comparison with previous designs and on rules of thumb. However, while this approach can still work with small/medium-size systems, its application to today’s more and more complex systems has become unrealistic and the need for a more systematic and accurate approach has become a necessity. TLM has appeared at the beginning of the last decade as a simulation-based approach raising the abstraction level above RTL and as a starting point for synthesis. In essence, TLM abstracts away the RTL details and models functionality and communication among the system modules. Communication is seen as an exchange of transactions between architectural resources. As a result, TLM has proved to be much faster than RTL [26]. In spite of that, even TLM could be too slow to allow proper simulation of future complex systems. More details about the TLM methodology are given in Chapter 2. In addition, the problem remains of how to obtain for example accurate power estimation at the system level, since TLM does
CHAPTER 1. INTRODUCTION

not provide intrinsic support for power estimation, and waiting until reaching the
gate-level design phase is not an option. The natural question that comes as a
conclusion of the above discussion and that is thus also the motivation behind this
thesis work is as follows:

As it has been discussed that system-level estimation is essential for a successful
system-level design, how is it possible to implement a fast and accurate methodology
for efficient system-level estimation?

Answering this question is instead the contribution of this thesis.

1.5 Thesis Contribution

The present work contributes to the area of system-level estimation by proposing
a system-level estimation framework for energy and performance prediction, which
can indeed help system designers take the best architectural decisions early in the
design flow.

The approach consists of three layers. The bottom layer is called IP Level (Level A) and relies on building a library of IP energy and performance models, where individual IP functionalities are pre-characterized in terms of number of cycles and energy consumption. This activity is justified by the fact that, as discussed before, the Design&Reuse concept has become a common industrial practice, and the modeling process is therefore only a one-time effort. Such models can come in the form of look-up tables or mathematical expressions. Characterization is done at the gate level and back-annotated with physical design data to enable highly accurate characterization. The availability of a physical layout for each IP also allows preliminary floorplans to be made for different architectures. This in turn enables us to get reasonably good estimates of the global wires length, which also plays a critical role in affecting the overall system energy and performance. Note that the characterization activity performed at this level is the prerogative of the IP provider and not of the system designer, who is the tool end user. As an example, Chapter 4 describes the implementation of an energy and performance model for a SPARC-based Leon3 processor. The impact that the external communication infrastructure – connecting IPs to each other – has on the individual IP properties is also taken into account. Chapter 5 discusses the case of switching from a bus-based to a NoC-based architecture.

The intermediate layer is called Algorithmic Level (Level B) and is where the
actual estimation takes place. As opposed to the IP Level, this level directly concerns the system designer. At this level, applications are run and profiled on a development host (a common PC). This allows us to create a trace of the executed source code, which is then mapped to the assembly code of the target architecture. This operation allows a trace of target instructions to be indirectly built without having to run the applications natively on the target architecture. This has a few clear advantages: first, the engineering effort is kept low, since there is no need of implementing a high-level simulation model of the target architecture,
for instance a transaction-level model. Second, the level of abstraction at which the approach proposed in this thesis performs estimation is the functional untimed level – from here the whole approach has been called Funtime – which is higher than the transaction level. As a consequence, estimation is faster. Once the target trace is inferred, energy and performance figures can be extracted by using the IP models from the bottom layer. To make the whole process possible, changes have been made to the GNU GCC compiler. Estimation examples are shown for a set of common image/video codec applications in Chapter 6.

The top layer is called Refinement Level (Level C) and accounts for non-idealities neglected at the layer below, such as the presence of caches and the fact that multiple applications normally run concurrently, share the same resources and are controlled by an operating system. Statistical models are built to predict the energy and performance impact of each of these components through extensive simulation. However, this is a one-time activity. When estimation is carried out, these statistical models are used and no simulation is run at all. An MPSoC hosting up to 15 processors and using both fixed-priority and round robin bus arbitration is used for modeling bus contention. This is part of Chapter 7. The RTEMS operating system is taken as a reference to model the OS energy and performance impact. OS modeling has considered both the round robin and fixed-priority scheduling cases. This is part of Chapter 8.

Validation for each layer is also carried out. The results show that the approach is within 15% of gate-level accuracy and exhibits an average speed-up of ~30X when compared to transaction-level modeling (TLM).

Note that the use-case size and the amount of factors that have to be taken into account when doing system-level estimation are extremely large. Since it was not possible to deal with all these aspects within this work, the Funtime framework presented in this thesis comes as a proof of concept. Smaller use-cases have been used as case studies and sensible simplifications have been made, the goal being to show the overall feasibility of the Funtime approach as a system-estimation tool. Further extensions are required to make this framework more general and complete, as is discussed in the future work section, at the end of the thesis.

The contribution of this thesis can also be classified into three components:

- **Concept-related**: this component concerns the definition of the Funtime estimation framework and the actual implementation of each of its layers. Starting from an initial abstract idea, each layer has in time been shaped and enriched with increasing details.

- **Tool-related**: each of the three layers has required consistent scripting work, aimed at speeding up and automating operations that could not have been carried out manually. For instance, at the IP Level, scripts have been written to automate the Leon3 processor characterization. At the Algorithmic Level, scripts have been written to profile the application running on the development host and to map it to the target. This activity has also implied a set of
changes to the GNU GCC compiler. Finally, at the Refinement Level, scripts have been written to automate the OS characterization process. Configuring the RTEMS OS has also required quite some effort, including writing basic driver routines.

- **Experiment-related:** experiments have been conducted throughout the entire development of the Funtime methodology to validate both its estimation accuracy and speed. At the IP Level, the accuracy of the Leon3 energy and performance model has been validated against gate-level measurements for a set of common applications. At the algorithmic level, the same applications have been used to test the accuracy of Funtime in inferring a trace of target transactions and in estimating its energy consumption. At the Refinement Level, experiments have been conducted to verify the accuracy of Funtime in inferring the OS overhead, the bus contention overhead, as well as the effect of caches. A transaction-level model has been also used to validate Funtime’s estimation speed. The target architectures used as a reference for the experiments are a shared-bus MPSoC based on the AMBA AHB bus, and a NoC using deflection routing algorithm. Leon3 has been taken as the reference processor.

The following publications have been used as a basis for the present thesis:

- Paper [61] gives first an overview of the three layers of the Funtime approach and then presents in more detail the bottom and intermediate layers. The work presented in this paper has been largely conducted by the author of the thesis, who has defined the details of the Funtime layers and has produced part of the numerical results. Luca Bolognino was a master’s student who helped produce the remaining part of the numerical results. Ahmed Hemani contributed by giving feedback and by conceptually discussing with the thesis author the role played by each of the Funtime layers.

- Paper [53] proposes an instruction-level characterization of the Sparc-based Leon3 processor. The resulting model, in the form of look-up tables, reports number of cycles and energy for each processor instruction. The steps of the characterization methodology proposed in the paper have been decided by the author of this thesis. Luca Bolognino was a master’s student who helped the author by doing the tool-related work, according to the guidelines given by the thesis author. Ahmed Hemani has supported the author with useful feedback.

- Papers [58] and [57] investigate the operating system overhead in terms of energy and performance. Characterization is first carried out by executing extensive simulation and energy extraction at the gate level. The results from characterization are then used to implement a high-level model for rapid and accurate OS overhead prediction. Paper [58] considers Round Robin scheduling, while paper [57] considers priority-driven scheduling. The work
presented in these papers has been conducted by the author of this thesis and supported by feedback from Ahmed Hemani and Ingo Sander.

- Paper [59] investigates the effects of bus contention on energy and performance in MPSoCs. Characterization is first carried out by executing extensive simulation and energy extraction at the gate level. The results from characterization are then used to implement a high-level model for rapid and accurate prediction of bus contention. The work presented in this paper has been conducted by the author of this thesis and supported by feedback from Ahmed Hemani and Ingo Sander.

The following publications, although not used as a basis for this thesis, are reported for the sake of completeness:

- Paper [54] is an initial concept paper which gives a general idea of the Funtime approach and its layers. The author of this thesis and Ahmed Hemani have equally contributed to this work.

- Paper [56] implements an energy and performance model for a switch of the Nostrum NoC. The model reports the switch energy per clock cycle, based on the variation of the switch inputs. The entire work has been conducted by the author of the thesis and supported by the feedback from Axel Jantsch.

- Paper [55] presents an initial work on the Algorithmic Level, where a set of applications is used to validate the accuracy of Funtime in inferring a target execution trace. No energy estimation results are presented in this paper, however. The work presented in this paper has been mainly conducted by the author of this thesis. Mohammad Badawi was a master’s student who helped with the experimental part. Ahmed Hemani has contributed with useful feedback.

- Paper [60] extends the paper in [55] by also validating Funtime estimation speed against a TLM implementation. The work presented in this paper has been mainly conducted by the author of this thesis. Mohammad Badawi has helped with the implementation of the transaction-level model. Ahmed Hemani has contributed with useful feedback.

- Journal paper [62] is an invited work, as a follow-up of the VLSI 2009 conference. It merges and partially extends the contents of the papers in [61] and [53]. The journal paper has been written by the author of this thesis.

1.6 Thesis Layout

Chapter 2 presents an overview of the most common approaches to system-level design/estimation in use today and compares them to the Funtime approach. Large space is given to the description of simulation-based approaches to system-level
design/estimation, with particular focus on Transaction Level Modeling (TLM). The reason is that TLM is a very common approach. Examples of TLM usage in real projects are then proposed, which also include modeling of Multi-Processor SoC (MPSoC)s and Operating System (OS). A minor section is left for discussing analytical approaches and combinations of multiple approaches.

Chapter 3 presents an overview of the whole Funtime approach. After describing the methodology user interface, a description follows of the purpose of each of the three layers of which Funtime consists. In particular, taking a bottom-up approach, an IP Level (Level A), an Algorithmic Level (Level B) and a Refinement Level (Level C) are identified. Afterward, the chapter continues by covering miscellaneous aspects relative to Funtime, with the purpose of providing a more complete and consistent picture of the approach.

Chapter 4 and 5 go into the details of the IP Level. Chapter 4 shows how to build an energy and performance model for a processor IP, namely for a Leon3. Chapter 5 shows instead how to account for the effects of IP-to-IP external communication when building IP models. A bus-based SoC and a NoC-based SoC are compared.

Chapter 6 gives further details about the Algorithmic Level. This chapter shows examples of energy and performance estimation on common applications. It also shows the accuracy and speedup obtained by Funtime against gate-level extraction and TLM respectively. Due to its tight binding with the Algorithmic Level, this chapter also includes two refinements from the Refinement Level, concerning transactions interdependency and the effects of caches.

Chapter 7 and 8 present extensively two other refinements. In particular, Chapter 7 proposes a high-level method to predict bus contention in MPSoC’s, while Chapter 8 presents a high-level method to predict the overhead of OS.

Chapter 9 draws the conclusions on the entire work and leaves space for some future work.
Chapter 2

Common Approaches to System-Level Estimation and Design Today

This chapter presents a survey of the most important methodologies for system-level estimation and design found in the literature, and compares them to Funtime. Three categories of tools are identified: simulation-based tools, analytical tools and tools that are a combination of multiple approaches. While discussing each of these categories, larger space is dedicated to simulation-based tools and, in particular, to describing SystemC/TLM. The reason is that Transaction-Level Modeling has become quite popular today, both in industry and academia, and is taken as a reference methodology to which Funtime is compared throughout this thesis.

2.1 Introduction

In the previous chapter, the conclusion was drawn that being able to efficiently carry out system-level estimation (SLE) is a prerequisite to making design-space exploration (DSE) at system level possible. In turn, DSE is critical to enable a successful system-level design (SLD), since it both helps system designers take the best architectural decisions and it is also an essential component for system-level synthesis.

Due to their importance and to the lack of a standard approach, both system-level estimation and system-level design in general are a hot research topic today and the focus of a high number of research groups. The present chapter presents a survey of the most significant estimation tools for SLD found in the literature. In doing so, the surveys presented in [79] and [28] are partially taken as a reference and adapted to the purposes of this work. At the same time, a comparison is presented between these approaches and Funtime, which emphasizes key similarities and differences.
CHAPTER 2. COMMON APPROACHES TO SYSTEM-LEVEL ESTIMATION AND DESIGN TODAY

System-level estimation tools can roughly be classified into three broad categories: simulation-based tools, analytical tools, and tools that are a combination of different approaches. Although the following subsections review each category, large space is dedicated to the simulation-based approaches and, in particular, to describing SystemC and Transaction Level Modeling (TLM). The reason is that this simulation-based approach has lately gained consensus and has become quite popular in both the industrial and academic community. This is why SystemC/TLM is also used in this work as the reference system-level approach when validating Funtime for estimation speed.

2.2 Simulation-Based Estimation Tools

As the name suggests, these tools rely on simulation to produce estimation results. Simulation allows us to trace the behavior of the system in specific states and given a certain set of input stimuli. Simulation-based approaches are therefore suitable for non-deterministic system behavior and their results are generally representative of the average-case scenario. Simulation-based approaches have two drawbacks: one is the large engineering effort required to develop a model of the system – the architectural model – and the second is that the simulation of an application model in such an architectural model is very slow and for large and complex use cases the simulation time can be unreasonably large. Below, a few simulation-based approaches are presented.

Ptolemy [19] is a framework capable of modeling and simulating concurrent and hierarchical heterogeneous systems. Hierarchy is supported in that the whole system model can consist of a tree of nested sub-models. At each level of hierarchy, such sub-models are composed to form a network of interacting components. Each local network is constrained to be homogeneous, whereas heterogeneity is allowed between networks at different levels of hierarchy. The interaction mechanism within a certain local network of components handles both data and control flow among the local components. Such an interaction mechanism is called model of computation (MoC). Note that, since different MoCs are allowed at different levels of hierarchy, different parts of the system can be modeled at different levels of abstraction, depending on the required degree of detail.

More concretely, Ptolemy relies on an actor-oriented view of a system, where actors are concurrent components communicating with each other by using communication channels connected to the actor’s ports. Every actor can either run in its own thread or multiple actors can run sequentially in a unique thread. This is determined by the local MoC. An actor is called atomic if it is at the bottom of the hierarchy, while it is called composite when it contains other actors. The implementation of a MoC related to a composite actor is called domain. A domain defines both the communication semantics and the order of execution among the local actors. In detail, communication is controlled by receivers, which are located
2.2. SIMULATION-BASED ESTIMATION TOOLS

at each actor input port and are unique for each channel. Receivers can represent FIFOs, mailboxes, proxies for a global queue or rendezvous points. The execution order of the local actors is instead controlled by a director.

Figure 2.1 shows a hierarchical example with two domains which are, by definition, associated with two composite actors. The top-level actor contains a director D1, an atomic actor A1 and a composite actor A2, which contains a director D2 and the atomic actors A3 and A4. D1 controls the execution order of A1 and A2, while D2 the execution order of A3 and A4.

![Figure 2.1: Ptolemy hierarchical model using two different domains. Source: [19].](image)

**SpecC** [24], [13] is both a system-level design language and methodology in which computation is separated from communication. Computation happens inside so called behaviors, while communication is implemented by channels. Channels are connected to behaviors through behaviors’ ports, as long as the respective interfaces match. Hierarchy is also supported, in that both behaviors and channels can reside within a parent behavior. Execution occurs within each behavior and synchronization among behaviors is implemented by events. An example of a SpecC design is shown in Figure 2.2.

SpecC is based on the C language, which is extended with hardware and software modeling constructs. As will be evident in the next few subsections, SpecC is in many respect similar to SystemC, as this also allows a separation between computation and communication. However, SystemC is an extension of C++.

**Metropolis** [8] provides a tool-set targeting embedded systems development, which supports simulation, formal analysis and synthesis. The authors state that using a unified framework, rather than a collection of unlinked tools, as is the case today, can considerably speed up the entire design flow, by making it more efficient and less error prone.

The Metropolis infrastructure relies on a so called metamodel, which is a model with precise semantics and general enough to support both existing and new models of computation. In this respect, Metropolis includes a standard API, which allows feeding the tool with inputs coming from any external tool. The Metropolis metamodel is a language, similar to SystemC, that specifies networks of concurrent objects. This metamodel can be used to represent function, architecture, mapping
A function is seen as a set of objects – called *processes* – that concurrently carry out some actions while communicating with each other through *ports*. Ports come with *interfaces*, which declare a set of methods usable by the process through the port. The interface methods are actually implemented in other objects called *media*, which are used to connect ports to each other. Channels are the SystemC equivalent of Metropolis media. The behavior of this network of processes is modeled as a set of executions, where each execution consists of a sequence of events. Events represent a program’s entries or exits to some piece of code. Note that Metropolis is in many respects similar to Ptolemy, SpecC and SystemC, in that they all rely on the concept of concurrent processes that communicate via channels.

An architecture is identified in Metropolis by two aspects: the functionality that it has to implement and the efficiency of such implementation. As was discussed above that functionality is basically the expression of some services implemented by methods, efficiency is measured by accounting for the cost of such services. This is done by annotating the cost of each atomic event executed within a process. So called *quantity managers* are used for this purpose. The decomposition of services into event sequences is done by using networks of media and processes, as is also done for the functional model. Architecture networks often match the actual physical structure of the architecture.

The Metropolis metamodel also takes care of mapping the functional model to the architectural model. This is done by using a new network – called *Mapping Network* – that contains the functional and architectural networks and synchronizes one to the other by means of events. Figure 2.3 shows an example of a functional model on the left side and of a corresponding architectural model on the right side. Note the presence of quantity managers in the architectural model. The text with green background describes instead the mapping networks that maps and
synchronizes the functional and architectural models to each other.

The Metropolis metamodel also supports the concept of platforms, in that it allows us, while keeping the functional model unchanged, to sweep across a number of different architectures and thus to evaluate the cost impact of each architecture considered.

Figure 2.3: Mapping network in Metropolis. The functional network on the left side and the architectural network on the right side. Source: [8].

SPADE [41] stands for System-level Performance Analysis and Design-space Exploration. It proposes a methodology for exploring signal processing architectures at system level. Applications are modeled as a network of concurrent communicating processes, based on the model of computation provided by Kahn Process Networks. Communication happens via channels that are bounded to processes’ ports. When executing, each process produces a trace of application events, which represent the application workload. Traces are then passed as an input of corresponding architectural models, which associate a defined latency to each trace event. An example of functional and architectural model, as well as their mapping, is shown in Figure 2.4.

Artemis [63], [64] stands for Architectures and Methods for Embedded Media Systems. Artemis targets the following two challenges: first, developing a modeling and simulation environment for efficient design-space exploration of heterogeneous embedded systems; second, investigating the possibility of using reconfigurable embedded architectures – such as FPGAs – to give high performance for specific applications and limiting power consumption. The Artemis development environment is heavily based on the SPADE framework. The left side of Figure 2.5 shows the operation of mapping an application model to an architecture model in Artemis.
CHAPTER 2. COMMON APPROACHES TO SYSTEM-LEVEL ESTIMATION AND DESIGN TODAY

Figure 2.4: Mapping between the application and the architecture model in SPADE. Source: [41].

– note the similarity with SPADE (in Figure 2.4). The right side shows instead how Artemis integrates the design-space exploration activity and the use of reconfigurable architectures.

Figure 2.5: Left: mapping between the application and the architecture model in Artemis. Right: integration between DSE activity and use of reconfigurable architectures. Source: [64].

MILAN [47], [49] stands for Model-based Integrated simuLAtioN. It is a hierarchical methodology for embedded system design, estimation and design-space exploration, which can integrate third-party simulators and tools into a unique environment. MILAN’s main focus is energy-efficient design of signal-processing
applications. The following steps are identified in this framework flow: application models are first created using synchronous data flow (SDF) graphs. MILAN supports hierarchical modeling of SDF graphs. Functional simulation is enabled by the generation of high-level source code in C or Matlab and by the integration of functional simulators. Second, a model for the architectural resources is created, on which the user defines a set of performance constraints, in terms of latency and energy. Once these steps are completed, the user invokes the DSE tools. The predefined DSE tool in MILAN is called DESERT. This tool relies on Ordered Binary Decision Diagrams for constraint satisfaction. The authors also report that DESERT is able to cover a design space of around $10^{20}$ to $10^{40}$ designs in a few minutes. The output of DESERT is then passed to and evaluated by the High-level Performance Estimator (HiPerE) tool. This tool estimates system-level energy dissipation and latency. Estimation is carried out at the task level abstraction, which confers the tool high speed. Both this tool and the actual architectural model are based on a so called General Model (GenM) [48]. The designs selected by HiPerE are then passed to lower-level simulator/estimator for the final design selection. Figure 2.6 shows the complete MILAN flow. From left to right, it is possible to identify the user’s application and architectural resources model, together with the constraints definition; the usage of DESERT and of HiPerE is also shown.

![Figure 2.6: MILAN design, DSE and estimation framework. Source: [49].](image)

**TAPES** [76] stands for Trace-based Architecture Performance Evaluation with SystemC. In order to keep simulation as fast as possible, the functionality of each resource is modeled as a sequence of processing delays interleaved with external transactions, and resources are considered as black boxes. Such a sequence is called a trace. External transactions are typically read/write transactions used to model
communication which, as opposed to functionality, is not modeled with the same degree of abstraction. A more detailed model for communication is considered necessary in order to properly account for the occurrence of contention events. At the communication level, the read/write transactions emitted by functional models are thus expanded to account for the actual communication behavior. An example is shown in Figure 2.7: the trace on the top reflects the functionality of a CPU, which emits two consecutive read transactions, executes some internal functionality modeled by a delay, and finally emits a write and read transaction. The trace at the bottom shows instead how the CPU trace is transformed during architectural simulation, to account for the bus and memory accesses.

Figure 2.7: TAPES trace example. Source: [76].

TAPES is not only an estimation tool, but also a framework in which to carry out architectural exploration. Besides high simulation speed, this also implies that fast changes of the system architecture must be made possible. For this purpose, the hardware configuration of the simulation model is dynamically changed at the beginning of the simulation according to a system configuration file. This means that users do not have to manually modify the SystemC description. Note that the trace specification for the system architecture is instead a manual process, and the timing behavior of the resources is taken either from data sheets or, for the CPU, by using an instruction-set simulator.

MESH [52], [46] stands for Modeling Environment for Software and Hardware. It is defined by the authors as a thread-level simulator, as opposed to traditional instruction-set simulators. In this way, the authors want to emphasize that MESH increases the granularity for which estimation is carried out and thus the simulation speed can be much higher. MESH is a three-layer approach, which considers resources (hardware blocks), software and schedulers. Such layers are modeled by software threads on the evaluation host. Software threads are annotated with time budgets for the corresponding hardware elements. Such time budgets are extracted beforehand by estimation or profiling. Scheduler threads work as arbiters for the software threads. Power estimation capabilities are also implemented in MESH. As
2.2. SIMULATION-BASED ESTIMATION TOOLS

far as microprocessors are concerned, the authors rely on the fact that compilers
tend to produce quite regular instructions patterns, from which a power estimate
can be extracted, that is representative of the average case.

As part of the ASSET project, Joshi et al. [34] propose a performance evaluation
methodology for system-level design exploration. The application behavior
is modeled as a set of statistical parameters, which are generated either by static
analysis and profiling of the application, or by using some simulation framework
like SimpleScalar [7]. Application parameters are independent of the architecture
and their extraction is a one-time activity. The target architecture components
are instead modeled using SystemC. A set of components on which to map the
application models is taken from a library. Building these components also relies
on probabilistic models, which are extracted by making a compromise between analytical and cycle-accurate simulation, and which only account for the interaction
the component has with the outside, while the internal functionality is modeled as
a delay. Note that, during the mapping phase, application parameters must also
match with the parameters of the component the application is mapped to. For
example, an application parameter could be the distribution of load instructions,
whereas the corresponding architectural parameter could be the number of words
be loaded.

J. Kreku et al. in [35] also present a methodology for system-level design and
performance evaluation. Their work relies on describing application workloads in
UML and platform services in SystemC. The methodology is meant to enable early
system-level performance modeling and evaluation through transaction-level simu-
lation, which also allows timing information to be collected.

Simulators have also been developed to investigate the properties of some pro-
cessor micro-architectures. Such simulators are often cycle-accurate. Some exam-
ple are SimpleScalar [7] or SimOS [65]. Simics [42] is instead a virtualization
framework that allows entire platforms to be emulated.

2.2.1 An overview of SystemC and Transaction Level
Modeling (TLM)

Among the simulation-based approaches, those relying on SystemC/TLM or an
equivalent modeling style deserve a dedicated section. The reason is that, in recent
years, SystemC/TLM has become quite popular and has found a relatively wide
range of applications both in academia and industry. SystemC’s popularity is in
part also the result of backing by leading industrial players like Coware (now part
of Synopsys), Synopsys and Cadence.

SystemC is both an HDL and a High-Level Language (HLL) [30]. In fact, it
models hardware at a higher abstraction level than Register Transfer Level (RTL)
and, in order to do this, it uses C++ as a programming language. Higher ab-
straction level means higher simulation speed but also less accuracy. The way Sys-
temC/TLM trades-off these two important metrics has been characterized in [67].
The Open SystemC Initiative (OSCI) started back in 1999, promoted by companies like Coware and Synopsys, and had the ambition of creating an HDL that could model hardware at a higher level than commonly-used languages like VHDL and Verilog. The main motivation was to improve both the implementation and the simulation efficiency compared to RTL, which has proved to be a bottle neck in modeling a system-level architecture. SystemC is an extension of C++, in the form of a hardware-oriented library of C++ classes. This is illustrated in the bottom part of Figure 2.8.

TLM is not an HLL in itself, but rather a library of functions built on the top of an HLL which is very often SystemC (see Figure 2.8). In the TLM terminology, a transaction represents the information being exchanged between the different system modules. TLM is particularly interested in separating the computational component from the communication component. For this purpose, TLM provides constructs to efficiently model the inter-module communication component, while the intra-module computational component is generally modeled at the functional/behavioral level. Standard routines have been implemented in TLM which model unidirectional versus bidirectional and blocking versus non-blocking communication. Communication is modeled using channels, interfaces and ports, which are objects provided by the underlying HLL.

In the context of this thesis, TLM 1.0 is used as a reference. Note also that SystemC/TLM is meant to be used for system-level modeling and performance estimation, but it does not provide any intrinsic support for evaluating other metrics, such as power for example.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Predefined channels</th>
<th>Utilities</th>
<th>Core Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-valued logic type</td>
<td>Signals</td>
<td>Tracing</td>
<td>Modules</td>
</tr>
<tr>
<td>4-valued logic vectors</td>
<td>Clock</td>
<td>Report handling</td>
<td>Processes</td>
</tr>
<tr>
<td>Bit vectors</td>
<td>FIFO</td>
<td></td>
<td>Events</td>
</tr>
<tr>
<td>Finite-precision integers</td>
<td>Mutex</td>
<td></td>
<td>Ports</td>
</tr>
<tr>
<td>Limited-precision integers</td>
<td>Semaphore</td>
<td></td>
<td>Exports</td>
</tr>
<tr>
<td>Fixed-point types</td>
<td></td>
<td></td>
<td>Interfaces</td>
</tr>
</tbody>
</table>

**Figure 2.8: C++, SystemC and TLM relation.**
2.2.2 Time representation in TLM

The way in which time is modeled represents a key factor when considering hardware modeling. In addition, time representation equally concerns the computation and communication components. This is summarized in Figure 2.9 where the communication and computation components are shown on the horizontal and vertical axis respectively.

![Figure 2.9: Time abstraction levels. Source: [9].](image)

RTL models time with a very fine granularity, which has the size of a clock cycle for both components. The clock signal controls the advance in time of the simulation kernel. A direct consequence of this behavior is great accuracy, but also a slow simulation speed.

Unlike RTL, TLM gives a much higher flexibility when it comes to modeling time and provides different granularities, ranging from cycle-timed (or cycle-accurate) to untimed when considering the computation domain, while only approximate-timed and cycle-timed when considering the communication domain. This means that some notion of time has still to be maintained in TLM. The term approximate-timed means that the advance in time happens with the granularity of a transaction. As the name suggests, an untimed implementation instead does not model the notion of time at all, thus allowing a higher simulation speed.

On the very top of the timing abstraction level, it is visible what in the figure is identified as SAM, which basically corresponds to a software model. In this case, no timing information is present, either in the computation domain or in the communication domain. In essence, the software model is a functional/algorithmic model, which has no notion of the architecture used to implement such functionality. Modeling at this level certainly gives the greatest benefits in terms of implementation
and simulation speed, but at the same time introduces the biggest challenges to preserving estimation accuracy. It is at this level that the estimation approach proposed in this work, i.e. Funtime, operates. In the upcoming chapters, challenges, advantages and drawbacks of this choice are discussed.

2.2.3 TLM advantages

Based on what has been said so far, using TLM has some clear benefits:

1. Compared to a traditional RTL approach, TLM allows a higher implementation speed. Implementation speed comes from the fact that all the RTL details are abstracted away. In practice, this means that the actual implementation details of the computational blocks are omitted and replaced by a functional/behavioral description. Behavioral description is faster to implement than architectural description. In [26], the author reports a speedup of up to 10X when modeling in TLM compared to RTL.

2. Compared to a traditional RTL approach, TLM also enables a higher simulation speed, which in [26] is estimated to be up to 1000X higher than RTL. This result is mainly related to the time representation in TLM. From what was discussed above, an untimed implementation in the computational domain combined with an approximate-timed implementation in the communication domain give the best TLM simulation performance.

3. F. Ghenassia in [26] states that using the TLM design flow can allow a more efficient HW/SW co-design. This is shown in Figure 2.10. In essence, the TLM flow would allow a concurrent development of hardware and software: the architectural TLM of the hardware infrastructure enables early software development and verification of hardware software interfaces. This is also a consequence of the fact that, in the classic design flow, software is usually implemented in C/C++, while hardware in VHDL/Verilog. In the TLM flow instead, both the software and hardware models are implemented in C/C++; thus concurrent testing becomes more feasible and the overall design time shorter.

4. According to Yi et al. in [78], TLM, and SystemC in general, have helped to solve some annoying synchronization issues. This aspect is mainly related to the fact that, in a simulation-based approach, multiple simulation models may have to run together and synchronize to each other. The introduction of SystemC has contributed to reducing the severity of this problem by providing a homogeneous environment for hardware/software co-simulation, thus replacing the considerable Inter-Process Communication (IPC) overhead with a light-weighted thread switch overhead. However, Yi et al. state that “for more accurate cosimulation, ISSs are typically used in SystemC cosimulation [...] ISSs are attached to a cycle accurate transaction level communication architecture model and communicate via IPC.”. Handling and reducing the IPC
2.2. SIMULATION-BASED ESTIMATION TOOLS

2.2.4 TLM drawbacks

While TLM has many benefits as discussed, it also has some drawbacks. The work proposed in this thesis overcomes these drawbacks.

1. Although the modeling effort happens at a higher abstraction level than RTL and is therefore less time-consuming, TLM is still based on a programming language. This means that the system designer has first to learn how to use this language – learning SystemC in this case – and then s/he has to implement the system using this language. Both the learning and implementation process imply a considerable engineering effort and, since the RTL modeling step cannot yet be replaced by HLS the TLM step is an extra step in the overall SoC design flow.

5. Finally, SystemC/TLM is considered the right modeling language from where High-Level Synthesis (HLS) should start. An example to substantiate this claim is the CtoS tool sold by Cadence. Starting from a SystemC description, this tool outputs RTL code. In spite of this, HLS is far from being as mature as logic synthesis and therefore requires more research.
2. Although the TLM simulation is more efficient than the RTL time, yet TLM relies on the availability of a hardware simulation model. Simulation of the target system on a host system, which is usually a common x86 PC, is time-consuming and will always introduce a significant overhead compared to native execution. Keeping in mind that the overall HW/SW complexity of SoCs is steadily increasing, even TLM at some point might become too slow to simulate complex and real use-case scenarios.

2.2.5 Examples of SystemC/TLM usage in RTOS and MPSoCs modeling

In [77], Yi et al. present an approach to Real-Time OS (RTOS) modeling that relies on pre-characterizing the key RTOS components, like the tick interrupt (triggered by the system timer) and the context switch. The authors rely then on Instruction-Set Simulation (ISS) to run their applications and measure the task duration, in order to derive the actual number of RTOS occurrences. Once this number is known and thanks to the pre-characterization activity, the total RTOS impact can thus be estimated. In [78], Yi et al. extend the work done in [77] by using a trace-driven co-simulation in SystemC. In [11], Brandolese et al. also propose a method to pre-characterize system calls of an OS for embedded applications. To do this, they rely on measurements “based on executing stubs, i.e. suitable programs calling a given function for a large number of times under possibly varying conditions”. However, they only characterize the impact on latency and neglect the impact on energy.

In contrast to [77], [78] and [11], Hessel et al. [32] do not pre-characterize the RTOS, but propose an RTOS model in SystemC, by extending the built-in scheduler that also does the power estimation. The authors state that “This estimation is based on previous analysis of the scheduling algorithms. The estimation parameter is update [sic] by back-annotation techniques.”. However it is not very clear what this analysis consists of and at what level the back-annotation is done. In [33], the authors extend the work done in [32] by introducing an abstract RTOS scheduling model that “[...] enables the system designer to quickly evaluate different scheduling policies and make the best choice in early design stages.”. Partially based on [33], Shaout et al. [69] define and provide the primitives for an ideal API for a generic RTOS model in SystemC.

In a similar fashion, also Le Moigne et al. [40] describe “a generic RTOS model for simulation of real-time systems at a high abstraction level with SystemC.”. In their implementation, they are particularly concerned with achieving a very accurate RTOS time representation, although the RTOS energy impact is ignored. As an example, when modeling context switches, they distinguish among context load and save durations and scheduling algorithm duration.

Other research groups have also used the SystemC language as the basis for building their own RTOS models. In [5], an extension of the SystemC simulation engine is proposed, where a new simulation library is added to implement the so called T-THREADs, which are Task Thread processes derived from the origi-
nal SystemC SC_THREAD processes. In a similar fashion, in [31] the SystemC SC_THREAD processes are also exploited to create an OS model. The authors rely on timed OS simulation based on delay annotation, where timing information is derived from benchmark data provided by the OS vendor.

As for modeling the OS, high-level simulation-based tools are also used for modeling other system-level aspects, such as MPSoCs and bus contention. Many of them rely on TLM. In [71], the authors choose an approximately-timed transaction-level implementation to evaluate the timing effects due to bus contention when mapping applications to MPSoCs. The model is written in SystemC. In [68], the authors also use TLM to increase communication speed in MPSoC design. However, since they are also interested in speeding up computation, they build abstract processor models for fast software simulation. Such processor models, which are based on SpecC, do not simulate the instruction set architecture; instead, they abstract it away and only reflect the processor and software behavior. In [66], a methodology is proposed for software estimation in heterogeneous multiprocessor systems. Applications are represented as a set of tasks forming a fork-join task graph, while the architecture is specified using the High-level Machine Description (HMDES) language [16].

2.3 Analytical Estimation Tools

The advantage of analytical methods over simulation-based methods is that they do not rely on an executable system model and, in general, their estimation speed is much higher than the speed of their simulation-based counterpart. Nonetheless, analytical approaches often take into account the worst-case scenario and therefore they may be too pessimistic in certain circumstances. For this reason, analytical models are well suited to systems for which it makes sense to assume a deterministic or worst-case behavior, regardless of the input stimuli.

Event stream-based models are an example of an analytical approach. In this case, estimation relies on evaluating the task execution on shared resources for event streams, like periodic events. Network calculus [39] and queuing theory are examples of how to use event-stream models for making estimation. The former has been applied to network processor design [72], [73], [29] and embedded real-time systems in general [15]. The latter has been used, among other things, for doing performance estimation and modeling contention in MPSoCs [6].

2.4 Estimation Tools Based on the Combination of Multiple Approaches

These tools aim at reducing the total estimation time by running the minimum amount of simulations. From there, all the useful information is collected and reused through analytical considerations to make further estimations.
CHAPTER 2. COMMON APPROACHES TO SYSTEM-LEVEL ESTIMATION AND DESIGN TODAY

This approach is common for the evaluation of caches and memory structures. In this case, a single initial simulation generates a trace of all the memory accesses. Since the cache model is known, this trace can then be used to calculate the overall performance and hit/miss figures. The same trace can be reused for different cache structures, which allows us to save time otherwise wasted in simulation. Interesting work using such trace-driven approach for memory subsystems exploration can be found in [74], [20], [27]. The same trace-driven technique has also been applied in [36], [37], [26] to the design of on-chip communication structures: through an initial simulation, a trace recording the inter-module communication history has been generated, which has then be used to make estimations on the best communication configuration for the given design.

A variant of this mixed approach consists in going through an initial characterization process by running a set of simulations. The information extracted in this way is collected and given as an input to analytical models for further estimation purposes. This technique has been applied to caches performance analysis and described in [21], [22].

In [10], a stochastic bus contention model for [MPSoc] is used together with code execution on the host platform. Estimation relies on the alternation of two phases: a first calibration/characterization phase aims at building and training a stochastic model for a sufficiently long time on a cycle-accurate simulation. In a second phase, the stochastic model is used together with code execution on the host platform to detect contention. Since this abstraction level is much higher than the cycle-accurate level, it allows us to increase the granularity used to model each single advance in time of the target and, consequently, also the estimation speed. The length and the number of times these two phases alternate with each other during a use-case estimation is variable. This work has been developed within the context of MESH [52].

2.5 Which Category Does Funtime Belong to?

This section compares the key features of Funtime to significant approaches reviewed so far. This allows us to conclude that Funtime is a combination of simulation-based and stochastic tools.

**Funtime as a simulation-based tool.** Simulation happens in Funtime during the characterization phase, where energy and latency figures are collected from a back-annotated gate-level netlist. This information allows us to create energy and performance models for IPs (see Chapter 4 and 5), for the OS (see Chapter 8), and it allows us to model bus contention (see Chapter 7) or the effect of caches (see Chapter 6). However none of these are executable models, but just look-up tables or mathematical expressions. It is emphasized that the characterization phase is a one-time activity to be carried out by the IP or the OS provider and, as such, gate-level simulation is not part of the actual estimation activity and does not
at all concern the system designer using Funtime. In fact, estimation in Funtime does not involve simulating the architecture, either at gate level or at a higher abstraction level. Instead, it only involves running and profiling an instrumented application on the development host (a common PC). In this last aspect, Funtime differentiates itself from the majority of the simulation-based approaches reviewed in the past sections, especially from those based on SystemC/TLM. In fact, many of these approaches carry out estimation through a comprehensive simulation, which includes simulating architectures and applications.

As far as OS modeling is concerned, Funtime partially resembles the approach presented in [77], since this approach also uses simulation only to pre-characterize the OS, and not during the actual estimation. However, in [77] characterization is done only to factor in the extra latency induced by RTOS and it ignores the impact on energy, which is instead considered in this work. Besides, the authors rely on Instruction-Set Simulation (ISS) to run their applications and measure the task duration, in order to derive the actual number of clock ticks. In [78], Yi et al. extend the work done in [77] by using a trace-driven co-simulation in SystemC. The Funtime methodology, in contrast to both [77] and [78], does not simulate the architecture in SystemC or in ISS. As described in detail in the next chapters, estimation in Funtime consists of a first phase where applications run on the development host and are profiled with target-specific details, and of a second phase where the estimation of the OS contribution, as well as of bus contention and cache effects, comes in the form of a refinement activity which does not involve simulation at all.

**Funtime as a stochastic tool.** The energy and performance models resulting from the characterization activity are in the form of look-up tables or analytical expressions. Funtime uses these models during the estimation activity in order to back-annotate the applications that, let us recall it, are compiled for, and are running on the evaluation host. Because it relies on models that are calibrated on the average case, Funtime can also be considered as a stochastic tool.

As discussed in Section 2.3, the approach adopted in [10] in the context of the MESH framework [52] also estimates bus contention by using stochastic models together with execution of the application on the host. However, Funtime differs mainly in the following two aspects. First, while the authors in [10] alternate multiple times between the training and the execution phases during the estimation of a single application, training is done in Funtime only once on a set of significant reference benchmarks used to calibrate the contention model. This avoids introducing extra delays. The calibrated model for contention prediction is then applied at the end of the code execution. For the details about bus contention prediction in Funtime, please refer to Chapter 7. Second, the approach here proposed also accounts for the bus contention impact on energy consumption. In order to have accurate energy figures, the stochastic model is trained on a gate-level netlist.
Chapter 3

An Overview of the Funtime Estimation Framework

This chapter presents an overview of the Funtime estimation framework. The presentation begins with the Funtime user interface, followed by a detailed introduction to the three layers of the Funtime: the IP Level, the Algorithmic Level and the Refinement Level. This is followed by miscellaneous aspects related to Funtime, with the purpose of providing a more complete and consistent picture of the approach.

3.1 Introduction

As mentioned in the previous chapters, the contribution of this work is to propose a fast and accurate energy and performance estimation framework that operates at the functional untimed (or algorithmic) level of abstraction. Subsection 1.5 and Chapter 2 have both summarized the key features of the Funtime approach as well as compared it to other common approaches, with particular focus on TLM. This chapter describes the Funtime methodology in detail. Figure 3.1 is used as a reference throughout the chapter in order to ease the approach description.

Figure 3.1 is vertically split into three columns: the external left column identifies the Funtime User Interface, the right column identifies the Funtime output in terms of energy and performance prediction, while the central column contains the Funtime core. This is further split horizontally into three layers which, adopting a bottom-up view, are as follows: the IP Level (Level A), the Algorithmic Level (Level B) and the Refinement Level (Level C). In Figure 3.1, annotations in red point to the section/chapter where each level is discussed in detail. In addition, while describing each level from A to C in the next sections, a pedagogical example is built up with the purpose of helping the reader to better understand the way the methodology operates in reality.
3.2 Funtime User Interface

Funtime expects to receive two pieces of information from the user: an Architecture Specification (AS) and a Use Case Specification (UCS). The AS symbolically tells which IPs compose the design, how they are connected to each other and how they are configured. However, this information does not represent or refer to a hardware simulation model, since this would go against the whole idea that Funtime rests on: avoiding any high-level simulation of the hardware. The IPs that can be used in the AS must of course have a corresponding model in the IP models database $\mathcal{A}$ to allow estimation. This constraint is shown in Figure 3.1 by a dashed arrow between the IP models database and the AS.

The UCS instead contains information on how the design described in the AS is meant to be used. Examples can be the following: 1. The applications used for the specific test case; 2. The way such applications have been mapped to the resources specified in the AS; 3. The time offset at which each application gets triggered; 4. The input data; 5. The I/O from which the data is received; 6. The I/O through which the data is sent out and 7. OS-related details, such as scheduling policy, clock tick frequency, possible interrupts, etc. As for the AS case, the applications mentioned in the UCS must be present in the applications database $\mathcal{B}$. This constraint is shown by a dashed arrow between the applications database.
3.3. LEVEL A – THE IP LEVEL

and the UCS.

The AS and UCS can be also seen in this way: while the AS contains the information about the physical connection of the IPs, the UCS contains the information about the logic one.

As of today, both the AS and UCS information are fixed in an ad hoc manner, while a more formal representation, for example in the form of a file, is part of the future work.

3.3 Level A – The IP Level

The IP Level is concerned with characterizing the Intellectual Properties as SoC building blocks. The characterization happens in terms of performance and energy. It is done at gate level and back-annotated with physical design data to enable highly accurate characterization. The availability of a physical layout for each IP also allows preliminary floorplans to be made for different architectures. This in turn enables us to get reasonably good estimates of the global wires length, which also plays a critical role in affecting the overall system energy and performance, especially in today’s SoCs where the delay in wire length dominates the gate delay by a huge margin.

The accuracy obtained at Level A comes at the expense of large amounts of engineering time, which is justified as characterization happens only once and is done by the IP provider, while the end user of the Funtime methodology is not concerned with it. The energy and performance model can come in the form of analytical expression or as a set of look-up tables, depending on the nature of the IP.

3.3.1 Level A1 – Individual IPs

SoC systems are largely composed today in terms of IPs in a buy & assemble model. System Level Design methodology based on such models has evolved to develop industry-wide standards like IP-XACT to enable easy exchange of IPs and promote system-level design and verification in terms of IPs, whose architectural aspects are captured in the XML-based standard IP-XACT. The IP-level energy and performance models in Funtime can be naturally represented in IP-XACT with simple extensions. Funtime in this sense is in line with major industry trends and standards.

The IP-level energy and characterization is done for IP transactions. A transaction in this context is defined as an atomic IP-level function, such as transmit and receive for modems, encode/decode for codecs, send/receive for buses, instructions for a processor, etc. Transactions can also be the coarse-grain routines of an operating system (OS), such as those handling the clock tick, the scheduler, or any synchronization activity in general. In fact, an OS can also be considered as an IP. Transactions have parameters, some design-time and others run-time. The characterization happens for a specific technology and physical design and is in
CHAPTER 3. AN OVERVIEW OF THE FUNTIME ESTIMATION FRAMEWORK

terms of energy in Joules and performance in cycles. This energy and performance characterization of IP-level transactions then forms the input to the Level B and C of the Funtime methodology.

The whole database of IP models in the form of countable set is formally denoted with $\mathcal{A}$, and $a_1 \cdots a_M$ are its elements. Each generic IP model $a_i$ contains its set $\mathcal{T}_a$ of pre-characterized transactions $tac_1 \cdots tac_N$. The letters $E$ and $C$ are used to express the energy and cycles quantities respectively. Keeping in mind that each generic transaction $tac_i$ has an energy and cycles value associated with it, it can be written that $tac_i := tac_i(E,C)$. Chapter 4 presents a detailed example of how an energy and performance model is built for a Leon3 processor.

Figure 3.2 shows a set of IP models that are going to be used in the pedagogical example built throughout these sections. As can be seen, the following IPs will be used: processor (P), memory (M), AES (Advanced Encryption System), Ethernet MAC (ETH), AHB controller (AHB Ctrl), APB controller (APB Ctrl) and AHB/APB bridge. At this stage, no assumption is made on how these IPs are going to be interconnected in a SoC.

![Figure 3.2: Example: database of IP models $\mathcal{A}$](image)

3.3.2 Level A2 – Architecture of IPs

When characterizing transactions of IPs for energy and performance, Funtime distinguishes between two categories of transactions. The first category, called simple transactions, is internal. Such transactions can be characterized by treating the IP in isolation. There are other transactions, whose energy and performance model requires taking into account that the IP being characterized is connected to other IPs through an external communication link. Such transactions are called composite transactions and are modeled in Funtime at Level A2. Chapter 5 details how this is done. With the Leon3 as an example, the simple transactions would be the arithmetic instructions, which act on operands whose value is present in internal registers. In contrast, Leon3 instructions that deal with the external world depend not only on the IP at the other end but also on the interconnect in between, which could be a shared bus based or the more modern NoC-based scheme. It will also depend on the layout. Note that accounting for the impact of the external communication link does not only imply considering what components are present in the link, but also their accessibility, which could be compromised by contention issues. A classic example in this respect is contention in shared-bus architectures, which is discussed in Chapter 7.
3.4 LEVEL B – THE ALGORITHMIC LEVEL

Level A2 also introduces the concept of propagation of transactions. This expression is again related to the fact that, once plugged into a system, individual IPs interact to achieve the system functionality. In general, it is always possible to identify one or more IPs that act as an initiator and one or more IPs that act as a terminator. What is actually initiated/terminated is the propagation of transactions, which starts at the initiator side and ends at the terminator side. For instance, a processor is an initiator IP, while a memory is a terminator IP. At this layer of the Funtime framework, the granularity considered during estimation has the size of an initiator-terminator path. As a continuation of the pedagogical example, Figure 3.3 is proposed to better visualize the initiator-terminator path concept.

![Figure 3.3: Example: architecture composition of IPs](image)

In the figure, five initiator-terminator paths are identified, which cover the whole system functionality. For each path, the green color is used to identify the initiator and the red color to identify the terminator. Some IPs, namely the AES and the ETH, act both as an initiator and terminator. Why this is so is further clarified in the following sections. The physical connectivity, denoted by yellow blobs in Figure 3.3, is defined by the Architecture Specification, whereas the logical connectivity, denoted by the thick blue lines, is defined by the Use Case Specification. When energy and performance estimation is to be carried out by Funtime, the contribution of each of the five paths is taken into consideration, starting from the initiator IP and ending at the terminator IP. In this way, the full system can be evaluated. Note that, at this level, no assumption is yet made about the applications to be run on this system and even less about the way they are going to be mapped to it. In addition, the system schematic shown in Figure 3.3 is not an executable architectural model as TLM requires, but just a symbolic representation of the architecture.

3.4 Level B – The Algorithmic Level

Once each IP transaction has been characterized for energy and performance, the total energy and performance of a system can be calculated, provided that it is
known what applications (algorithmic models) the system is running, how these applications have been mapped to the hardware resources, as well as the total number and type of transactions triggered by such applications inside each IP resource. As opposed to Level A, which is a characterization level and is the responsibility of the IP provider, Level B is therefore Funtime’s main estimation level and is the end user’s entry point for using the Funtime framework. The end user specifies: a) the application, b) the architecture in a symbolic and not executable form, and c) how it is mapped to the architecture in the form of mapping use case. It is emphasized that this information is provided as an input to Level B in the form of UCS. The whole applications database is formally denoted with \( B \), where \( b_1 \cdots b_N \) are its elements.

It is clear that the applications specified in the UCS must be present in the applications database. This is shown in Figure 3.1 by a dashed arrow between the applications database and the UCS.

The same figure also shows that the following step at Level B instruments the selected applications. Instrumentation is a key step in the Funtime flow, since it makes it possible for the applications, which are intrinsically devoid of any architecture information, to become architecture-aware.

Applications can mainly be mapped either to software or hardware and the instrumentation process is different in the two cases. This section outlines both cases, although the main focus of the present work is mapping to software. Mapping to hardware is being pursued in another project that deals with a coarse-grain reconfigurable fabric called DRRA \([43]\).

Once the instrumentation step is completed, the applications are executed to infer and emit a trace of target transactions. Transactions inferred at this level are called primary transactions, as opposed to secondary transactions inferred at the Refinement Level (Level C) above. Since for each IP transaction an energy and performance model is available, provided as an output of the characterization Level A, the total energy and performance (execution time) numbers can be extracted for each application. This is summarized in Equations 3.1 and 3.2 respectively.

\[
E_i = \sum_{j=1}^{J} tac_j(E) \tag{3.1}
\]
\[
ex_i = T \cdot \sum_{j=1}^{J} tac_j(C) \tag{3.2}
\]

In essence, Equation 3.1 says that the total energy \( E_i \) for the i-th application is given by the summation of the energy components associated with each j-th

---

1DRRA stands for Dynamically Reconfigurable Resource Array. This project, which relies on the work of several PhD students, is carried out under the supervision of Prof. Ahmed Hemani at KTH, School of ICT, Electronic Systems Department.

2Note that the execution happens on end user’s machine and not in the target architecture.
3.4. LEVEL B – THE ALGORITHMIC LEVEL

transaction in the transactions trace. The same applies to Equation 3.2 where \( ex_i \) defines the execution time for the i-th application and \( T \) defines the clock period. \( C \) is instead the number of clock cycles associated with a certain transaction, as already defined at the end of Subsection 3.3.1.

It is emphasized that Funtime generates a trace of architectural transactions by simulating functional untimed models instrumented with architectural mapping details, whereas other approaches generate a trace of architectural transactions by actually simulating a model of architecture, usually in the form of an Instruction Set Simulator or a model of the hardware block. This difference is fundamental to Funtime’s higher simulation efficiency. And since the architectural transactions in Funtime are characterized at gate level and annotated with post-layout parasitics, the accuracy of Funtime estimation is also very high.

Next comes a more detailed presentation of how Funtime instruments applications mapped to software (processors).

3.4.1 Instrumentation and trace generation for applications mapped to software

The instrumentation of applications mapped to software resources relies on associating the basic blocks of target assembly instructions with the application source lines, typically written in C. By definition, a basic block is the smallest block of instructions code that is executed sequentially, from beginning to end, without branching. The whole estimation process for applications mapped to software is now presented in detail, and Figure 3.4 is used as a reference for the explanation.

1. An application is compiled for the evaluation host (typically a common x86-based PC) with profiling support enabled. In the context of this work, C is always used as a source code language and the GNU GCC compiler tools chain is used for compilation and profiling purposes. To enable source code profiling, the switch \(-\text{coverage}\) has to be used at the GCC compiler command line. Alternatively, using the switches \(-\text{fprofile-arcs} \) and \(-\text{ftest-coverage}\) together also gives the same result.

2. Mixed source/assembly code files are generated for the same application for the target architecture. Figure 3.4(b) shows what a mixed source/assembly file for a SPARC architecture can look like. Two areas of the file have been highlighted in red color. They correspond to two basic blocks, both related to source code line 21, visible in Figure 3.4(a). The reference source code line is printed on the top of the basic block.

3. The application is executed on the evaluation host (the user’s machine) and a code coverage tool is used to extract the number of times that each source code line has been executed. The profiled source code is also shown in Figure 3.4(a). In this case, the GCOV tool has been used to annotate the source. This file provides the following useful information: the leftmost column tells the
CHAPTER 3. AN OVERVIEW OF THE Funtime ESTIMATION FRAMEWORK

3 - Instrumented source code (C)

4 - Relating source lines to assembly

2 - Mixed source (C)/assembly (SPARC) code

Figure 3.4: Instrumentation of applications mapped to software

3 - Instrumented source code (C)

 CHAPTER 3. AN OVERVIEW OF THE Funtime ESTIMATION FRAMEWORK

3 - Instrumented source code (C)

4 - Relating source lines to assembly

2 - Mixed source (C)/assembly (SPARC) code

Figure 3.4: Instrumentation of applications mapped to software

number of times that the source line has been executed; the "$" symbol indicates that the source line is never executed; the "-" symbol indicates that the line does not contain any source code. Moving to the right, another column is visible, telling the source line number. Note that the same number can appear multiple times, as it is the case for line 21. This depends on the fact that the same source line may have multiple basic blocks associated with it. There are two basic blocks associated with line 21. A classic example is the for(x=0; x<M; x++) loop construct. In this case, if the code is compiled with no optimization, there are typically three basic blocks that are created: one block containing the index x initialization, a second block containing the increment of the index variable x and a third block containing the comparison between the index x value and its up-bound limit M.

4. By associating the information about the number of times that each source code line has been executed with the instructions and basic blocks of the same application for the target processor, it is possible to generate a report of primary transactions. This association process is shown in the figure by two arrows going from the two lines 21 in the profiled source code to the corresponding basic blocks in the mixed source/assembly file. Once the executed primary transactions are known, each of them can be associated with its own model of energy and number of cycles. By summing up these values for all the transactions, the total application energy and execution time can be found.
3.4. LEVEL B – THE ALGORITHMIC LEVEL

Figure 3.4 also shows two examples of output report files. Figure 3.4(c) contains the total count of transactions – instructions in this case – inferred for a certain application and also a more fine-grain per-transaction count. Figure 3.4(d) contains instead an example of inferred transactions trace. It is pointed out that these are primary transactions, which are then fed to the Refinement Level, which uses them to infer the secondary transactions.

3.4.1.1 Advantages and challenges of the estimation approach for applications mapped to software

The estimation approach detailed above presents some advantages:

1. It is very general, since it does not depend on a specific architecture. This leads to high re-usability and requires no extra engineering effort: one just needs a compiler for the evaluation host, a cross-compiler for the target architecture, a code coverage tool and a relatively simple script that automatically performs the source-to-assembly mapping operation and that is application and architecture independent as well.

2. This way of generating a transactions execution trace, i.e. by inferring it from source code execution, is much faster than extracting it from simulation of the architectural model. As already discussed in the previous chapters, the speed advantage comes from the fact that no time has to be invested in creating and simulating a hardware executable model.

This approach also presents some challenges though, which are discussed below:

1. Profiling done with GCOV only allows us by default to count the number of times each source code line is executed, but it does not keep track of their execution order. As a consequence, the knowledge of the execution order of the corresponding assembly instructions is also limited to each single basic block, which contains sequential code by definition. Knowing the complete instructions execution order is however critical to allow accurate energy estimation. In fact, while there are IPs whose transactions execution order has no impact on the energy consumption, there are other IPs where the order considerably affects the energy. This is especially true in the case of pipelined IPs, such as processors. The knowledge of the complete transactions order also allows us to know the time when an application will reach a certain point in its execution. This information is very useful when trying to predict the inter-application synchronization aspects, to establish for instance which application will lock a shared resource first. This aspect is discussed in more depth in Chapter 8, where details on how to model the operating system impact on energy and performance are presented. Later in this section, a description follows on enabling the inference of a full execution order trace by enhancing the GNU GCC compiler.
2. Instrumentation relies first of all on the availability of source code that can be profiled and of mixed source/assembly code for the target. However, when the application under test contains calls to library routines (sin, cos, printf, etc.), no source file is included corresponding to those routines, since their code has already been pre-compiled into the library – GlibC is the standard C library. As a consequence, there is no source code to profile or from which to create a mixed source/assembly file. The contribution made by these routines would therefore be neglected by Funtime during estimation. While neglecting this contribution may be harmless if the execution code of these routines is a small percentage of the total, it could instead lead to a large estimation error if they have a high contribution. A workaround to this problem can be compiling directly the routines source code files with the profiling switch enabled instead of using the library. This would be feasible in the case of the GlibC routines, since their code is open-source. A real solution, and also the only viable way for closed-source routines code, should, however, be looked for at the library level. The approach adopted in this work is to avoid the problem by implementing our own routines instead of using the library ones and by commenting out all the print-related routines.

3.4.1.2 Enhancing the GNU GCC compiler

This section describes the implementation of how the GNU GCC \(^3\) compiler was enhanced to enable it to generate a trace of architectural transactions in Funtime. However, the overall principle is general and compiler-independent.

A typical compiler is composed of three main components: a front-end, a middle-end and a back-end. The front-end is responsible for parsing the high-level language (HLL) source code files and generating a language-independent intermediate representation (IR) in the form of a control-flow graph. The output of the front-end is the input for the middle-end, which lowers the abstraction level down by applying a set of transformations and optimizations. The number of steps performed at this stage is highly dependent on the level of optimization requested and, in general, on the parameters passed at the compiler command line. Finally, the back-end has to translate the intermediate representation received from the middle-end into a machine-description (MD) file.

GNU GCC can also be described according to the above execution flow, as shown in Figure 3.5. The middle-end is itself composed of three different intermediate representations: GENERIC, which is a language-independent tree representation; GIMPLE, which is a reduced GENERIC subset used during optimization; RTL (Register Transfer Language), where the instructions to be output are defined in an algebraic form describing what the instruction does \(^{45}[3][75]\).

GNU GCC comes with some built-in profiling capabilities. In particular, the command-line switch \texttt{--coverage} adds a counter at the end of each basic block, so

\(^3\)Version 4.4.1
as to enable a final total execution count of each source code line. Although this information provides a good code-coverage overview, it only allows us to infer the total number each target instruction is executed, but not the exact execution order. To overcome this limitation, the GCC profiling capabilities have to be extended. In detail, at the GENERIC intermediate level, at the end of each basic block belonging to the application control-flow graph, a call is inserted to a function that records the current file name, source line number and basic block number. This is symbolically shown in red in Figure 3.5. This allows the generation of the exact source code execution trace when the application is run on the host and, indirectly, it also enables the inference of the full execution trace for the target transactions. Figure 3.6 shows an example of the source execution trace inferred.

<table>
<thead>
<tr>
<th>File name</th>
<th>Line</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_2.c</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>file_3.c</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>file_3.c</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>file_2.c</td>
<td>13</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 3.6: Example of source execution trace

### 3.4.2 Estimation of applications mapped to hardware

While processors can implement any arbitrary functionality, when the performance and/or computational efficiency demand is high, hardware macros are used. These macros implement only a limited and specific set of functionalities. As a consequence, the estimation process of applications mapped to hardware macros is also different.

First of all, the hybrid nature that usually characterizes hardware macros is pointed out: on one hand, they actually act as pure hardware blocks and implement the functionality for which they have been designed. On the other hand instead, they require some control directives to initialize and drive them. Such directives are usually implemented by software drivers running on one or more processors
CHAPTER 3. AN OVERVIEW OF THE FUNTIME ESTIMATION FRAMEWORK

connected to the same SoC. The consequence is that hardware macros act both as an initiator and as a terminator. This is also the reason why the AES and the ETH blocks in Figures 3.3 to 3.8 are highlighted both in green and red.

This aspect has also to be considered when estimating the overall energy and performance for such hardware macros. In particular, in order to include the contribution of the software driver in the model, the method explained above for applications mapped to software could in principle be applied. However, the possibility to do this is subordinated to the following assumption, which is unlikely to be true. This assumption is that the source code is available for the driver. This is usually not the case, since the driver may be covered by copyright and thus the system architect would not have access to its source. Besides, assuming that the source code is available, software drivers are by definition a very special piece of code, implemented to write/read to/from specific hardware. As such, it would be very difficult to be able to run such software on the evaluation host. Finally, drivers are usually required to be as optimized as possible and direct assembly coding is therefore preferred to the general C coding.

Since an IP and its own software driver are usually implemented by the same IP provider, it makes sense that the IP provider not only implements IP energy and performance models, but that it also characterizes the software drivers for its IPs. This characterization can be done at the granularity level of a single driver routine. Since this same idea is applied to the operating systems characterization, the reader may want to see Section 8.2 for further details.

Estimating energy and performance for the actual functional part of the hardware macro can instead be enabled as follows. Recalling that estimation in Funtime relies first of all on the ability to infer the execution trace of target transactions without having to implement or simulate the target architecture, two different ways can be taken.

- If the application to be mapped to hardware shows quite a simple functionality, the trace of transactions can be inferred through an intuitive approach. For instance, for an AES or ETH, transactions correspond to encrypt/decrypt or transmit/receive operations respectively. It is also observed that the number, type and order of transactions only depend in these cases on the total volume and the chunk of data consumed by each encrypt/decrypt or transmit/receive transaction. These details are known to the system architect and can be passed to Funtime through the UCS and AS respectively. The reason is that the first detail is a run-time configuration parameter, while the second is an application-related parameter. For these two macros, a simple mathematical expression can thus be derived as a function of the parameters listed above and the trace of transactions can be inferred. Also note that the estimation speed in this case is not proportional to the amount of data and to the size of each data chunk, as it would be instead if a simulation-based estimation was run, but is constant. This is a critical advantage.

- If the hardware macro exhibits a more complex functionality, then another
3.4. LEVEL B – THE ALGORITHMIC LEVEL

approach has to be followed to infer the trace of transactions. For example, what happens if the number, type and order of transactions depend on the actual data being processed? This would be the case of an image or video encoder/decoder. While the steps to take in this case have not been investigated within the present work, this is being done as part of the DRRA Project. For further details, please refer to [43].

The pedagogical example introduced at the beginning of this chapter has been enhanced in Figure 3.7 to present Funtime Level B.

Compared to Figure 3.3 shown in Level A2, this new figure also shows the applications that have to run on the architecture defined in Level A2, how they have been mapped to the hardware resources and in what order they are executed. This information is specified in the UCS by the system architect. In this example, Funtime has to carry out estimation for a streaming video application, where the video stream is received by the Ethernet MAC, decrypted by the AES and finally decoded using the H.264 codec. There are three applications in total. If the purple and light blue colors are used to indicate the applications mapped to software and hardware respectively, it is straightforward that the H.264 decoder is to be implemented in software and mapped to processor P2, while the Ethernet and AES applications are mapped to their respective hardware IPs. However, these last two applications also have a second arrow indicating a mapping to processor P1 and P3 respectively. This is to highlight the extra software driver component for these applications. At this point, all the necessary information is available to infer the transactions trace induced in every initiator IP and propagated to the respective terminator IP. The transactions trace inference follows the steps detailed in Subsections 3.4.1 and 3.4.2. In particular, if a closer look is taken at path 1 - path 5 following their actual order, as detailed in the UCS, the following can be observed:
CHAPTER 3. AN OVERVIEW OF THE FUNTIME ESTIMATION FRAMEWORK

- **path 1**: this path concerns the software driver component related to the ETH hardware macro. P1 is the initiator, while ETH is the terminator. Transactions propagation involves an action by the AHB Ctrl, the AHB/APB bridge and the APB Ctrl.

- **path 5**: this path concerns the software driver component related to the AES hardware macro. P3 is the initiator, while AES is the terminator. Transactions propagation involves an action by the AHB Ctrl, the AHB/APB bridge and the APB Ctrl.

- **path 2**: this path concerns the actual functionality of the ETH hardware macro. ETH is the initiator, while the memory M is the terminator. Transactions propagation involves an action by the AHB Ctrl.

- **path 4**: this path concerns the actual functionality of the AES hardware macro. AES is the initiator, while the memory M is the terminator. Transactions propagation involves an action by the AHB Ctrl.

- **path 3**: this path concerns the JPEG application mapped to software. P2 is the initiator, while the memory M is the terminator. Transactions propagation involves an action by the AHB Ctrl.

Note that, in the case of the initiators P1, P2, P3, the propagated transactions can be inferred by knowing the number of store/load instructions; instead, in the case of the initiators ETH and AES, they can be inferred by knowing the volume of data and the size of each chunk of data.

Observe that applications at this level are considered individually and as if they had all the resources available to themselves. Consequently, the ideal scenario is envisaged at this level where no concurrency occurs, no bus contention either, and where there are no caches. This trace of primary transactions is then refined at Level C to factor the induced secondary transactions like arbitration and scheduling.

### 3.5 Level C – The Refinement Level

Once each IP transaction has been characterized for energy and performance (see Section 3.3 - Level A), and the trace of primary transactions generated by each IP as a result of the application mapped to it has been inferred (see Section 3.4 - Level B), the total energy and performance of the system can finally be estimated as well. However, in most cases this estimation would lead to inaccurate results, both for energy and performance, with respect to the actual values collected from a gate-level extraction. This is why, as shown in Figure 3.1, no output is read from Level B. Instead, it is sent directly upwards as an input of the Refinement Level. As will be discussed in more detail when describing the different refinements, the quantity \( \text{ex}\_i \) (execution time of the \( i^{th} \) application) estimated at Level B plays a central role in leading the refinement process.
The reason why refinements are needed is that there are some implications of sharing resources resulting from mapping, which are ignored at Level B and which need to be taken into account as well. Such implications are mainly related to the fact that, in reality, applications do not have all the resources always available to themselves and often they even have to interact with other applications. The most visible effect of these non-idealities is a variation of the total application execution length, in terms of number of cycles and, consequently, of the final energy too. In traditional cycle-accurate architectural simulations, this would not be an issue, as the exact number of cycles would come along as part of the simulation itself. In contrast, in the approach proposed in this thesis, this contribution needs to be factored in separately. In particular, it is demonstrated that this can be done by refining the trace of primary transactions collected at the Level B below. Such a refinement produces what has been called secondary transactions. Accounting for the joint contribution of both primary and secondary transactions enables accurate estimation of the total system energy and overall performance. This is shown as an output of Level C in Figure 3.1. The refinements implemented in this work are as follows.

- **Transactions interdependency** (Chapter 6): energy and cycles associated with a transaction often not only depend on the properties of the transaction itself, but also on the actual sequence in which transactions are issued. For example, in a processor case, its instructions execution order plays an important role in affecting the total application energy and cycles. Transactions interdependency is modeled as stall conditions, which increase the ideal execution time.

- **Caches** (Chapter 6): the goal is quantifying and modeling the variation of an application length and energy in a system with cache, given the cache miss ratio and compared to the case with no cache. As opposed to the effect of transactions interdependency, caches reduce the total number of application cycles, since there are fewer stall conditions caused by communication with memory. It is pointed out that Funtime does not estimate cache miss or hit rates. This is given as input to the Funtime methodology to gauge the impact on total execution time and energy consumption.

- **Bus contention** (Chapter 7): in a real MPSoC, all the processors are typically connected to the same bus and contention occurs when more than one application, mapped to a certain processor, requests an exclusive access to a shared resource. The main consequence of contention is an overall degradation in performance and an increase in energy consumption.

- **Operating System** (Chapter 8): although they are critical in allowing multi-tasking and in managing the sharing of hardware resources, operating systems have an overhead, due to the execution of extra software code, both in terms of latency and energy increase.
Understanding in what order the different refinements should be applied is also important and is discussed in Section 3.9.

Although the approach followed in handling each refinement is different in the details, they all share a basic idea: characterization is done to quantify, in terms of energy and cycles, the impact of the factor to be refined; the characterization results are then translated into an analytical expression or an algorithm for later use during the estimation phase.

As for the case of the IP energy and performance models, the characterization activity for the Refinement Level is time-consuming, since it relies on gate-level simulation for accurate energy and timing extraction. However, it is justified since it is a one-time effort, which should be carried out by the IP provider. Note that the operating system is also considered as an IP. Applying the refinements themselves, during the actual estimation phase by the end user, is in contrast quite fast. More details about this are presented in Section 3.8.

The pedagogical example in Figure 3.8 has been extended to represent the Level C of Funtime. Compared to Figure 3.7, the new figure points out two non-idealities, whose impact has to be accounted for at the Refinement Level. First, the presence of an operating system can be observed, which controls the scheduling of all the applications mapped to software, as well as the presence of the software drivers for the ETH and AES hardware macros. This is visualized by arrows going through

![Figure 3.8: Example: Real scenario including RTOS, concurrent execution and competition for shared resources (bus)](image)
the OS circle. Second, applications run concurrently, thus having to deal with bus contention issues. In particular, the applications relative to path 1 and 5 are triggered first, then all the others are triggered together. Also note that in this case there are three extra paths caused by the presence of applications \( b_3, b_{76} \) and \( b_{120} \).

### 3.6 From Single Transactions to a Final Trace

This section wants to give the reader a visual example of the way a final trace of primary and secondary transactions can be inferred starting from single, unrelated transactions. As a starting point, a 2D representation of a single transaction \( tac_i \) is proposed in Figure 3.9. The horizontal axis shows the execution time \( ex \), whereas the vertical axis shows the power \( P \). The transaction energy is therefore \( E = P \cdot ex \).

![Figure 3.9: Visual representation of a single transaction](image)

Figure 3.10 shows instead how, starting from single transactions \( tac_i \), Funtime performs the inference of a full trace of primary and secondary transactions. This figure maintains the Funtime layered structure as well. Starting from Level A at the bottom, it is assumed that \( a_{23} \) identifies a processor IP energy and performance model. Such a model consists of a set \( T_a \) of pre-characterized transactions \( tac_i(E, C) \) with different execution time and power consumption. At Level B, the three applications \( b_1, b_2 \) and \( b_3 \) have been mapped to the processor IP for which there is model \( a_{23} \). By following the steps described in Section 3.4.1, Funtime infers a trace of primary transactions for each of the three applications. The total processor execution time estimated by Funtime is thus given by summing up the time assigned to each application. The total energy is instead given by summing up each single transaction energy. Level C takes the trace of primary transactions and injects secondary transactions according to the rules of each refinement. This figure shows three kinds of secondary transactions: stalls introduced by the transactions interdependency refinement in red; stalls introduced by the bus contention refinement in green; extra instructions introduced by the OS refinement in light blue. The figure shows how the OS introduces context switches between the three applications. At this point, the total energy and execution time can be calculated.

### 3.7 Time Representation in Funtime

When presenting the related work and especially Transaction-Level Modeling in Chapter 2, some space has been dedicated to discussing the time representation
CHAPTER 3. AN OVERVIEW OF THE Funtime ESTIMATION FRAMEWORK

Figure 3.10: From single transactions to a full trace of primary and secondary transactions in TLM. This section focuses instead on the way Funtime deals with time and compares it to TLM. Figure 3.11 is used as a reference to discuss the comparison. The graph displayed on the left side represents the TLM case, while the one on the right represents the Funtime.

Figure 3.11: Comparison between the time representation in TLM and Funtime

Two kinds of time are considered: first, the time required by the tool to complete its estimation, which is indicated with \( ex_{\text{tlm}} \) and \( ex_{\text{funtime}} \) in the TLM and Funtime case respectively; second, the execution time of the system under test, which is the result of the estimation. This time is indicated as \( ex_{\text{sut}} \).

In the figure, \( ex_{\text{tlm}} \) and \( ex_{\text{funtime}} \) are plotted on the horizontal axis, while \( ex_{\text{sut}} \)
3.7. TIME REPRESENTATION IN FUNTIME

is plotted on the vertical axis. In addition, both graphs show several time windows \( w_i \) which correspond to different phases of the estimation tool execution time \( exe_{tlm} \) or \( exe_{funtime} \). Note also that, due to practical drawing constraints, the graphs are not scaled to each other as they would be in reality, and the same applies to the width of each \( w_i \). In reality, from the experimental results that have been carried out, the \( \frac{exe_{tlm}}{exe_{funtime}} \) ratio has been found to be on average around 30.

The graph related to TLM is analyzed first. Here, two time windows are identified. Window \( w_1 \) corresponds to the compilation time of the application that has to run on the executable model of the target. This window only implies an increment in \( exe_{tlm} \), but no increment in \( exe_{sut} \). Window \( w_2 \) is instead representative of the simulation, which also corresponds to the actual estimation process. The length of this window is proportional to the duration of the application to be simulated. In this case, both \( exe_{tlm} \) and \( exe_{sut} \) become incremented. However, \( exe_{sut} \) does not grow linearly, but as a step function. The height of each step \( y \) depends on the time granularity chosen for the TLM model. In the case of a cycle-accurate implementation, the simulation advance in time has the granularity of a clock cycle. Conversely, in the case of an approximately-timed implementation, which is more likely for a TLM model, the advance in time has a much higher granularity. The width of each step \( x \) depends instead on the complexity of the system being simulated and can vary for each advance of time. The higher the complexity, the longer the time \( exe_{tlm} \) to simulate a single advance in time.

Considering now the graph related to Funtime on the right, four windows are identified. Window \( w_1 \) corresponds to the compilation time of the application that has to run on the evaluation host. Window \( w_2 \) is representative of the actual time spent to execute the application on the evaluation host. As for the TLM case, the width of this window is proportional to the duration of the application. However, for the estimation of the same system, the width of this window in Funtime case is much smaller than in the TLM case, since running an application directly on the host machine is much faster than simulating a system that runs that application. Window \( w_3 \) corresponds to the time spent by Funtime to process the profiling results and build a trace of primary transactions. In essence, windows \( w_1, w_2 \) and \( w_3 \) identify themselves well with the Level B of Funtime. At the end of \( w_3 \), the first estimate on the execution time (and energy) of the application being estimated is produced. Window \( w_4 \) corresponds instead to the time spent for refinement and thus to Level C. At the end of this window, the updated trace of primary and secondary transactions and the final execution time \( exe_{sut} \) for the system under test are produced. The \( exe_{funtime} \) spent for refinement in window \( w_4 \) has a minor impact on the total value of \( exe_{funtime} \). The reason is that, as seen in Section 3.5, refinements are a static activity which relies either on applying an analytical function or on iterating through an algorithm loop. The number of iterations is, however, not proportional to the application length, but to other parameters, different in each refinement, which allows the number of iterations to be kept low. For more details, the reader is referred to Chapters 7 and 8, where such algorithms are presented in
the case of the bus contention and OS refinement respectively.

3.8 Engineering Effort in Funtime

The goal of this section is to quantify the engineering effort required by the Funtime approach. For this purpose, the Funtime approach is considered as consisting of two well-defined phases: a characterization phase and an estimation phase.

The characterization phase is time-consuming and thus it requires a large engineering effort. This phase includes for example implementing IP energy and performance models (at Level A). As summarized in Section 3.3, the creation of IP models implies characterizing each IP transaction for energy and number of cycles. This phase also includes implementing the refinements (at Level C). As summarized in Section 3.5, refinements consist of a characterization phase as well, where the impact of a certain element (transactions interdependency, caches, bus contention and OS) is quantified and translated into an analytical expression or an algorithm for later use during the estimation phase. None of these activities is carried out by the end user of Funtime though. In fact, the engineering effort of this characterization phase is shared among the different IP providers when creating IP energy and performance models or implementing the refinements, including the OS refinement, which can also be considered as an IP. Although time-consuming, this characterization phase is justified for the following three reasons: first, it is done only once; second, it gives high accuracy to the whole Funtime framework since it is carried out at the gate level; third, it is a shared activity, which helps to bring the engineering effort down.

As opposed to the characterization phase, the estimation phase is very fast and interactive, and the engineering effort minimal. In fact, this phase involves Level B and C of Funtime. As seen before, Level B produces a trace of primary transactions after running an instrumented application on the evaluation host; Level C adds the contribution of the secondary transactions by applying the refinement implemented during the characterization phase. What happens in the evaluation phase is the responsibility of the system architect.

Based on these considerations, the overall engineering effort required by the Funtime approach is low, and in general lower than the effort required by TLM. The reason is that TLM also relies on the availability of IP energy models if the system architect expects to get energy numbers out of the TLM simulation. However, in addition to this, TLM also requires the time-consuming activity of using a HLL to build executable models of the target architecture. This step, which is absent in Funtime, is not negligible within the overall design time budget.

3.9 Refinements Order and Interaction

In Section 3.5, the importance of the refinement activity has been justified and an overview of the refinements implemented in the context of the present work has
been presented. So far, refinements have been presented individually. In practice, however, the refinement order is significant and impacts the estimation accuracy. The following discussion presents this aspect of the refinement process in Funtime. The same MPSoC example presented throughout Sections 3.3 to 3.5 is used here as a reference architecture.

The initial assumption is made that the reference MPSoC has a single application mapped on itself and running on one processor. It is also assumed that executing this application only implies write/read operations to/from a memory connected to the MPSoC shared bus. In this case, no bus contention is expected, since this processor is the only active initiator. In addition, there is no need to assume any running Operating System either, which means that the OS overhead can be neglected. The energy and execution time of this unique application can thus be estimated. However, as mentioned in Section 3.5, in doing this it is not possible just to sum up the energy $E$ and cycles $C$ associated with each individual transaction $tac_i(E, C)$: it is also necessary to account for the extra cycles (stall events) caused by the different order in which transactions – processor instructions in this case – can be triggered.

The transactions interdependency refinement, which is discussed in Chapter 6, is thus the first refinement to be applied. Its action is symbolically shown in Figure 3.12 and consists of the synthetic injection of extra cycles (in red) after each transaction.

![Figure 3.12: First refinement: transaction interdependency](image)

When modeling processor transactions and when applying the transactions interdependency refinement, it makes sense to assume either the case where the cache miss ratio $MR = 100\%$ or $MR = 0\%$. Besides, both cases can in principle be representative of real-case scenarios. The case of $MR = 100\%$ can indicate an architecture where no cache is used, which is common in systems with hard real-time constraints, since they have to be as predictable as possible. Conversely, the case of $MR \approx 0\%$, especially when looking at the instruction cache, can occur when running very regular applications, such as algorithms that often iterate a large number of times through the same code loop.

The present work, as detailed in Chapter 4, shows how to characterize a processor assuming both the case of $MR = 100\%$ and $MR = 0\%$. Together with the transactions interdependency refinement, the cache refinement allows us to get the transactions trace shown on the right side of Figure 3.12. However, in reality, there are also many cases where intermediate $MR$ values are possible. Accounting for this possibility is part of the cache refinement, which is discussed in Chapter 6 and
is thus the second refinement to be applied.

In more detail, the cache refinement aims at quantifying and modeling the variation of an application length and energy in a system with cache, given the cache miss ratio. In principle, this variation can be predicted with respect to the case of $MR = 100\%$ or $MR = 0\%$. The present work uses the case of $MR = 100\%$ as a reference. In addition, the variation prediction is done so far with respect to an entire application, and not to its single transactions. The reason is that, to make this second option possible, it would be necessary to simulate the application on the target, in order to have the details on which transactions actually cause a cache hit or miss. However, this is not a viable approach, since it would go against the whole idea Funtime rests on.

Note that the transactions interdependency refinement and the cache refinement, i.e. the first and second refinement to be applied, act on individual applications and do not depend on other refinements. In fact, even if in principle the cache miss ratio would be affected by the effects of bus contention and operating system overhead, in Funtime the miss ratio is passed as an input parameter to the UCS by the system designer and the cache refinement is applied as a function of this value.

Considering now that the MPSoC can run multiple applications in parallel, where more than one application can be mapped to the same processor, and where an OS is used to control their execution, the bus contention refinement and the operating system refinement also start playing an important role. This is the case represented in Figure 3.8. The bus contention refinement aims at predicting the extra cycles (stall events) and energy introduced by bus contention based on the number of initiators sharing the same bus and on the distinction between store/load-related instructions vs. other instructions. The OS refinement instead aims at predicting the extra cycles and energy introduced by the presence of an OS.

Unlike the two previous refinements, these refinements have the following dependencies. First, they depend on the previous refinements. Intuitively, in fact, the probability of bus contention events or the number of context switches is likely to increase along with the application’s execution length, where the estimated application length is a function of the transactions interdependency and cache refinements. Second, the OS and bus contention refinements also depend on each other. In fact, the OS itself introduces extra instructions, which increase the number of memory accesses and thus the probability of bus contention. In turn, a higher bus contention increases the overall applications time, thus leading to more context switches/clock tick interrupts and, in general, to a higher OS overhead. This again will affect the bus contention probabilities, and so on.

For these reasons, the order in which the bus contention refinement and the OS refinement have to be applied cannot be sequential as for the previous refinements. Instead, an iterative process has to be adopted.
Chapter 4

Level A1 – Implementing IP Energy and Performance Models: The Leon3 Processor Case

This chapter presents a methodology to create the energy and performance model for an IP corresponding to the Level A1 in the Funtime framework. This activity is time consuming, but it is justified as being a one-time activity that is used by multiple end-users and is intended to be performed by the IP provider. The methodology is applied to the Leon3 as a case study. It is also shown that an accuracy can be achieved within 15% of gate-level measurement. The work presented in this chapter is based on the author’s publication in [53].

4.1 Introduction

As discussed in Section 3.3.1, creating IP energy and performance models makes sense since, although time-consuming, it is done only once by the IP provider. This is possible thanks to the spreading of the Design&Reuse concept, which emphasizes that reusing already existing IPs is more efficient than writing new ones from scratch. This modeling activity relies on characterizing each IP transaction in terms of energy in Joules and performance in cycles, where each transaction corresponds to one IP functionality. Examples of transactions can be write/read for a bus, code/decode for a codec, transmit/receive for an I/O, the full instructions set for a processor, or the coarse-grain routines of an operating system.

The activity of building individual IP energy and performance models is part of the Level A1 of the Funtime methodology. Such models are then collected in a database and used by the above Level A2. This is shown in Figure 4.1.

In this chapter, the creation of an energy and performance model for the Leon3 processor is used as a case study. The Leon3 model proposed in the present work is based on instruction-level characterization. Although this modeling approach
for processors has already been adopted in the past, as reviewed in Section 4.2, a further contribution is made in the present work through the following:

- instructions interdependency: the energy of an instruction is heavily dependent on the actual sequence in which instructions have been issued. For this reason, two different energy models are implemented and compared: one where characterization is done on a single instruction basis, the other where characterization is done on pairs of instructions.

- data switching activity: it is shown that instructions energy varies significantly depending on the switching activity of the data that the instruction handles. As a consequence, a sensible way is proposed to model energy accounting for this aspect also.

- registers correlation: it is shown that, for a subset of instructions pairs, when two subsequent instructions use at least one same register, the execution time increases and, consequently, the energy consumption changes too.

The model is validated on a set of real applications by comparing its estimates to the numbers extracted from gate-level simulation. An error is obtained within 6% and 12% for the models applied to Leon3 configurations without and with cache respectively.

### 4.2 Related Work

Building processor energy models has been the focus of several research groups. In [51], the authors describe a method to create instruction-level power models based on directly measuring the average current from the board. In order to make this measurement more accurate, they suggest executing a sequence where the same instruction under test (IUT) is repeated several times in a row, and taking the average value.

In [70], the authors characterize instructions for energy by executing a predefined sequence of instructions, where the IUT is always interleaved by a certain number of NOPs. Besides, measurements are in this case carried out from back-annotated gate-level simulation. The approach presented in this thesis, as detailed in Section 4.4, is similar, but enhanced to consider pairs of instructions, in order to model instructions interdependency on energy.
In [23], Beltrame et al. describe a work aimed at also accounting for inter-instruction effects arising from a pipelined execution of the code. The estimates derived from this work, which concentrates on timing aspects, are obtained by means of a dynamic analysis carried out on a significant benchmark set; for this purpose, the execution trace of the chosen benchmarks is used. In a similar fashion, the work presented in this thesis also relies on the execution trace (at gate level) to generate the models; however, besides timing figures, predefined energy values are associated directly with each instructions pair. Furthermore, an analysis is proposed of the impact that both switching activity in the data and register correlation have on the actual instruction-set energy.

In [12] and [23], the authors present a method to account for instruction energy consumption and inter-instruction dependency’s impact on timing. They use a significant benchmark set and use the trace from this benchmark to build their timing model that accounts for inter-instruction dependency. The present thesis goes beyond this work by building a model also accounting for the dependency of data switching activity and registers correlation on energy.

4.3 Configuration and Synthesis

Leon3 is an open-source SPARC-compliant processor designed by AEROFLEX GAISLER [1] and provided as part of GRLIB\textsuperscript{1} in the form of a VHDL description. The Leon3 configuration that was selected has a multiplier/divider unit. Two versions of this base configuration have been characterized, one with cache controller and one without. Synthesis and simulation were performed using Cadence Encounter(R) RTL Compiler\textsuperscript{2} and Cadence Incisive\textsuperscript{3} respectively. The technology library used is TSMC 90nm and the target frequency 400MHz. Table 4.1 reports area figures for the two configurations chosen.

<table>
<thead>
<tr>
<th></th>
<th>No Cache Ctrl</th>
<th>With Cache Ctrl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area [(\mu m^2)]</td>
<td>75 315</td>
<td>77 291</td>
</tr>
</tbody>
</table>

4.4 Energy Characterization

This section describes the method used to build the Leon3 energy model, which considers both single instructions and pairs of instructions. As shown later on in the chapter, the model considering pairs of instructions is more accurate. Thanks to a highly-automated set of scripts, the full model generation on a PC with a Xeon

---

\textsuperscript{1}Version 1.0.16-b2314 has been used in this work.
\textsuperscript{2}Version 07.10-s021_1 has been used in this work.
\textsuperscript{3}Version 06.20-s004 has been used in this work.
processor @2.5GHz takes five hours for the 1-instruction-based case and around 10 days for the 2-instruction-based case.

### 4.4.1 1-instruction-based model

The main idea is to isolate and extract energy and number of cycles associated with each processor instruction for the selected configurations. Since no Floating-Point Unit (FPU) or Co-Processor (CP) are provided within the present work, the corresponding instructions are not part of the model.

The method used for characterization relies on executing for 100 consecutive times a basic sequence composed of five NOPs, one instruction under test (IUT) and five NOPs, for a total of 1100 executed instructions, as shown in Expression 4.1.

\[
100 \times (5 \cdot NOP, IUT, 5 \cdot NOP)
\]

The reason behind choosing such a sequence as a reference is to be able to isolate each IUT, and this operation is easier if the IUT is surrounded by NOP instructions, which in principle do not have any active functionality, do not show interdependency and consume the least energy. In effect, in this way each IUT is isolated by 10 NOPs. However, since the pipeline is made of seven stages, seven NOPs would probably have been enough.

The basic sequence is then repeated 100 times. The reason is to find a reasonable average energy value that can be considered statistically valid. The higher the number of iterations, the better the result from a statistical perspective, but the overhead introduced by the increased simulation time also needs to be considered. That is why the number of iterations is limited to 100.

In addition, in each iteration, different registers are used for the same IUT; this is to keep the correlation between subsequent executions as low as possible. However, this does not apply to instructions that do not operate on any register, like NOP. Correlation between instruction energy and registers usage is dealt with in Section 4.6.

The first selected IUT is the NOP itself. From Expression 4.1, this leads to the execution of 1100 NOPs, for which energy is calculated. The energy and cycles for a single NOP are therefore given by Eq. 4.2 and Eq. 4.3 respectively.

\[
E_{NOP} = \frac{E_{1100\_NOP}}{1100}
\]

\[
C_{NOP} = \frac{t_{stop} - t_{start}}{1100 \cdot T_{clk}}
\]

In Eq. 4.3, \(t_{start}\) and \(t_{stop}\) correspond to the initial and final gate-level simulation time for the 100-time loop of the basic sequence.

Once the energy of a single NOP is known, calculating the energy for the other IUTs is quite straightforward: since in these cases 100 IUTs and 1000 NOPs are
executed, the energy of 100 IUTs is given by Eq. (4.4)

\[ E_{100\_IUT} = E_{1000\_NOP+100\_IUT} - 1000 \cdot E_{NOP} \] (4.4)

and therefore the energy for a single IUT is given by Eq. (4.5)

\[ E_{IUT} = \frac{E_{100\_IUT}}{100} \] (4.5)

Similarly, the cycles for each IUT are given by Eq. (4.6)

\[ C_{IUT} = \frac{t_{stop} - t_{start}}{T_{clk}} - 1000 \cdot C_{NOP} \] (4.6)

As a final step, the energy values for the whole instructions set are collected in a look-up table (LUT) together with the number of cycles. Note that reporting energy rather than power has the advantage of making the model independent of the frequency at which the IP runs, and therefore also more general.

Table 4.2 shows a fragment of the LUT generated for Leon3 using the two different configurations mentioned in Section 4.3. In detail, the second column shows energy and cycles figures for the configuration without cache controller and, therefore, without cache. Both the third and fourth column refer instead to the case with cache controller and cache, showing what happens in the case of cache miss and hit respectively.

As expected, the highest values for energy and cycles correspond to the case when a cache miss occurs. The difference in cycles between No cache and Cache miss, as they both effectively refer to a miss condition, is more intricate. The reason for the difference is that, when a cache controller is instantiated, a check is always performed to see if the instruction is present in cache or not. If it is not, this operation introduces an overhead that is absent in the case of No cache. The lowest values for energy and cycles are instead obtained in the case of cache hit, as reported in column 4. These values correspond to those that can be found in the Gaisler IP Cores Manual and that represent the ideal execution case. Finally, in order to maintain consistency among the different models, the same data set has been used.

Table 4.2: Leon3 LUT for energy and performance

| IUT | No cache | | | Cache miss | | | Cache hit | |
|-----|----------|---|---|------------|---|---|-------------|
| add | 100.71   | 1.25|   | 109.47     | 1.37|   | 92.75      | 1.00|
| and | 97.59    | 1.25|   | 106.33     | 1.37|   | 89.64      | 1.00|
| jmp | 237.72   | 5.26|   | 276.46     | 7.24|   | 125.59     | 3.00|
| nop | 34.73    | 1.25|   | 42.16      | 1.37|   | 25.34      | 1.00|
| or  | 108.90   | 1.25|   | 115.89     | 1.37|   | 99.29      | 1.00|
| st  | 235.58   | 5.12|   | 200.47     | 5.40|   | 108.72     | 2.00|
4.4.2 2-instruction-based model

The method used for characterization is in principle similar to that presented in Subsection 4.4.1, with the difference that now a sequence of two IUTs is considered instead of one. The basic sequence is shown in Expression 4.7.

\[
100 \times (5 \cdot \text{NOP}, IUT_1, IUT_2, 5 \cdot \text{NOP}) \tag{4.7}
\]

Table 4.3 reports a few examples where the energy is compared between the 1-instruction-based and the 2-instruction-based model for the case of cache hit. Similarly, Figure 4.2 shows the overall energy difference for all the possible 2-tuples of instructions, which means around 6000 pairs. In 90% of the cases, the energy of a 2-tuple is lower than the energy given by summing single instructions energy, and the average difference is around -12%. The methodology proposed in the present work is, however, general and based on the actual measurement of increment and decrement in energy and is thus not dependent on a specific architecture property. No table is shown comparing the number of cycles extracted in the two approaches, since they are identical.

Table 4.3: Energy models: 1-instr. vs. 2-instr.

<table>
<thead>
<tr>
<th>IUT</th>
<th>1-instr-model</th>
<th>2-instr-model</th>
<th>Diff.[%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>92.75</td>
<td>182.39</td>
<td>-33.85</td>
</tr>
<tr>
<td>and</td>
<td>89.64</td>
<td>188.93</td>
<td>-32.40</td>
</tr>
<tr>
<td>or</td>
<td>99.29</td>
<td>127.72</td>
<td>-32.40</td>
</tr>
<tr>
<td>ld</td>
<td>75.17</td>
<td>168.30</td>
<td>-23.81</td>
</tr>
<tr>
<td>xor</td>
<td>93.13</td>
<td>128.24</td>
<td>-23.81</td>
</tr>
</tbody>
</table>

Figure 4.2: \( E_{IUT_1} + E_{IUT_2} \) versus \( E_{IUT_1-IUT_2} \)
This section presents a study that shows how the energy consumed by the same instruction can vary depending on the level of switching activity in the data handled by the instruction between subsequent executions. The \textit{add} instruction is taken as IUT, which executes the operation \( r e g_d = r e g_{s1} + r e g_{s2} \), and run 32 times the full sequence reported in Expression 4.1, increasing every time of 1 unit the maximum possible number of switching bits in both the operands of the IUT.

A more precise idea of the method used for the analysis is shown in Table 4.4, where ‘s’ and ‘c’ indicate a possibly switching and a constant bit respectively. For this experiment, the value \( c=0 \) was chosen.

<table>
<thead>
<tr>
<th>#</th>
<th>\text{reg_s1}[31 \text{ downto } 0] &amp; \text{reg_s2}[31 \text{ downto } 0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ccccccccccccccccccccccccccccccccccccccc s</td>
</tr>
<tr>
<td>1</td>
<td>cccccccccccccccccccccccccccccccccccccss</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30</td>
<td>c sssssssssssssssssssssssssssssssssssssssssssssss</td>
</tr>
<tr>
<td>31</td>
<td>sssssssssssssssssssssssssssssssssssssssssssssss</td>
</tr>
</tbody>
</table>
4.6 Registers Correlation

This section presents an analysis to evaluate the impact of registers correlation between two instructions in sequence on the energy consumption of an instructions pair. The modeling sequence reported in Expression 4.7 is used. Specifically, for registers correlation the case is meant where the destination register for the first instruction becomes a source register for the second instruction. An example is given below:

$$\text{add \ reg1 + reg2 -> reg3}$$
$$\text{sub \ reg3 + reg4 -> reg5}$$

Table 4.5 compares the time and energy variations caused by registers correlation to the case where correlation is not present. This table also refers to the case of cache hit. The table is further split horizontally into two groups: the upwards portion refers to the case where correlation increases the number of cycles. This increase can be justified by using the ld_add pair as an example: since one of the two operands on which the add has to operate is not available until the ld is completed, no advantage can be taken by the pipelined architecture of the processor. As a consequence, two extra cycles are required for the add to complete.

In principle, this reasoning also applies to the downwards portion of the table. Here however, registers correlation does not affect the execution time. This happens because the first instruction in the pair takes only one cycle to execute, which is too short a time to allow any level of parallelization, even in the case that no correlation between registers exists.

The real occurrence of registers correlation has been estimated on the same test applications used in Section 4.5. The results show that such a correlation is present on average for 50% of the executed instructions, despite a relatively high standard deviation $\sigma \simeq 13\%$. It is therefore sensible to believe that also factoring in this aspect when building the energy model would increase the estimation accuracy. However, since this work is not yet complete, the present thesis validates a Leon3 model that does not yet account for registers correlation.
4.7 Validating the Leon3 Model Estimation Accuracy

Validation for the Leon3 model has been carried out on a set of common benchmark applications and takes gate-level extraction as a reference. The accuracy of the Leon3 modeling approach has been tested for a processor with and without cache (4k I/D cache). When applying the models, the estimated number of total cycles is always lower than the actual one extracted from gate-level simulation. This difference has been associated with the occurrence of stall conditions, whose energy per cycle $E_{\text{stall}}$ has been modeled according to Eq. 4.8.
Table 4.7: Leon3 energy models (1-instr-based with/without data dependency) vs. gate-level energy measurement (4kB I/D-cache)

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td># Instructions</td>
<td>1 371 584</td>
<td>4 008 651</td>
<td>1 585 313</td>
</tr>
<tr>
<td># Cycles</td>
<td>2 001 308</td>
<td>6 499 470</td>
<td>2 751 001</td>
</tr>
<tr>
<td>Real energy [mJ]</td>
<td>0.0936</td>
<td>0.3427</td>
<td>0.1318</td>
</tr>
</tbody>
</table>

with data dependency (toggle = 7 bits)

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.1072</td>
<td>0.2886</td>
<td>0.1119</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.0162</td>
<td>0.0644</td>
<td>0.0322</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.1234</td>
<td>0.3530</td>
<td>0.1441</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+31.92</td>
<td>+3.00</td>
<td>+9.36</td>
</tr>
</tbody>
</table>

without data dependency (toggle = 15 bits)

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.1276</td>
<td>0.3174</td>
<td>0.1201</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.0162</td>
<td>0.0644</td>
<td>0.0322</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.1438</td>
<td>0.3818</td>
<td>0.1523</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+53.66</td>
<td>+11.41</td>
<td>+15.58</td>
</tr>
</tbody>
</table>

Viterbi  Queens  Dhrystone  MD5

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td># Instructions</td>
<td>2 452 903</td>
<td>4 266 783</td>
<td>1 950 813</td>
</tr>
<tr>
<td># Cycles</td>
<td>4 101 483</td>
<td>5 867 291</td>
<td>3 988 119</td>
</tr>
<tr>
<td>Real energy [mJ]</td>
<td>0.2057</td>
<td>0.2644</td>
<td>0.1842</td>
</tr>
</tbody>
</table>

with data dependency (toggle = 7 bits)

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.1840</td>
<td>0.3045</td>
<td>0.1483</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.0593</td>
<td>0.0505</td>
<td>0.0598</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.2433</td>
<td>0.3551</td>
<td>0.2081</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+18.28</td>
<td>+34.28</td>
<td>+12.99</td>
</tr>
</tbody>
</table>

with data dependency (toggle = 15 bits)

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.2098</td>
<td>0.3608</td>
<td>0.1633</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.0593</td>
<td>0.0505</td>
<td>0.0598</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.2691</td>
<td>0.4114</td>
<td>0.2231</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+30.82</td>
<td>+55.57</td>
<td>+21.12</td>
</tr>
</tbody>
</table>

\[
\frac{E_{\text{stall}}}{cycle} = \frac{E_{IUT\_miss} - E_{IUT\_hit}}{C_{IUT\_miss} - C_{IUT\_hit}} \quad (4.8)
\]

where, for a certain instruction, the numerator shows the energy difference between a cache miss and hit case, while the denominator shows the cycles difference. As a result, it is obtained that \( \frac{E_{\text{stall}}}{cycle} \approx 32.83 \text{pJ} \). The two separate contributions given by the ideal model and the stalls are also shown. The total estimated energy results from the summation of these two components.

Table 4.6 and 4.7 compare the accuracy of the Leon3 1-instruction-based models, both when data switching activity (data dependency) is factored in and when it is not. In the former case, the models have been built according to the considerations reported in Section 4.5, while in the latter case an arbitrary value of 15 toggling bits in the data has been chosen to generate the models. Table 4.6 refers to the processor without cache, while Table 4.7 refers to the processor with a 4kB I/D cache. Note that both the Leon3 models always overestimate the energy compared to the golden values extracted from gate-level simulation. However, the models not accounting for data switching activity perform even worse, with an error that is
4.7. VALIDATING THE LEON3 MODEL ESTIMATION ACCURACY

on average 3.68% and 12.53% higher that the corresponding models accounting for data switching activity, for the configuration without and with cache respectively. Furthermore, observe that the component of the total estimated energy coming from the contribution of the *stalls* is identical in both models, while it is only the other component associated with the estimated instruction energy that varies. This is explained by the fact that, although a *stall* can be caused by multiple reasons, the energy associated with it is constant. The results shown in these two tables are also consistent with what is reported in [14]: the author states that, in order to compensate the overestimation given by an energy model built without accounting for data switching activity (all the data bits were assumed to toggle between subsequent executions), a correction factor had to be introduced. The reader should also remember that, for the sake of simplicity, the Leon3 models have been built to represent a unique, yet realistic, average data switching activity, as discussed in Section 4.5. It would clearly be possible to increase the accuracy of the models by further refining them, to represent the different data switching activities associated with any application. However, this would also imply the need of reading and processing the applications data every time, which can represent a considerable overhead.

Table 4.8: Leon3 energy models (1-instr-based/2-instr-based with data dependency) vs. gate-level energy measurement (no cache)

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td># Instructions</td>
<td>1 371 584</td>
<td>4 008 651</td>
<td>1 585 313</td>
</tr>
<tr>
<td># Cycles</td>
<td>6 485 094</td>
<td>31 649 232</td>
<td>13 902 019</td>
</tr>
<tr>
<td>Real energy [mJ]</td>
<td>0.2396</td>
<td>1.1508</td>
<td>0.4906</td>
</tr>
</tbody>
</table>

### 1-instr.-based model

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.1388</td>
<td>0.5209</td>
<td>0.1917</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.1319</td>
<td>0.6766</td>
<td>0.3218</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.2706</td>
<td>1.1975</td>
<td>0.5355</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+12.96</td>
<td>+4.05</td>
<td>+4.67</td>
</tr>
</tbody>
</table>

### 2-instr.-based model

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.1187</td>
<td>0.4116</td>
<td>0.1504</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.1319</td>
<td>0.6766</td>
<td>0.3218</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.2506</td>
<td>1.0882</td>
<td>0.4722</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+4.59</td>
<td>-5.45</td>
<td>-3.75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Viterbi</th>
<th>Queens</th>
<th>Dhrystone</th>
<th>MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td># Instructions</td>
<td>2 452 903</td>
<td>4 266 783</td>
<td>1 950 813</td>
<td>1 802 380</td>
</tr>
<tr>
<td># Cycles</td>
<td>17 511 989</td>
<td>24 193 885</td>
<td>15 938 144</td>
<td>14 428 588</td>
</tr>
<tr>
<td>Real energy [mJ]</td>
<td>0.6250</td>
<td>0.8571</td>
<td>0.5641</td>
<td>0.5228</td>
</tr>
</tbody>
</table>

### 1-instr.-based model

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.3009</td>
<td>0.4389</td>
<td>0.2581</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.3871</td>
<td>0.5166</td>
<td>0.3529</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.6880</td>
<td>0.9555</td>
<td>0.6110</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+10.08</td>
<td>+11.48</td>
<td>+8.31</td>
</tr>
</tbody>
</table>

### 2-instr.-based model

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.2458</td>
<td>0.3622</td>
<td>0.2060</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.3871</td>
<td>0.5166</td>
<td>0.3529</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.6329</td>
<td>0.8788</td>
<td>0.5589</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+1.25</td>
<td>+2.53</td>
<td>-0.92</td>
</tr>
</tbody>
</table>

Table 4.8 and 4.9 compare the accuracy of the Leon3 1-instruction-based and 2-instruction-based models, both factoring in data switching activity, for the configu-
CHAPTER 4. LEVEL A1 – IMPLEMENTING IP ENERGY AND PERFORMANCE MODELS: THE LEON3 PROCESSOR CASE

ration without and with cache respectively. It can be noticed that the 2-instruction-based model is in general more accurate than the 1-instruction-based one, with a margin of error <6% and <12% compared to gate level, for the configuration without and with cache respectively. Besides, the model based on a single instruction always overestimates the actual energy, making a worst-case error up to \( \approx 32\% \) when considering the configuration with cache; compared to the 2-instruction-based model, it also overestimates on average \( \approx 8.65\% \) and \( \approx 17.45\% \) for the configuration without and with cache respectively, which is consistent with what has been reported in Subsection 4.4.2.

Table 4.9: Leon3 energy models (1-instr-based/2-instr-based with data dependency) vs. gate-level energy measurement (4kB I/D-cache)

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td># Instructions</td>
<td>1 371 584</td>
<td>4 008 651</td>
<td>1 585 313</td>
</tr>
<tr>
<td># Cycles</td>
<td>2 001 308</td>
<td>6 499 470</td>
<td>2 751 001</td>
</tr>
<tr>
<td>Real energy [mJ]</td>
<td>0.0936</td>
<td>0.3427</td>
<td>0.1318</td>
</tr>
</tbody>
</table>

**1-instr.-based model**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.1072</td>
<td>0.2886</td>
<td>0.1119</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.0162</td>
<td>0.0644</td>
<td>0.0322</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.1234</td>
<td>0.3530</td>
<td>0.1441</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+31.92</td>
<td>+3.00</td>
<td>+9.36</td>
</tr>
</tbody>
</table>

**2-instr.-based model**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.0843</td>
<td>0.2380</td>
<td>0.0985</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.0162</td>
<td>0.0644</td>
<td>0.0322</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.1005</td>
<td>0.3024</td>
<td>0.1307</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+7.37</td>
<td>-11.75</td>
<td>-0.82</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Viterbi</th>
<th>Queens</th>
<th>Dhrystone</th>
<th>MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td># Instructions</td>
<td>2 452 903</td>
<td>4 266 783</td>
<td>1 950 813</td>
<td>1 802 380</td>
</tr>
<tr>
<td># Cycles</td>
<td>4 410 483</td>
<td>5 867 291</td>
<td>3 988 119</td>
<td>3 734 118</td>
</tr>
<tr>
<td>Real energy [mJ]</td>
<td>0.2057</td>
<td>0.2644</td>
<td>0.1842</td>
<td>0.1787</td>
</tr>
</tbody>
</table>

**1-instr.-based model**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.1840</td>
<td>0.3045</td>
<td>0.1483</td>
<td>0.1292</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.0593</td>
<td>0.0505</td>
<td>0.0598</td>
<td>0.0592</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.2433</td>
<td>0.3551</td>
<td>0.2081</td>
<td>0.1884</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+18.28</td>
<td>+34.28</td>
<td>+12.99</td>
<td>+5.47</td>
</tr>
</tbody>
</table>

**2-instr.-based model**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated instructions energy [mJ]</td>
<td>0.1472</td>
<td>0.2333</td>
<td>0.1224</td>
<td>0.1045</td>
</tr>
<tr>
<td>Estimated stalls energy [mJ]</td>
<td>0.0593</td>
<td>0.0505</td>
<td>0.0598</td>
<td>0.0592</td>
</tr>
<tr>
<td>Estimated total energy [mJ]</td>
<td>0.2065</td>
<td>0.2839</td>
<td>0.1822</td>
<td>0.1638</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+0.42</td>
<td>+7.36</td>
<td>-1.10</td>
<td>-8.33</td>
</tr>
</tbody>
</table>

From these results, it is possible to draw two important conclusions: first, accounting for data switching activity helps improve the overall model accuracy; second, the 2-instruction-based model is reasonably accurate and, on average, significantly more accurate than the 1-instruction-based model, which justifies the extra effort in making it.

It would also be interesting to compare the results presented in this thesis with someone else’s, but at present no equivalent work has been found to compare with.
4.8 Summary

This chapter has described a methodology to implement an energy and performance model for a processor, a Leon3 in this case, based on instruction-level characterization. Specifically, the impact of instructions interdependency on energy has been investigated by building an energy model with two degrees of accuracy, one based on characterizing each single instruction, the other based on characterizing instructions pairs. The study also takes into account the effect that data switching activity and registers correlation have on energy. Validation has then been done against gate level for estimation accuracy, incurring an error within 6% and 12% for the models applied to the configurations without and with cache respectively.

Note that, thanks to the high level of automation applied to the energy models generation process, this approach is quite general and can therefore be easily adapted and used on different architectures. More important, this modeling exercise has the goal of showing how an IP energy and performance model can be implemented in practice.

Another example of how an IP energy and performance model can be built for a NoC switch is presented in the previous work published by the author in [50]. In this case, due to the different nature of the IP, the model has been expressed using an analytical representation instead of LUTs.
Chapter 5

Level A2 – Modeling IPs
Composite Transactions

This chapter shows how it is possible to model composite transactions. This activity is necessary in order to be able to identify and account for the components representing the external communication aspects. These components are architecture-dependent and change when the system architecture changes. The activity of modeling composite transactions is illustrated in this chapter by making a switch from an AHB bus-based SoC to a NoC-based SoC. The model, when adapted for NoC, has an accuracy within 10% compared to the gate-level measurement.

5.1 Introduction

Subsection 3.3.2 has discussed the distinction in Funtime between simple and composite transactions. While simple transactions are internal and can be characterized by treating the IP in isolation, composite transactions consist, as the name suggests, of both an internal and external component, which accounts for the impact of external communication. The advantage of being able to model composite transactions is in the possibility to generalize an IP model and to reuse it within architectures based on different communication infrastructures.

Modeling composite transactions is part of the Level A2 of the Funtime methodology, which receives in input the individual IP models from the Level A1 below and then sends the extended models, accounting for composite transactions, to the Level B and C above. This is shown in Figure 5.1.

With the Leon3 processor in mind, examples of simple transactions are the arithmetic instructions that act on operands whose value is present in internal registers. In contrast, composite transactions are store/load-related instructions that deal with the external world and depend not only on the Leon3 itself, but also on the IP at the other end and on the interconnect in between, which could be a shared bus-based or a NoC-based scheme.
This chapter illustrates the activity of modeling composite transactions by making a switch from an AHB bus-based SoC to a NoC-based SoC. The Leon3 modeled in Chapter 4 is again taken as the reference IP. After discussing the method, the resulting IP model is also validated for energy estimation accuracy. The results show it to be within 10% of gate-level estimation.

5.2 Bus-based and NoC-based SoC: Architectural Similarities and Differences

This section has the purpose of pointing out the similarities and differences between the original bus-based SoC where the model for the Leon3 processor was first created and a multi-core NoC-based SoC, which from now on will be called McNoC. To ease the task, Figure 5.2 and 5.3 are used as reference.

Figure 5.2 shows the reference SoC, where a Leon3 processor communicates with an SRAM memory, both bound to the same AMBA AHB bus. Figure 5.3 shows instead McNoC, which is a multi-processor NoC-based SoC using the same IP building blocks as for the SoC in Figure 5.2. McNoC is a network of Processor - Memory nodes (PM). This is highlighted in green dashed color in Figure 5.3. The different PMs communicate with each other using the network infrastructure, which is mainly composed of interconnects and routers.
5.3. CHARACTERIZING COMPOSITE TRANSACTIONS

Besides being based on NoC, the SoC in Figure 5.3 involves a block called Data Management Engine (DME) [17], which has been highlighted in red dotted color in Figure 5.3 with the purpose of supporting distributed shared memory, cache coherence and memory consistency across McNoC.

As a consequence of the point above, the memory architecture is also different in each single PM compared to the original SoC. In particular, each local memory is split into two sections: a private section, accessed through a physical addressing scheme, and a shared section, accessed through a virtual addressing scheme. The private section corresponds to the SRAM memory of the original SoC and can be accessed only by the resources in the local PM node. This is also the memory containing all the application instructions. The shared section is instead accessible both by the local and any remote PM node. While instructions can by construction only be stored in the private memory section, data can be stored in both private and shared memory section, depending on the accessibility level that they are required to have.

Note that the considerations above, especially the last one about how instructions and data are stored into memory, are at the basis of the choices made to model composite transactions when porting the Leon3 model from the original SoC to McNoC.

5.3 Characterizing Composite Transactions

In order to characterize composite transactions for the Leon3 processor and implement a general model out of it, different scenarios are considered, where the internal transaction component is kept constant, while the external one - the one related to the external communication - is made variable. Four cases are considered, as shown in Figure 5.4. The main difference among all these cases is in the location of data, which each time are moved to a different SRAM memory. This is how a
variation in the communication pattern is achieved.

Figure 5.4: Different cases considered for characterization of composite transactions

The instructions location is indicated using a red dotted rectangle and the data location using a green dashed rectangle. Sub-figure 5.4(a) shows the original Leon3-based SoC. In this case, both instructions and data are located in the same memory, i.e. the SRAM. The remaining three sub-figures show instead McNoC nodes. Sub-
figure 5.4(b) shows an McNoC node where both instructions and data are in the local private memory. Note that this condition corresponds to the original SoC, with the only difference that here the DME is also introduced. Sub-figure 5.4(c) shows an McNoC node where instructions are in the local private memory and data are in the local shared memory. The reader is reminded that, by construction, instructions can only be located in local private memory, while data can be in both private and shared memory [17]. In the former case, data are only accessible by the resources in the same node; in the latter case, they can also be accessed by resources in other nodes. Finally, sub-figure 5.4(d) shows an McNoC node where instructions are in the local private memory and data are in the remote shared memory of the adjacent node to the right. This means that the number of nodes $N$ involved in the communication is 2. Although not shown in the figure, the cases where $N$ is equal to 3, 4 and 5 are also characterized. Note that $N$ represents the actual hop count between source and destination node, and not their relative (x,y) distance. This also means that contention and any non-ideality are accounted for in $N$.

Table 5.1 summarizes the most significant results from characterizing composite transactions for the Leon3 processor. Examples of simple transactions are also reported in the table to show their independence from the external communication and architectural variation. Since it has proved to be more accurate in Chapter 4, the characterization process carried out in this chapter relies on the 2-instruction-based model for the Leon3.

The table only shows the results related to a Leon3 configuration without cache. However, characterization has also been carried out for a Leon3 with a 32kB Instruction cache and an 8kB Data cache. Although the numbers are different from those shown in this table, the following considerations still also hold for the Leon3 configured with cache.

The table, which has been drawn on two levels due to lack of space, is split horizontally into three sub-tables. Sub-table 5.1A contains the results for a subset of instructions that only exhibit an internal component (simple transactions); Sub-table 5.1B and 5.1C contain instead results for all the load/store-related instructions respectively (composite transactions). Each sub-table reports energy in pJ and number of cycles for any of the cases described above. The following conclusions can be drawn.

First, energy and cycles for any of the cases in Sub-table 5.1A are identical. This is due to the fact that none of these instructions handles external data communication and thus they are not affected by the data location. Second, all the energy and cycle numbers extracted for the original SoC (Figure 5.4(a)) and the McNoC in the case of data in the local private memory (Figure 5.4(b)) are also identical, in any of the three sub-tables. This fact proves that the DME component does not introduce any extra delay and that these two cases are equivalent. Third, energy and cycles start increasing in Sub-tables 5.1B and 5.1C for the case where data is in the local shared memory. This is due to the fact that the DME has to first perform a virtual-to-physical address translation. This operation takes
Table 5.1: Summary of the characterization results for simple and composite transactions in Leon3 (no cache)

<table>
<thead>
<tr>
<th>E=energy</th>
<th>Original SoC</th>
<th>Local private</th>
<th>Local shared</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(A) Simple transactions (non data-related instructions)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>96</td>
<td>2.50</td>
<td>96</td>
</tr>
<tr>
<td>sethi</td>
<td>132</td>
<td>2.50</td>
<td>132</td>
</tr>
<tr>
<td>and</td>
<td>124</td>
<td>2.50</td>
<td>124</td>
</tr>
<tr>
<td>andcc</td>
<td>124</td>
<td>2.50</td>
<td>124</td>
</tr>
<tr>
<td>andn</td>
<td>128</td>
<td>2.50</td>
<td>128</td>
</tr>
<tr>
<td>andncc</td>
<td>130</td>
<td>2.50</td>
<td>130</td>
</tr>
<tr>
<td>or</td>
<td>129</td>
<td>2.50</td>
<td>129</td>
</tr>
<tr>
<td>orcc</td>
<td>131</td>
<td>2.50</td>
<td>131</td>
</tr>
</tbody>
</table>

| (B) Composite transactions (Load data-related instructions) | | | | | |
| load      | | | | | | |
| ldsb      | 372          | 11.22         | 372          | 11.22         | 894          | 31.23         |
| ldsh      | 372          | 11.22         | 372          | 11.22         | 894          | 31.23         |
| ldub      | 369          | 11.22         | 369          | 11.22         | 893          | 31.23         |
| lduh      | 368          | 11.22         | 368          | 11.22         | 893          | 31.23         |
| ld        | 368          | 11.22         | 368          | 11.22         | 892          | 31.23         |
| ldd       | 359          | 11.22         | 359          | 11.22         | 1417         | 51.33         |

| (C) Composite transactions (Store data-related instructions) | | | | | |
| store     | | | | | | |
| stb       | 123          | 2.62          | 123          | 2.62          | 650          | 22.63         |
| sth       | 123          | 2.62          | 123          | 2.62          | 649          | 22.63         |
| st        | 118          | 2.62          | 118          | 2.62          | 644          | 22.63         |
| std       | 437          | 12.52         | 437          | 12.52         | 1497         | 52.53         |

<table>
<thead>
<tr>
<th>E=energy</th>
<th>Remote shared (2)</th>
<th>Remote shared (3)</th>
<th>Remote shared (4)</th>
<th>Remote shared (5)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(A) Simple transactions (Non data-related instructions)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>96</td>
<td>2.50</td>
<td>96</td>
<td>2.50</td>
</tr>
<tr>
<td>sethi</td>
<td>132</td>
<td>2.50</td>
<td>132</td>
<td>2.50</td>
</tr>
<tr>
<td>and</td>
<td>124</td>
<td>2.50</td>
<td>124</td>
<td>2.50</td>
</tr>
<tr>
<td>andcc</td>
<td>124</td>
<td>2.50</td>
<td>124</td>
<td>2.50</td>
</tr>
<tr>
<td>andn</td>
<td>128</td>
<td>2.50</td>
<td>128</td>
<td>2.50</td>
</tr>
<tr>
<td>andncc</td>
<td>130</td>
<td>2.50</td>
<td>130</td>
<td>2.50</td>
</tr>
<tr>
<td>or</td>
<td>129</td>
<td>2.50</td>
<td>129</td>
<td>2.50</td>
</tr>
<tr>
<td>orcc</td>
<td>131</td>
<td>2.50</td>
<td>131</td>
<td>2.50</td>
</tr>
</tbody>
</table>

| (B) Composite transactions (Load data-related instructions) | | | | | |
| load      | | | | | | |
| ldsb      | 926              | 38.29             | 1137             | 40.29             | 1191             | 42.29             | 1243             | 44.29             |
| ldsh      | 926              | 38.29             | 1137             | 40.29             | 1191             | 42.29             | 1243             | 44.29             |
| ldub      | 925              | 38.29             | 1136             | 40.29             | 1190             | 42.29             | 1242             | 44.29             |
| lduh      | 925              | 38.29             | 1136             | 40.29             | 1190             | 42.29             | 1242             | 44.29             |
| ld        | 925              | 38.29             | 1135             | 40.29             | 1189             | 42.29             | 1241             | 44.29             |
| ldd       | 1478             | 65.36             | 1896             | 69.36             | 2004             | 73.36             | 2109             | 77.36             |

| (C) Composite transactions (Store data-related instructions) | | | | | |
| store     | | | | | | |
| stb       | 681              | 30.64             | 918              | 32.64             | 972              | 34.64             | 1024             | 36.64             |
| sth       | 680              | 30.64             | 917              | 32.64             | 972              | 34.64             | 1023             | 36.64             |
| st        | 675              | 30.64             | 912              | 32.64             | 967              | 34.64             | 1019             | 36.64             |
| std       | 1564             | 68.49             | 2030             | 72.49             | 2137             | 76.49             | 2243             | 80.49             |
5.3. CHARACTERIZING COMPOSITE TRANSACTIONS

exactly 20 cycles. In fact, all the load/store instructions handling data in the local shared memory take 20 cycles more than the cases where data is in the local private memory. The only exception is represented by the double-load (ldd) and double-store (std) instructions, which take 40 (20 x 2) extra cycles. Fourth, if data is located in a remote shared memory, a further increase in cycles is measured for the load/store instructions. In particular, when communication occurs between two adjacent nodes, 7 extra cycles for load instructions (14 in the case of ldd) and 8 extra cycles for store instructions (16 in the case of std) are added compared to the case of data in local shared memory. Given that each router is buffer-less and that it takes by construction one clock cycle to send a packet from input to output, 4 of these extra cycles are due to the path through two routers back and forth. The remaining cycles are instead due to the DME activity in the remote node. Note also that, for any extra router included in the path, two extra cycles are added. This is visible in the table for paths composed of 3, 4 and 5 nodes.

The extended Leon3 energy and performance model has to be able to express the impact of composite transactions as the superposition of the transaction internal and external contribution. This is shown in Equations 5.1 and 5.2, which refer to number of cycles and energy associated with the jth composite transaction.

\[
C_j = C_{j,\text{int}} + C_{j,\text{ext}} \quad \text{(5.1)}
\]

\[
E_j = E_{j,\text{int}} + E_{j,\text{ext}} \quad \text{(5.2)}
\]

While the internal contribution - \(C_{j,\text{int}}\) or \(E_{j,\text{int}}\) - is constant as depending on the intrinsic IP properties, the external contribution - \(C_{j,\text{ext}}\) or \(E_{j,\text{ext}}\) - is variable and a function of the number and type of components in the communication link from initiator to terminator. Note that this information can be extracted from the AS and UCS, which specify the physical and logical connectivity of the architecture, as discussed in Section 3.2. In addition, the delay introduced by each component in the communication link is known, since it is part of the energy and performance model for that specific component, available from Level A1.

Going back now to the Leon3 case study and bearing in mind the considerations above, it can be observed that the components in the communication link between initiator (Leon3) and terminator (SRAM) can vary as follows: in the original SoC, there is no intermediate component. In the case of McNoC instead, there is the DME and a number of network switches equal to the number of hops the data packet has to make across the network.

Based on the characterization results in Table 5.1, the plots in Figure 5.5 summarize both the cycles and energy variation for the ld instruction in the six different cases considered for McNoC. The plots also point out the quantities associated with internal \((C_{j,\text{int}}\) or \(E_{j,\text{int}}\)) and external \((C_{j,\text{ext}}\) or \(E_{j,\text{ext}}\)) contributions. Results for other composite transactions have not been reported since they present equivalent behavior. It is also emphasized that the plots in Figure 5.5 are representative of applying Equations 5.1 and 5.2 to the McNoC case. Using other external communication links would produce different trends for \(C_{\text{comp},j}\) and \(E_{\text{comp},j}\).
Validation of the extended Leon3 model for composite transactions has been carried out on McNoC for all the cases discussed above in the characterization section. In addition, the cases for path length $N = 7$ and $N = 10$ have also been validated to confirm the viability of the approach proposed in this thesis. Validation is performed against gate-level energy/timing measurement.

Table 5.2 summarizes the results. The table, which has been drawn on two levels due to the lack of space, is split horizontally into four different sections, corresponding to the following four configurations and test applications: reading the table top-down, an application performing 1000 multiplications and a Leon3 without cache; an application performing 1000 multiplications and a Leon3 with cache (I:32kB, D:8kB); an application performing an FFT and a Leon3 without cache; an application performing an FFT and a Leon3 with cache (I:32kB, D:8kB).

Note that, for this validation, the actual applications chosen are not so important, since the main goal is to verify the model ability to capture the impact of the external communication component when varying the architectural configuration, and not the absolute accuracy for a specific configuration. Besides, validation based on more and larger applications like audio/video encoding would not be feasible, due to the very long time necessary to perform gate-level simulation and extraction.

To keep the table compact, no accuracy figures are shown about the prediction of the number of cycles. However, since energy is proportional to the number of cycles, good accuracy in the energy estimation also reflects good accuracy in cycles estimation.

From this table, it can be observed that the generalized model shows a very good degree of accuracy, with an error within 10% of the gate-level measurement.
5.4. VALIDATING THE ACCURACY OF THE EXTENDED MODEL FOR LEON3

Table 5.2: Leon3 model validation versus gate-level energy measurement

<table>
<thead>
<tr>
<th></th>
<th>Private memory</th>
<th>Local shared memory</th>
<th>Path through 2 nodes</th>
<th>Path through 3 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td># Instructions</td>
<td>11513</td>
<td>11513</td>
<td>11513</td>
<td>11513</td>
</tr>
<tr>
<td>1000 multiplications – no cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measured energy (µJ)</td>
<td>3.6738</td>
<td>5.2750</td>
<td>5.8626</td>
<td>6.0232</td>
</tr>
<tr>
<td>Estimated energy (µJ)</td>
<td>3.6058</td>
<td>5.1948</td>
<td>5.2864</td>
<td>5.9438</td>
</tr>
<tr>
<td>Error (%)</td>
<td>-1.85</td>
<td>-1.52</td>
<td>-9.83</td>
<td>-1.32</td>
</tr>
<tr>
<td>1000 multiplications - cache (I:32kB, D:8kB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Instructions</td>
<td>11513</td>
<td>11513</td>
<td>11513</td>
<td>11513</td>
</tr>
<tr>
<td>Measured energy (µJ)</td>
<td>0.9664</td>
<td>1.3435</td>
<td>1.5619</td>
<td>1.6162</td>
</tr>
<tr>
<td>Estimated energy (µJ)</td>
<td>1.0253</td>
<td>1.4064</td>
<td>1.5675</td>
<td>1.7114</td>
</tr>
<tr>
<td>Error (%)</td>
<td>6.10</td>
<td>4.68</td>
<td>0.36</td>
<td>5.89</td>
</tr>
<tr>
<td>FFT - no cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Instructions</td>
<td>683485</td>
<td>683485</td>
<td>683485</td>
<td>683485</td>
</tr>
<tr>
<td>Measured energy (mJ)</td>
<td>0.1232</td>
<td>0.1237</td>
<td>0.1238</td>
<td>0.1239</td>
</tr>
<tr>
<td>Estimated energy (mJ)</td>
<td>0.1267</td>
<td>0.1272</td>
<td>0.1272</td>
<td>0.1273</td>
</tr>
<tr>
<td>Error (%)</td>
<td>2.83</td>
<td>2.83</td>
<td>2.72</td>
<td>2.82</td>
</tr>
<tr>
<td>FFT - cache (I:32kB, D:8kB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Instructions</td>
<td>683485</td>
<td>683485</td>
<td>683485</td>
<td>683485</td>
</tr>
<tr>
<td>Measured energy (mJ)</td>
<td>0.0431</td>
<td>0.0434</td>
<td>0.0435</td>
<td>0.0436</td>
</tr>
<tr>
<td>Estimated energy (mJ)</td>
<td>0.0473</td>
<td>0.0477</td>
<td>0.0479</td>
<td>0.0479</td>
</tr>
<tr>
<td>Error (%)</td>
<td>9.97</td>
<td>10.00</td>
<td>9.96</td>
<td>10.00</td>
</tr>
<tr>
<td>1000 multiplications - no cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Instructions</td>
<td>11513</td>
<td>11513</td>
<td>11513</td>
<td>11513</td>
</tr>
<tr>
<td>Measured energy (µJ)</td>
<td>6.1841</td>
<td>6.3430</td>
<td>6.6644</td>
<td>7.1480</td>
</tr>
<tr>
<td>Estimated energy (µJ)</td>
<td>6.1070</td>
<td>6.2638</td>
<td>6.5848</td>
<td>7.0648</td>
</tr>
<tr>
<td>Error (%)</td>
<td>-1.25</td>
<td>-1.25</td>
<td>-1.19</td>
<td>-1.16</td>
</tr>
<tr>
<td>FFT - no cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Instructions</td>
<td>11513</td>
<td>11513</td>
<td>11513</td>
<td>11513</td>
</tr>
<tr>
<td>Measured energy (µJ)</td>
<td>1.6713</td>
<td>1.7250</td>
<td>1.8344</td>
<td>1.9998</td>
</tr>
<tr>
<td>Estimated energy (µJ)</td>
<td>1.7399</td>
<td>1.7915</td>
<td>1.9016</td>
<td>2.0653</td>
</tr>
<tr>
<td>Error (%)</td>
<td>4.11</td>
<td>3.86</td>
<td>3.66</td>
<td>3.27</td>
</tr>
<tr>
<td>FFT - cache (I:32kB, D:8kB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Instructions</td>
<td>683485</td>
<td>683485</td>
<td>683485</td>
<td>683485</td>
</tr>
<tr>
<td>Measured energy (mJ)</td>
<td>0.1239</td>
<td>0.1239</td>
<td>0.1240</td>
<td>0.1241</td>
</tr>
<tr>
<td>Estimated energy (mJ)</td>
<td>0.1274</td>
<td>0.1274</td>
<td>0.1275</td>
<td>0.1276</td>
</tr>
<tr>
<td>Error (%)</td>
<td>2.82</td>
<td>2.82</td>
<td>2.82</td>
<td>2.81</td>
</tr>
<tr>
<td>FFT - cache (I:32kB, D:8kB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Instructions</td>
<td>683485</td>
<td>683485</td>
<td>683485</td>
<td>683485</td>
</tr>
<tr>
<td>Measured energy (mJ)</td>
<td>0.0436</td>
<td>0.0437</td>
<td>0.0437</td>
<td>0.0439</td>
</tr>
<tr>
<td>Estimated energy (mJ)</td>
<td>0.0480</td>
<td>0.0480</td>
<td>0.0481</td>
<td>0.0482</td>
</tr>
<tr>
<td>Error (%)</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
<td>9.99</td>
</tr>
</tbody>
</table>
5.5 Summary

This chapter has presented a method aimed at modeling composite transactions when building IP energy and performance models. This is done with the intent of maintaining accurate models regardless of the platform used. A Leon3 processor has been taken as a reference IP, while a System on Chip and a Network on Chip have been taken as reference platforms. After discussing the method, the resulting IP model has also been validated for energy estimation accuracy. The results have shown it to be within 10% of gate-level estimation.
Chapter 6

Level B – Inferring Primary Transactions and Refinements for Transactions Interdependency and Caches

This chapter applies the steps described in Section 3.4 about inferring primary transactions at the Algorithmic Level. At the same time, it also discusses a refinement related to transactions interdependency and caches. Inference of primary transactions is done for a set of common benchmark applications. Energy and performance numbers for these applications are then estimated by Funtime and compared against gate-level extraction. The Funtime estimation speed is instead compared against Transaction-Level Modeling. It is found out that Funtime estimation accuracy is within 15% of the gate-level estimation and that Funtime is about 30X faster than the corresponding TLM implementation. The material presented in this chapter is based on the author’s publication in [61].

6.1 Introduction

As discussed in Section 3.4, the Algorithmic Level (Level B) is Funtime’s main estimation level and is the end-user’s entry point for using the Funtime framework. By relying on the availability of an applications database $B$, of an Architecture Specification (AS) and of a Use-Case Specification (UCS), the Algorithmic Level infers a trace of transactions (primary transactions) for individual applications. These traces are then passed up to the Refinement Level (Level C), which applies a set of refinements to them (secondary transactions) and enables accurate energy and performance estimation. For better clarity, a diagram of the Algorithmic Level is reported in Figure 6.1.

The figure shows that instrumentation is the first step performed at Level B.
Instrumentation enables applications, which are intrinsically devoid of any architecture information, to become architecture-aware. After the instrumentation step is completed, the applications are executed on the evaluation host to infer and emit a trace of target transactions (primary transactions). The details have already been discussed in Sections 3.4.1 and 3.4.2. The reader is reminded that, although applications can be mapped both to hardware and software, this thesis only focuses on the mapping to software. Mapping to hardware is being pursued within the frame of the DRRA project.

This chapter presents a case study for Level B and applies the steps described in Section 3.4.1 for applications mapped to software on a set of common benchmark applications. The applications are mapped to the same reference SoC used in the pedagogical example built up throughout the whole of Chapter 3: a Leon3 processor exchanges information with a memory through an AMBA AHB Bus. The accuracy of Funtime energy and performance estimation at level B is validated against gate-level estimation in Section 6.3. Funtime estimation speed is instead validated in Section 6.5 for very large H264 and JPEG examples against TLM.

Although part of the Refinement Level, this chapter also presents two refinements, namely the transactions interdependency refinement and the cache refinement. The reason is that such refinements are tightly bound to the process of inferring traces of transactions for individual applications, which is what Level B does. As previously discussed in Section 3.9 these are also the first and second refinement applied in the sequence of the refinements order.

The material presented in this chapter is based on the previous work published by the author in [61].

### 6.2 Transactions Interdependency Refinement

When presenting the Leon3 energy and performance model in Chapter 4 it was stated that the model would account for the impact of instructions interdependency on energy and number of cycles. It is emphasized that the expression “instructions
interdependency” refers to the fact that energy and cycles associated with a transaction often depend not only on the properties of the transaction itself, but also on the actual sequence in which transactions are issued. Therefore, in a processor case, its instructions execution order plays an important role in affecting the total application energy and cycles.

To account for instructions interdependency, a model based on pairs of instructions has thus been implemented, and it has proved to be more accurate than a single-instruction-based model. However, in Section 4.7 where the model has been validated, it was also stated that the total number of cycles estimated by the model was always lower than the actual number extracted from gate-level simulation. This difference has then been associated with the occurrence of stall conditions. As an example, in a system without cache, if a load instruction is issued right after a store to the same memory location, a stall condition will occur, since the data to be loaded has not yet been stored into memory.

This being said, the reason why the Leon3 model is not able to incorporate the stalls effect is related to the fact that it only considers interdependency effects between a certain instruction $I_x$ and the preceding one $I_{x-1}$. In other words, the model only accounts for first-order effects, and does not go beyond that. Given that the Leon3 pipeline has seven stages, a model considering interdependency effects up to the sixth-order would probably be able to also account for the impact of stalls. This is, however, only a purely theoretical consideration, since such a model would be unreasonably large in terms of space and time required to build.

In a traditional simulation-based approach, the exact number of extra cycles due to stall conditions comes along with the simulation itself and their energy contribution can therefore be included in the overall energy estimation. This is what has been done in Section 4.7. In that case, since the purpose was only to validate the Leon3 model, there was only one source of inaccuracy, coming from the model itself, while the number of cycles and instructions was taken from simulation.

Instead, at Level B, both number of cycles and instructions have to be inferred in a different way. Section 3.4 has already discussed how Funtime deals with inferring the number and the actual trace of instructions. The present section now explains how it can also infer the number of stalls through the refinement step described below.

The approach proposed in the present work relies on the insight that, even if applications differ in terms of inferred transactions (instructions) trace, their instructions are executed a sufficiently large number of times and in so many different combinations, that the average behavior as a model is justified. To find such an average-behavior model, the following empirical method has been used. A small number of reference applications was taken and used as the basis for calibration. Specifically, such applications are first run on a platform and the average number of cycles per instruction is collected for each such application. Afterwards, calibration is performed by taking a weighted average of the cycles per instruction over the reference applications. In each application, the weighting factor is given by the percentage of times that each instruction is issued over the total number
CHAPTER 6. LEVEL B – INFERRING PRIMARY TRANSACTIONS AND REFINEMENTS FOR TRANSACTIONS INTERDEPENDENCY AND CACHES

of instructions. The resulting numbers are then taken as a model and applied to new applications to estimate the total applications length in terms of cycles. The difference between the total application cycles found by using the refined values and the total application cycles found by using the ideal values given by the processor model represents the contribution of the stalls and, consequently, the effect of transactions interdependency.

The assumption about the possibility of finding typical patterns across different applications relies on the fact that predictability is usually strongly desired in embedded systems, and especially in those systems that have to fulfill real-time constraints. In addition, applications often consist of large loops which are identically repeated a large number of times. This further helps to support this assumption. To achieve the highest possible accuracy, applications may be grouped according to the main functionality that they implement and calibration can then be done on each individual group. For example, video, audio and image encoding/decoding applications could represent individual groups.

Since in Section 4.7 the quantity $E_{\text{stall}}/\text{cycle}$ has been defined as the stall energy per cycle, it now makes sense to associate the duration of a single stall to one cycle. Under these assumptions, the total number of stalls can be expressed as in Equation 6.1:

$$\#\text{Stalls} = \text{Tot}_{\text{estimated}} - \text{Tot}_{\text{ideal}} \tag{6.1}$$

Similarly, the total energy contribution due to the stalls can be calculated as in Eq. 6.2

$$E_{\text{stalls}} = \#\text{Stalls} \cdot E_{\text{stall}}/\text{cycle} \tag{6.2}$$

In the context of the research presented in this thesis, the transactions interdependency refinement has been applied to a SoC based on a Leon3 processor that exchanges information with a memory through an AMBA AHB bus. The Leon3 processor has been configured with no cache to avoid the introduction of extra sources of indeterminism. A set of three different applications has been taken as a reference and used for calibration, namely an FFT, a Quicksort and a Fibonacci series. Table 6.1 shows a comparison between the ideal processor cycles when a cache hit occurs and the refined ones for a bunch of Leon3 instructions. The transactions interdependency refinement applied to the Algorithmic Level has been validated in Section 6.3.

6.3 Validating the Algorithmic Level and the Accuracy of the Transactions Interdependency Refinement

Gate level has been used as a reference for validation, since it is very accurate and has been used to build the Leon3 energy and performance model. The results are reported in Tables 6.2 and 6.3. These tables validate the accuracy of the Algorithmic Level plus the transactions interdependency refinement for seven benchmarks.
6.3. VALIDATING THE ALGORITHMIC LEVEL AND THE ACCURACY OF THE TRANSACTIONS INTERDEPENDENCY REFINEMENT

Table 6.1: Comparison between ideal and real average cycles per instruction

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Ideal cycles</th>
<th>Real avg. cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1.00</td>
<td>7.58</td>
</tr>
<tr>
<td>ld</td>
<td>1.00</td>
<td>15.19</td>
</tr>
<tr>
<td>nop</td>
<td>1.00</td>
<td>3.23</td>
</tr>
<tr>
<td>st</td>
<td>2.00</td>
<td>5.82</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

by introducing a growing number of sources of inaccuracy. Note that the three applications on the top of each table represent the calibration suite referenced in the refinement section above, whereas the four applications at the bottom are the validation set, not part of the calibration. In both tables, the Leon3 configuration without cache has been selected.

Table 6.2 introduces two sources of inaccuracy: the first refers to the transactions interdependency refinement used to estimate the total number of cycles (ideal + stalls contribution); the second refers to the Leon3 energy and performance model. The number of actual instructions is measured. Table 6.3 introduces instead three sources of inaccuracy: the first two are the same as in Table 6.2 while the third refers to the number of transactions (instructions) estimated by the Algorithmic Level. Even in this case, the maximum error of Funtime for energy estimation is around 15% of the gate level.

Table 6.2: 2 sources of inaccuracy: Leon3 energy & performance model + estimated number of cycles (stalls)

<table>
<thead>
<tr>
<th>Calibration benchmarks</th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td># Instructions</td>
<td>1 371 584</td>
<td>4 008 651</td>
<td>1 585 313</td>
</tr>
<tr>
<td>Real # cycles</td>
<td>6 485 094</td>
<td>31 649 232</td>
<td>13 902 019</td>
</tr>
<tr>
<td>Estimated # cycles</td>
<td>7 217 187</td>
<td>31 153 927</td>
<td>13 251 730</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+11.29</td>
<td>-1.56</td>
<td>-4.68</td>
</tr>
<tr>
<td>Real energy [mJ]</td>
<td>0.2396</td>
<td>1.1508</td>
<td>0.4906</td>
</tr>
<tr>
<td>Estimated energy [mJ]</td>
<td>0.2746</td>
<td>1.0719</td>
<td>0.4509</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+14.62</td>
<td>-6.86</td>
<td>-8.10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Verification benchmarks</th>
<th>Viterbi</th>
<th>Queens</th>
<th>Dhrystone</th>
<th>MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td># Instructions</td>
<td>2 452 903</td>
<td>4 266 783</td>
<td>1 950 813</td>
<td>1 802 380</td>
</tr>
<tr>
<td>Real # cycles</td>
<td>17 511 989</td>
<td>24 193 885</td>
<td>15 938 144</td>
<td>14 428 588</td>
</tr>
<tr>
<td>Estimated # cycles</td>
<td>18 484 266</td>
<td>24 878 824</td>
<td>15 660 388</td>
<td>14 082 568</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+5.55</td>
<td>+2.83</td>
<td>-1.74</td>
<td>-2.40</td>
</tr>
<tr>
<td>Real energy [mJ]</td>
<td>0.6250</td>
<td>0.8571</td>
<td>0.5641</td>
<td>0.5228</td>
</tr>
<tr>
<td>Estimated energy [mJ]</td>
<td>0.6648</td>
<td>0.9013</td>
<td>0.5498</td>
<td>0.4989</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+6.36</td>
<td>+5.15</td>
<td>-2.54</td>
<td>-4.74</td>
</tr>
</tbody>
</table>
Table 6.3: 3 sources of inaccuracy: Leon3 energy & performance model + estimated number of cycles (stalls) + estimated number of instructions

<table>
<thead>
<tr>
<th>Calibration benchmarks</th>
<th>FFT</th>
<th>Quicksort</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real # instructions</td>
<td>1 371 584</td>
<td>4 008 651</td>
<td>1 585 313</td>
</tr>
<tr>
<td>Estimated # instructions</td>
<td>1 273 927</td>
<td>3 824 253</td>
<td>1 517 196</td>
</tr>
<tr>
<td>Error [%]</td>
<td>-7.12</td>
<td>-4.6</td>
<td>-4.3</td>
</tr>
<tr>
<td>Real # cycles</td>
<td>6 485 094</td>
<td>31 649 232</td>
<td>13 902 019</td>
</tr>
<tr>
<td>Estimated # cycles</td>
<td>6 731 528</td>
<td>29 427 456</td>
<td>12 185 120</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+3.8</td>
<td>-7.02</td>
<td>-12.35</td>
</tr>
<tr>
<td>Real energy [mJ]</td>
<td>0.2396</td>
<td>1.1508</td>
<td>0.4906</td>
</tr>
<tr>
<td>Estimated energy [mJ]</td>
<td>0.2666</td>
<td>1.0417</td>
<td>0.4061</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+11.29</td>
<td>-9.48</td>
<td>-17.23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Verification benchmarks</th>
<th>Viterbi</th>
<th>Queens</th>
<th>Dhrystone</th>
<th>MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real # instructions</td>
<td>2 452 903</td>
<td>4 266 783</td>
<td>1 950 813</td>
<td>1 802 380</td>
</tr>
<tr>
<td>Estimated # instructions</td>
<td>2 398 390</td>
<td>4 006 083</td>
<td>1 880 389</td>
<td>1 707 214</td>
</tr>
<tr>
<td>Error [%]</td>
<td>-2.22</td>
<td>-6.11</td>
<td>-3.61</td>
<td>-5.28</td>
</tr>
<tr>
<td>Real # cycles</td>
<td>17 511 989</td>
<td>24 193 885</td>
<td>15 938 144</td>
<td>14 428 588</td>
</tr>
<tr>
<td>Estimated # cycles</td>
<td>17 783 529</td>
<td>23 680 975</td>
<td>15 131 674</td>
<td>13 233 901</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+3.55</td>
<td>-2.12</td>
<td>-5.06</td>
<td>-8.28</td>
</tr>
<tr>
<td>Real energy [mJ]</td>
<td>0.6250</td>
<td>0.8571</td>
<td>0.5641</td>
<td>0.5228</td>
</tr>
<tr>
<td>Estimated energy [mJ]</td>
<td>0.6611</td>
<td>0.9054</td>
<td>0.5265</td>
<td>0.4588</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+5.78</td>
<td>+5.64</td>
<td>-6.67</td>
<td>-12.25</td>
</tr>
</tbody>
</table>

6.4 Cache Effect Refinement

The presence of caches is an additional factor that affects the estimation accuracy at the Level B of the Funtime approach. The refinement applied in this case aims at quantifying and modeling the variation of an application execution time and energy in a system with cache, given the cache miss ratio $MR$ and compared to the case with no cache. The cache miss ratio must thus be provided as an input to Funtime, in the UCS. As opposed to the effect of transactions interdependency, caches reduce the total amount of application cycles, since there are fewer stalls caused by communication with memory.

Also in this case, the way in which the refinement is implemented relies on the assumption that the percentage of cycles variation for a given cache miss ratio, compared to the case without cache, is independent from the application and use case for an application. This assumption is again justified by the fact that an average behavior can be modeled using a calibration set as shown in Figure 6.2 as a consequence of the large number of instructions and instruction sequences generated by each application. Similarly to the previous refinement, a small set of applications can therefore be taken as the basis for calibration and the results of calibration can then be applied to refine new applications, which form the validation suite as shown in Figure 6.2. Calibration in this case consists in finding an interpolation function.

The same calibration benchmarks and SoC have been used as for the trans-
actions interdependency refinement. A Leon3 has been configured with a 1k, 2k, 4k I/D cache. The interpolation function results in this case in a straight line, expressed in Equation 6.3:

\[ C\% = 1.09 \cdot MR\% + 19.19 \]  

where \( C\% \) represents the percentage of total application cycles compared to the case without cache and \( MR\% \) the miss ratio in percentage. Note that, for an ideal case of \( MR\% = 0\% \), an application would complete in about 20% of the time with respect to the case with no cache. For the opposite case of \( MR\% = 100\% \), the equation cannot be applied though, as \( C\% > 100\% \). It is expected that, starting from a certain point, the function should grow with a smaller slope, so that \( C\% = 100\% \) when \( MR\% = 100\% \). The experiments conducted in this work only consider an \( MR\% \) up to 25%, value for which the equation still applies.

Validation for the cache refinement is presented in Figure 6.2. In this case, Equation 6.3, found as an interpolation for the calibration benchmarks, has also been used to estimate the cycles variation also for the other four benchmarks on the right when using a Leon3 with 1k, 2k, 4k I/D cache. Figure 6.3 shows both the interpolation straight line extracted from the calibration benchmarks (dashed line) and the real interpolation straight line for the other four applications (thick line). The small difference between the two proves that using Equation 6.3 as a reference for all the applications is sensible and the error introduced is minimal.

![Validation benchmarks](image)

Figure 6.2: Validating the cache refinement

6.5 Comparing Funtime and Untimed TLM Estimation Speed

For speed comparison, untimed TLM has been chosen as a reference. The reason is that this represents the fastest high-level methodology for system-level estimation commonly used at present. For this purpose, a dedicated transaction-level model in SystemC has been implemented for the reference SoC architecture. The
implementation has been as abstract as possible, since it exclusively represents the transactions occurring across the platform among the different IPs. In addition, communication is handled using bidirectional blocking interfaces.

For this experiment, a set of very large application examples has been chosen. The reason was to both show that Funtime can handle very large examples and also to make the difference between Funtime and TLM execution speed more evident. The same applications could not be used to validate Funtime accuracy, however, because gate-level estimation for such large use cases is not realistic.

The applications chosen are the image compression codec JPEG2000 and the video compression codec H264. The first has been applied to three images of sizes 128x128, 256x256 and 512x512. The second has been used for two videos of three frames each, with a resolution of 176x144 and 352x288 respectively. The number of total executed instructions ranges from 80 milion to 1.6 billion. The results of this comparison are reported in Table 6.4 and show a mean speed improvement of 28X for Funtime compared to TLM. Besides, note that the advantage of Funtime over TLM increases as the number of instructions increases. This confirms the capacity of Funtime to be used for complex and real use-case scenarios.

Table 6.4: Timing comparison between Funtime and untimed TLM

<table>
<thead>
<tr>
<th># transactions</th>
<th>TLM</th>
<th>Funtime</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jpeg2k_128x128</td>
<td>82M</td>
<td>3.00 sec</td>
<td>0.62 sec</td>
</tr>
<tr>
<td>Jpeg2k_256x256</td>
<td>272M</td>
<td>10.30 sec</td>
<td>0.69 sec</td>
</tr>
<tr>
<td>Jpeg2k_512x512</td>
<td>986M</td>
<td>38.66 sec</td>
<td>1.04 sec</td>
</tr>
<tr>
<td>H264_176x144</td>
<td>1.68B</td>
<td>50.75 sec</td>
<td>1.41 sec</td>
</tr>
<tr>
<td>H264_352x288</td>
<td>2.55B</td>
<td>79.80 sec</td>
<td>1.58 sec</td>
</tr>
</tbody>
</table>
Chapter 7

Level C – Refining Bus Contention
Effects on Energy and Performance

At Level C, Funtime refines the traces of primary transactions, emitted by Level B for individual applications, by combining them with a new trace of secondary transactions. As opposed to Level B, which considers applications in isolation, this refinement process basically accounts for the implications of applications having to share common resources. This chapter shows the case study of an MPSoC, where applications mapped to different processors have to share the same bus. A high-level method is proposed for rapidly and accurately predicting the effects of bus contention on energy and performance. The approach consists of two parts: first, a one-time activity performed at Level A2, aimed at identifying and characterizing the main factors involved in bus contention; second, the development of a prediction mechanism, performed at Level C, to account for the occurrence of such factors for an arbitrary number of processors hosting arbitrary applications. The approach is also validated against gate-level estimation, and finds out an error within 2%. The material presented in this chapter is based on the previous work published by the author in [59].

7.1 Introduction

The ideal scenario has been considered so far where applications are independent and have all the system resources available to themselves. At the Level B, traces of primary transactions are emitted, which reflect the functionality of applications treated in isolation. Nevertheless, in reality there are some implications due to the fact that applications do not have all the resources always available to themselves and often they even have to interact with other applications. The most visible effect of these non-idealities is a variation of the total application execution time and final energy consumption.

Accounting for the impact of such non-idealities is done in Funtime at the Level
C, also called Refinement Level. At this level, the trace of primary transactions received from Level B is refined through the injection of secondary transactions. The combination of primary and secondary transactions represents the final trace for which total energy and execution time can be estimated. Estimation is made possible through the IP energy and performance models, extended at Level A2 and also passed as an input to Level C.

Multi-Processor Systems on Chip (MPSoCs) have emerged as one possible solution to cope with the demand of increasing performance and computational efficiency in modern embedded architectures. The performance advantage derives from running applications in parallel by distributing them across multiple computational resources, such as processors. In an MPSoC, all the processors are typically connected to the same bus and contention occurs when more than one application, mapped to a certain processor, requests an exclusive access to a shared resource. The main consequence of contention is an overall degradation in performance and an increase in energy consumption.

Refining the effects of bus contention for an MPSoC case study is the purpose of this chapter. A high-level method is proposed for rapidly and accurately predicting the effects of bus contention on energy and performance. The content of this chapter is based on the author’s previous work in [59]. In particular, the bus contention refinement is implemented through the following two steps:

1. **Characterizing bus contention**: this step, which is part of Level A2 and symbolically shown at the bottom of Figure 7.1 consists in identifying and charac-
7.2 CHARACTERIZING BUS CONTENTION

Characterizing the main factors responsible for bus contention in MPSoCs. Although time consuming, this step is a one-time activity that should be carried out by the IP provider and that relies on executing gate-level simulations and energy extractions for meaningful benchmarks. Such benchmarks are used for characterization only and are the basis for the following step. This is detailed in Section 7.2.

2. Predicting bus contention: this step, which is part of Level C and symbolically shown on the top of Figure 7.1, aims at predicting the occurrences of bus contention and quantifying them in terms of overall system performance degradation and energy consumption increase, for an MPSoC with \( N \) processors running a set of \( N \) user-defined applications. Such applications are different from the ones used in the characterization benchmark. Bus contention prediction is done with respect to the single-processor SoC, which is used as a reference. Unlike the step above, this step is very fast, since it does not involve simulation. In addition, this step is carried out by the system designer during design-space exploration. All this is detailed in Section 7.3.

7.2 Characterizing Bus Contention

The purpose of this section is to identify and characterize the architectural factors that are involved in bus contention.

Recall that \( e_{xi} \) and \( E_i \) have been previously defined as the execution time and energy consumption of the \( i \)-th application respectively, assuming that this runs on a single-processor SoC and has all the resources available for itself. In this ideal case, no bus contention occurs. Executing \( N \) times the same application on this single-processor SoC would result in an execution time \( e_{xN} = N \cdot e_{xi} \) and energy consumption \( E_N = N \cdot E_i \). Assuming now having an MPSoC with \( N \) processors and running the same application in \( N \) identical copies, each one mapped onto one processor, an energy consumption \( E_N = N \cdot E_i \) is still expected, as opposed to an execution time \( e_{xN} = e_{xi} \). However, because of bus contention, it is \( E_N = N \cdot (E_i + \Delta E_i) \) and \( e_{xN} = e_{xi} + \Delta e_{xi} \).

Identifying and analytically expressing the factors responsible for \( \Delta e_{xi} \) and \( \Delta E_i \) is the focus of this section. Being able to predict the value of these two quantities on MPSoCs with \( N \) processors running \( N \) user-defined applications is the focus of the next Section 7.3.

The reference architecture is a SoC composed of one Leon3 processor, one shared SRAM memory and one AHB controller. This basic configuration is then extended to a homogeneous MPSoC hosting up to seven processors, as shown in Figure 7.2.

The reason for using a single shared memory instead of multiple memories is that the reference bus is a single-layer AHB. In this case, having a single memory or multiple memories would make no difference, since the contention already happens at the bus level. The multi-layer AHB case is a more general case than the case investigated in this paper, for which the results presented here can be extended.
A homogeneous SoC consisting of Leon3 processors is used here. In the case of heterogeneous SoC, the analytical expressions shown in the upcoming sections would clearly look different; however, the approach proposed in this work would still be valid. In addition, processors without cache are always used in the experiments proposed for this chapter. This choice makes sense when considering systems for real-time applications, whose response predictability is essential. Finally, a Round Robin scheduling policy was chosen for bus arbitration.

A set of three different applications is used as a characterization benchmark. Such applications run at gate level, which allows us to get accurate numbers for execution time, energy consumption and a detailed instruction trace. The goal is to find meaningful analytical expressions for $\Delta ex_i$ and $\Delta E_i$. The benchmark applications are an FFT, a Fibonacci and a Viterbi. In this characterization phase, the number $N$ of processors is varied between 1 and 7. Besides, the same application runs in $N$ identical copies, each one mapped onto one processor. The following also needs to be clarified.

First, these applications have been chosen as they are of common use and their algorithms are quite regular. Second, the goal of using the same application on all processors is to keep the setup as simple as possible, by avoiding extra sources of non-determinism that come from running different applications and thus easing the process of modeling bus contention. Nonetheless, the resulting models are general and can also be used when each processor runs a different application. This is described in the case study in Sections 7.3.1 and 7.4. Third, although using $N$ identical applications on $N$ processors, these are not synchronized with each other. Finally, in order to ease the visualization of the results, regardless of the number of processors in the MPSoC, results are shown only for the processor bound to the bus at index 0.

Note that, due to the very long time necessary to run gate-level simulations and energy extractions, the number of applications used for calibration has been limited to three and only Round Robin has been considered for bus arbitration. It is emphasized that this activity would be a one-time effort carried out by the IP provider. Extending the applications set and investigating bus contention for priority-based bus arbitration is part of the future work.

Two major factors affecting bus contention are identified: the number of processors and the number of w/r accesses to memory. Both factors are detailed in the next Subsections 7.2.1 and 7.2.2 respectively.
7.2. CHARACTERIZING BUS CONTENTION

7.2.1 Factor 1: number of processors

Figure 7.3 shows the execution time of each of the three applications running on the different SoC configurations.

Figure 7.3: Execution time \( e_{x_i} \) (in number of cycles) for FFT, Fibonacci and Viterbi

The first factor involved in bus contention is the number of processors, which has a nearly linear impact and shows a similar progression in all the three benchmarks considered. The increase of execution cycles is associated with stall conditions for the processors, due to the fact that these have to compete against each other to get access to the bus.

The percentage of cycles difference between the \( N \)-processor and the 1-processor case is defined as \( C_{x,y}^\% \), where \( x = N \) and \( y = 1 \), for the benchmark applications chosen. The percentage is calculated having the 1-processor case as a reference. The results, shown in Table 7.1, suggest that this value is quite uniform for the same comparison interval across the different applications.

Table 7.1: Percentage of cycles difference \( C_{N,1}^\% \) between \( N \) and 1-processor case

<table>
<thead>
<tr>
<th></th>
<th>( C_{2,1}^% )</th>
<th>( C_{3,1}^% )</th>
<th>( C_{4,1}^% )</th>
<th>( C_{5,1}^% )</th>
<th>( C_{6,1}^% )</th>
<th>( C_{7,1}^% )</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>29.84</td>
<td>34.21</td>
<td>43.07</td>
<td>69.67</td>
<td>95.56</td>
<td>129.32</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>26.59</td>
<td>37.35</td>
<td>48.90</td>
<td>71.61</td>
<td>94.54</td>
<td>126.97</td>
</tr>
<tr>
<td>Viterbi</td>
<td>27.29</td>
<td>34.43</td>
<td>51.23</td>
<td>75.96</td>
<td>101.44</td>
<td>135.01</td>
</tr>
</tbody>
</table>

Assuming that \( N \) is the total number of processors and considering the i-th application running on the i-th processor, where \( i \leq N \), the execution time \( e_{x_i} \) and energy consumption \( E_{x_i} \) of each i-th application can be expressed as \( e_{x_i}(N) = e_{x_i}(1) + \Delta e_{x_i}(N) \) and \( E_{x_i}(N) = E_{x_i}(1) + \Delta E_{x_i}(N) \). Note that \( e_{x_i}(1) \) and \( E_{x_i}(1) \) correspond to the ideal case where an application runs on a single-processor SoC, with no bus contention. Also note that the quantity \( e_{x_i}(1) \) is an output of the Level B and is passed as an input to Level C. This is visible from Figure 7.1. The quantity \( C_i(N) \) is also introduced to indicate the number of execution cycles for the i-th application in the N-processor SoC case. Note that the value of \( C_i(1) \) is
available through the relation $C_i(1) = ex_i(1)/T$, where $T$ is the clock period. The quantity $\Delta ex_i(N)$ with $N > 1$ can thus be expressed using the simple Equation 7.1:

$$\Delta ex_i(N) = T \cdot C_{N,1}^\% \cdot C_i(1)$$

(7.1)

Table 7.2 shows the percentage of cycles difference $C_{x,y}^\%$ between the $N$-processor case and the $(N-1)$-processor case, where $x = N$ and $y = N - 1$, for the benchmark applications chosen. The $(N - 1)$-processor case is taken as a reference. A red box highlights an interesting result corresponding to the percentage of cycles difference between the 7 and the 6-processor configuration. For all the three benchmarks this difference is 16.67%.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$C_{2,1}^%$</th>
<th>$C_{3,2}^%$</th>
<th>$C_{3,3}^%$</th>
<th>$C_{5,4}^%$</th>
<th>$C_{6,5}^%$</th>
<th>$C_{7,6}^%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>29.84</td>
<td>3.36</td>
<td>6.61</td>
<td>18.59</td>
<td>15.85</td>
<td>16.67</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>26.59</td>
<td>8.50</td>
<td>8.41</td>
<td>15.25</td>
<td>13.36</td>
<td>16.67</td>
</tr>
<tr>
<td>Viterbi</td>
<td>27.29</td>
<td>5.61</td>
<td>12.49</td>
<td>16.35</td>
<td>14.48</td>
<td>16.67</td>
</tr>
</tbody>
</table>

In order to understand whether this quantity would also remain constant for configurations with a higher number of processors, the FFT benchmark was run on a SoC with up to 15 Leon3 processors. The results are shown in Figure 7.4.

![Figure 7.4: Execution time $ex_i$ (in cycles) for FFT, $1 \leq N \leq 15$](image)

It is evident that, taken the 6-processor configuration as a reference, it results that $C_{N,6}^\% = (N - 6) \cdot 16.67$, for $N > 6$. This indicates a strong regularity in the increment of cycles for configurations with a number of processors $N > 6$. Note that these results are representative of the case where Round Robin is used for bus arbitration.
Three regions are identified in Figure 7.4 highlighted by dashed circles. The circle in the middle shows an almost constant number of cycles for the SoC configurations with 2, 3 and 4 processors. In this case the bus controller allocates to each processor requesting the bus a time slot with the width of a bus transfer. In the case of a 4-processor SoC, each processor will thus be granted a bus access every fourth bus transfer. It appears that this time frame is short enough to allow a processor to continue its execution without stalling until it regains control over the bus. That is why the performance penalty introduced is insignificant.

However, for configurations with more than 4 processors, this time frame becomes too long and the processor has to stall for some time before it can access the bus again and continue its execution. This leads to the behavior highlighted by the circle on the right and to the 16.67% time penalty for any extra processor.

Finally, the circle on the left highlights an execution time difference between the configurations with 1 and 2 processors, which apparently contradicts the interpretation given above for the configurations with 2 to 4 processors. The explanation is found in the way the bus controller operates: if there is only one master requesting the bus, no arbitration is done and thus no arbitration penalty is introduced.

The energy difference has also been measured for the same benchmark applications on the 7 SoC configurations, which results in a linear energy increase proportional to the processors number. The increase in energy consumption is a direct consequence of the increased execution time, which has been previously attributed to stall conditions for a processor. Indirectly, it can thus be stated that the energy difference \( E_i(N) - E_i(1) \), with \( N > 1 \), is also due to stall conditions. In particular, assuming that \( E_{\text{stall cycle}} \) is the stall energy overhead per clock cycle, the following Equation 7.2 can be written:

\[
\frac{E_{\text{stall cycle}}}{C_i(N) - C_i(1)} = \frac{E_i(N) - E_i(1)}{C_i(N) - C_i(1)}
\]  

(7.2)

The quantity \( E_{\text{stall cycle}} \) is expected to be a constant value. Table 7.3 below reports the value of \( E_{\text{stall cycle}} \) calculated by applying Equation 7.2 to every SoC configuration with \( 2 \leq N \leq 7 \) and running the above-mentioned three reference benchmarks. The consistency of the results confirms the predictability of the stall energy consumption per clock cycle, which has an average value \( \hat{E}_{\text{stall cycle}} = 24.85 \text{ pJ} \).

Note that the value of \( E_{\text{stall cycle}} \) reported in this chapter differs from the one calculated in Chapter 4 when presenting the Leon3 energy and performance model. This difference can be justified by the difficulty of isolating the exact contribution of each single instruction when building the Leon3 model, and thus by the risk of including extra contributions. Therefore, applying Equation 4.8 to every such instruction has not always produced the same result. The average value has then been taken and reported as the reference \( E_{\text{stall cycle}} \) value in Chapter 4.

With this in mind, the goal is to analytically express the energy overhead \( \Delta E_i(N) \) for the i-th application as a function of the number of processors \( N \). This quantity was previously defined as the extra energy consumed by the i-th applica-
CHAPTER 7. LEVEL C – REFINING BUS CONTENTION EFFECTS ON
ENERGY AND PERFORMANCE

Table 7.3: Value of $E_{\text{stall \ cycle}}$ [pJ] calculated by Equation \ref{eq:7.2} for $2 \leq N \leq 7$

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>25.49</td>
<td>23.48</td>
<td>25.05</td>
<td>24.23</td>
<td>24.88</td>
<td>24.94</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>25.21</td>
<td>24.18</td>
<td>25.02</td>
<td>24.15</td>
<td>25.02</td>
<td>24.87</td>
</tr>
<tr>
<td>Viterbi</td>
<td>25.52</td>
<td>25.43</td>
<td>24.88</td>
<td>24.68</td>
<td>25.17</td>
<td>25.14</td>
</tr>
</tbody>
</table>

The relation because of bus contention, which is as a function of the number of processors $N$. The relation is shown in Equation \ref{eq:7.3}

$$\Delta E_i(N) = \frac{E_{\text{stall \ cycle}}}{\Delta ex_i(N)}$$
(7.3)

7.2.2 Factor 2: number of w/r accesses to memory

Besides the number of processors, investigation is also conducted on the impact that w/r memory accesses have on bus contention compared to instruction fetching. Recall that the reference SoC is composed of processors without cache.

Using the FFT as a reference, the instruction-level difference $C_{7,1}^\%$ is measured in the execution cycles between the 7-processor and the single-processor SoC. Architectures with such a high different number of processors are compared in order to make the differences more tangible. Table 7.4 shows the results.

Table 7.4: Percentage of instruction-level cycles difference $C_{7,1}^\%$ for the FFT

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Distr. [%]</th>
<th>$C_{7,1}^%$</th>
<th>Instr.</th>
<th>Distr. [%]</th>
<th>$C_{7,1}^%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>0.00</td>
<td>45.00</td>
<td>st</td>
<td>7.19</td>
<td>135.25</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>sub</td>
<td>0.20</td>
<td>135.29</td>
</tr>
<tr>
<td>be,a</td>
<td>0.37</td>
<td>126.36</td>
<td>sl</td>
<td>1.63</td>
<td>136.31</td>
</tr>
<tr>
<td>ble,a</td>
<td>0.07</td>
<td>129.80</td>
<td>sra</td>
<td>0.32</td>
<td>136.36</td>
</tr>
<tr>
<td>be</td>
<td>1.74</td>
<td>130.00</td>
<td>restore</td>
<td>0.89</td>
<td>137.67</td>
</tr>
<tr>
<td>ret</td>
<td>0.89</td>
<td>131.12</td>
<td>bg</td>
<td>0.22</td>
<td>140.25</td>
</tr>
<tr>
<td>addx</td>
<td>0.69</td>
<td>131.91</td>
<td>std</td>
<td>1.03</td>
<td>141.88</td>
</tr>
<tr>
<td>bge,a</td>
<td>0.00</td>
<td>133.33</td>
<td>bleu</td>
<td>0.67</td>
<td>149.94</td>
</tr>
<tr>
<td>srl</td>
<td>3.20</td>
<td>133.61</td>
<td>add</td>
<td>11.48</td>
<td>154.17</td>
</tr>
<tr>
<td>bne,a</td>
<td>0.01</td>
<td>134.23</td>
<td>clr</td>
<td>0.03</td>
<td>178.84</td>
</tr>
<tr>
<td>mul/sc</td>
<td>24.01</td>
<td>134.59</td>
<td>ldd</td>
<td>0.71</td>
<td>181.88</td>
</tr>
<tr>
<td>bl</td>
<td>0.06</td>
<td>135.05</td>
<td>ld</td>
<td>2.29</td>
<td>208.01</td>
</tr>
</tbody>
</table>

Table 7.4 consists of two columns, each of which contains three sub-columns. From left to right, these sub-columns show the name of the instruction, its occurrence compared to the other instructions, and the percentage of execution cycles difference $C_{7,1}^\%$. The items are sorted according to increasing values of $C_{7,1}^\%$. Clearly, for all instructions the value of $C_{7,1}^\%$ is positive. This makes sense, since processors
without cache are being used, which implies that all instructions need to be fetched directly from memory. Contention can therefore happen for any instruction.

The reader is reminded that the purpose of this analysis is to see whether the value of $C_{T,1}^{\%}$, for instructions related to store/load operations, is above the average compared to the rest of the instructions. If this is the case, considering the load/store-related instructions as a second factor in modeling bus contention can be useful to improve the estimation accuracy. In our example, the average value of $C_{T,1}^{\%}$ is 129.32\%, when measured on an application-level basis. A dashed line has been used as a separator between those instructions whose $C_{T,1}^{\%}$ is below and above average respectively. Red boxes have then been used to highlight all the load/store-related instructions. Not surprisingly, all such instructions show a value for $C_{T,1}^{\%}$ above average. In fact, in this case, not only fetching represents a possibility for contention, but also writing/reading data to/from memory. Nonetheless, there are also some non-store/load-related instructions exhibiting execution time above average. Some of them operate on data, however – such as add, sll and sub – and a delay can be introduced if the data is not available when the instruction is issued.

Although the occurrence of load/store-related instructions is lower than 12\% of the total in the FFT case, this value rises to $\sim$27\% in the Fibonacci and Viterbi cases. Thus, it seems that explicitly considering the load/store-related instructions as a second factor in modeling bus contention can improve the estimation accuracy for $\Delta ex_i$. This can be done by separately considering the contribution of the load/store-related instructions and the non-store/load-related ones. If the former and latter group of instructions are denoted by the symbol $wr$ and $\overline{wr}$ respectively, Equation 7.1 can be reformulated as Equation 7.4:

$$\Delta ex_i(N, wr) = T \cdot [C_{wr,N,1}^{\%} \cdot C_{i,wr}(1) + C_{\overline{wr},N,1}^{\%} \cdot C_{i,\overline{wr}}(1)] \quad (7.4)$$

Note that the methodology presented above will be generalized in the next sections and applied to the case study in Section 7.3.1.

### 7.2.3 Summary

The key concepts introduced so far can be summarized as follows:

1) The symbols $ex_i(1)$ and $E_i(1)$ refer to the execution time and energy consumption of the i-th application for the single-processor case, where no contention occurs. These are also the quantities produced by the Level B of Funtime.

2) The execution time variation $\Delta ex_i(N)$ and energy variation $\Delta E_i(N)$ have been analytically expressed for the N-processor case with $N > 1$ compared to the case where $N = 1$. See Equations 7.1 and 7.3.

3) Two main factors affecting bus contention have been considered: the number of processors and the number of w/r accesses to memory. Considering this second factor allows the extension of Equation 7.1 into Equation 7.4.

4) Since simulating and extracting energy at gate level is very time-consuming, the analysis has been limited to three examples using configurations with up to
seven processors. These examples are used for characterization and for the creation of analytical expressions for \( \Delta ex_i(N) \) and \( \Delta E_i(N) \). During validation in Section 7.4 a different set of examples is used to ensure the generality and accuracy of these expressions. Only for the FFT application has the analysis been extended to a SoC with up to 15 processors. This allowed a very regular behavior to be measured for configurations with more than seven processors. See Figure 7.4.

5) Despite the very long characterization time, this is a one-time activity, which should be done by the IP provider. The equations and the quantities extracted in this section will be the basis for the upcoming bus contention prediction section.

### 7.3 Predicting Bus Contention

The present section elaborates on how the results from Section 7.2 can be used and extended to predict the bus contention occurrences on an \( N \)-processor SoC running \( N \) arbitrary applications. It is emphasized that bus contention occurrences are evaluated in terms of variation in the execution time \( \Delta ex_i(N, wr) \) and energy consumption \( \Delta E_i(N, wr) \) for each \( i \)-th application.

As opposed to the bus contention characterization process, which is a one-time activity carried out by the IP provider at Level A2, the bus contention prediction is made at the Level C by the SoC architect during design-space exploration. In addition, while the characterization process is time consuming since it relies on a dynamic estimation based on gate-level simulation, the prediction activity is very fast, since it applies well-defined equations, such as Eq. 7.4 and 7.3, on a code profiled during execution on the host and back-annotated with target-specific information. The following case study shows how efficient bus contention prediction can be made.

#### 7.3.1 Case study

This case study is meant to generalize Equation 7.4 so that it can still be applied to the case of an MPSoC with \( N \) processors running \( N \) different applications that can start and finish at any time.

An MPSoC execution time \( ex_{mpsoc} \) is defined as the range of time beginning when the first of the \( N \) processors starts its execution at time \( t_s \) and ending when the last processor finishes its execution at time \( t_f \). It is therefore true that \( ex_{mpsoc} = max(t_f) - min(t_s) \). In addition, for this case study the number of active processors \( n_a \) can vary in time, until it becomes 0 when the last application has completed its execution. Thus, it is true that \( 0 \leq n_a \leq N \). Since bus contention can only be caused by an active processor trying to access the bus, \( ex_i(N, wr) \) and \( E_i(N, wr) \) can be rewritten more precisely as \( ex_i(n_a, wr) \) and \( E_i(n_a, wr) \). Keeping track of the variation of \( n_a \) over time is critical. Note also that, when \( 0 \leq n_a \leq N \), there will be \( N - n_a \) processors in the idle state. This scenario is shown in Figure 7.5 where five different applications are considered running on five processors [\( P_1, ..., P_5 \)], and
7.3. PREDICTING BUS CONTENTION

where each i-th application starts and finishes at different times $t_{s,i}$ and $t_{f,i}$. The processors in idle state are represented by dotted lines.

![Diagram](image)

Figure 7.5: MPSoC execution time for $N$ processors and $N$ different applications

Using Equation 7.4 derived in the previous Section 7.2, the goal is to predict the value $ex_i(n_a, wr)$ for any possible application. The only unknown elements in the equation are $C_{wr,N,1}$ and $C_{wr,N,1}$, which indicate the percentage of cycles variation between the N-processor case and the 1-processor case, for instructions respectively non-related and related to store/load operations. The approach proposed in this section is to use the average values $\hat{C}_{wr,n_a,1}$ and $\hat{C}_{wr,n_a,1}$ (see Table 7.5) calculated on the characterization benchmarks, i.e. FFT, Fibonacci and Viterbi, and apply such averages to other applications as well for. Section 7.4 proves the generality of the model and that the error introduced is minimal. These average values are shown in Table 7.5. Equation 7.4 generalized to any i-th application becomes therefore:

$$
\Delta ex_i(n_a, wr) = T \cdot [\hat{C}_{wr,n_a,1} \cdot C_{wr,1} + \hat{C}_{wr,n_a,1} \cdot C_{i,wr}(1)]
$$

(7.5)

Table 7.5: Average $\hat{C}_{wr,n_a,1}$ and $\hat{C}_{wr,n_a,1}$ calculated on the FFT, Fibonacci and Viterbi benchmarks

<table>
<thead>
<tr>
<th>$wr$</th>
<th>$\hat{C}_{2,1}$</th>
<th>$\hat{C}_{3,1}$</th>
<th>$\hat{C}_{4,1}$</th>
<th>$\hat{C}_{5,1}$</th>
<th>$\hat{C}_{6,1}$</th>
<th>$\hat{C}_{7,1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.38</td>
<td>30.87</td>
<td>37.92</td>
<td>59.66</td>
<td>82.76</td>
<td>112.16</td>
<td></td>
</tr>
<tr>
<td>27.16</td>
<td>48.77</td>
<td>75.56</td>
<td>108.57</td>
<td>141.03</td>
<td>183.43</td>
<td></td>
</tr>
</tbody>
</table>

Since it also consumes energy, the idle state has to be taken into account. Energy consumption due to an idle state can be expressed as follows:

$$
E_{i, idle} = P_{idle} \cdot (ex_{mpsoc} - ex_i)
$$

(7.6)

where $P_{idle}$ is the power consumption of the processor when in idle state. The complete equations for execution time and energy consumption can thus be rewritten as $ex_i(n_a, wr) = ex_i(1, wr) + \Delta ex_i(n_a, wr)$ and $E_i(n_a, wr) = E_i(1, wr) + \Delta E_i(n_a, wr) + E_{i, idle}$ respectively.

Note that the starting time $t_{s,i}$ for each i-th application is a user-defined parameter. Although ignored in the present paper, normally there would be an operating
system taking care of releasing the different applications (or tasks) according to the user settings. The value of \( t_{f,i} \) is instead calculated by the prediction algorithm, as explained below. The times \( t_s \) and \( t_f \) are the only points where the \( n_a \) value is updated. The reason is that there is no other time when this value can change. The result is a static decomposition of the whole applications set into multiple time windows \( w_k \). In Figure 7.5, six time windows \([w_1, w_6]\) are identified.

Based on all these considerations, an algorithm can be used for predicting bus contention, where Equations 7.3, 7.5 and 7.6 introduced above are used iteratively to predict both execution time and energy consumption for each \( i \)-th application running on an \( N \)-processor SoC. Note that, since the values are updated only in correspondence to the beginning/end of a time window, prediction is very fast and its speed is proportional to the number of applications, not to their execution length. Specifically, the prediction algorithm consists of the following four main steps, which are performed whenever a new window \( w_k \) starts:

1) The value of \( n_a \) is updated.
2) The ending time of the time window is identified as the minimum time between all the \( t_{f,i} \) of the applications that are still running and the next closest starting time \( t_{s,i} \). The value of \( t_{f,i} \) is calculated as a function of the \( n_a \) value for the present window.
3) For each \( i \)-th application, the values of \( \Delta e_{x,i}(n_a, wr) \) and \( \Delta E_i(n_a, wr) \) are updated using Equations 7.5 and 7.3.
4) The process iterates until all applications have completed and \( n_a = 0 \). The value of \( E_{i, idle} \) can then also be updated using Equation 7.6, since \( ex_{mpsoc} \) is known.

### 7.4 Validating the Accuracy of Bus Contention Prediction

To validate the accuracy of Funtime in estimating bus contention penalty, gate-level measurements are used as a reference. The reason is that gate-level estimation is very accurate. The reference SoC architecture is the same as that used during the bus contention characterization phase, as reported in Figure 7.2. The 1-processor configuration is taken as a reference and measured values \( ex_i(1) \) and \( E_i(1) \) are used.

\( N \) different applications are considered, which run on \( N \) processors. Each \( i \)-th application is mapped onto only one processor. The case presented here runs on a 3-processor SoC. The three applications, which are different for the ones used during the characterization phase, are a Quick sort on \( P_1 \), a Dhrystone on \( P_2 \) and the N-queen problem on \( P_3 \). Since there is no OS running, all applications start at the same time, but can finish at different times.

Table 7.6 shows the results for measured versus predicted execution time and energy. It also shows the number of active processors \( n_a \) and the current state of each processor, i.e. active or idle. The predicted values for execution time and energy are calculated by using the algorithm described in Subsection 7.3.1. It is found that the prediction error is below 2%. In the energy prediction case, the idle energy is also considered in the error calculation.
Table 7.6: Measured vs. predicted execution time $e x_i [ms]$ and $E_i [\mu J]$. 3 different applications on a 3-processor SoC.

| $n_a$ | measured | $e x_i$ | $E_i$ | active | | | Predicted | $e x_i$ | $E_i$ | Predicted | $e x_i$ | $E_i$ | Predicted | $e x_i$ | $E_i$ | Predicted | $e x_i$ | $E_i$ | Predicted | $e x_i$ | $E_i$ | Predicted | $e x_i$ | $E_i$ |
|-------|---------|---------|-------|--------|---------|----------|---------|---------|--------|---------|---------|--------|---------|---------|--------|--------|---------|---------|--------|---------|---------|--------|---------|---------|--------|---------|
| 3     | measured | $98.89$ | $145.18$ | | | | predicted | $97.69$ | $143.42$ | | | | | | | | | | | | | | |
| 2     | measured | $248.87$ | $377.38$ | | | | predicted | $248.94$ | $377.51$ | | | | | | | | | | | | | | |
| 1     | measured | $463.29$ | $760.50$ | | | | predicted | $463.36$ | $760.65$ | | | | | | | | | | | | | | |

7.5 Summary

A high-level method has been presented for rapidly and accurately predicting bus contention effects on energy and performance in MPSoCs. The proposed approach consists of two steps: first, a one-time activity performed at Level A2, aimed at identifying and characterizing the main factors involved in bus contention; second, the development of a prediction mechanism, performed at Level C, to account for the occurrence of such factors for an arbitrary number of processors hosting arbitrary applications. As a result, a high prediction accuracy is achieved, which is within $\sim 2\%$ of the gate-level estimation.
Chapter 8

Level C – Refining the Operating System Overhead on Energy and Performance

This chapter continues with the description of the Refinement Level. A high-level method is here presented for rapidly and accurately estimating energy and performance overhead of Real-Time Operating Systems. Two main components are distinguished in the approach: first, an accurate one-time pre-characterization of the main RTOS functionalities in terms of energy and cycles; this activity is performed at Level A1 by the IP provider. Second, the development of an algorithm to rapidly predict the occurrences of such RTOS functionalities; this activity is performed at Level C. The approach is also validated against gate-level estimation, obtaining a worst-case error of 12%. The material presented in this chapter is based on the previous work published by the author in [58] and [57].

8.1 Introduction

The previous chapter has discussed the Refinement Level (Level C) by presenting a method for predicting the occurrences of bus contention in MPSoCs. The present chapter continues the description of the Refinement Level by proposing a method to account for the operating system overhead.

Real-Time Operating Systems (RTOS) are critical infra-structural components of embedded SoCs. They enable and manage sharing of hardware resources: computational, storage, interconnect and I/O. However, such services have an overhead, both in latency and energy. This overhead needs to be accounted for when doing design-space exploration and energy/performance estimation at system level, early in the design cycle. Without factoring in this overhead, the accuracy of the estimates would be severely compromised.

At the Level B of the Funtime approach, the simplified scenario has been consid-
Chapter 8. Level C – Refining the Operating System Overhead on Energy and Performance

C - Refinement level
- Transactions interdependency
- HW/SW optimizations (caches, etc.)
- Operating System
- Bus arbitration

Output
- Energy ($E_i$)
- Execution time ($t_{x_i}$)

Figure 8.1: The Refinement Level (Level C) – Refining the OS overhead

... where applications are independent and have all the system resources available to themselves. In that scenario, applications complete their execution, from start to end, without any interruption, for example by an OS. In reality however, it is often the case that an Operating System is present as a middleware between the hardware and the software component.

As discussed in Chapter 2, the impact of the OS is often evaluated through a high-level architectural simulation, which is often performed at transaction level. However, since the key idea of Funtime is to avoid architectural simulation, a different approach has been adopted. In particular, the refinement process to account for the OS impact, both in terms of energy and latency, consists of the following steps:

1. **RTOS characterization**: the main components of a generic RTOS are characterized in terms of typical number of execution cycles and energy consumption. This characterization is a one-time activity to be done by the IP provider at Level A1, as symbolically shown at the bottom of Figure 8.1. The OS is in fact considered as an IP. This is detailed in Section 8.2.

2. **RTOS activity prediction**: a static analysis strategy is proposed to predict how many occurrences of such OS components would be counted if the OS was really executed in a real use-case scenario. This prediction method, which is part of Level C and symbolically shown on the top of Figure 8.1, is invoked by the Funtime end-user for each use-case scenario. A detailed description of this step is presented in Section 8.3.

This refinement has been applied to both cases of Round Robin and Priority-driven schedulers. Unlike the former case, which only considers independent tasks not competing for shared resources, the latter one also accounts for interdependent...
tasks that do compete to lock a shared resource. The material presented in this chapter is based on the author’s previous work published in [58] and [57].

8.2 RTOS Characterization

This section proposes a general approach to characterizing the transactions – secondary transactions in this case – due to the RTOS overhead in terms of latency and energy. RTOS overheads are implied by its components that are called to implement the RTOS functionality. Characterization is a one-time activity done by the RTOS provider. The process consists in simulating the RTOS in a post-layout, back-annotated gate-level netlist of the representative SoC architecture. Although time consuming, this characterization process is justified because it is done only once and the precision gained is important for the accuracy of estimates at high-level. Although this chapter presents the characterization of the RTEMS RTOS [4] for a Leon-based SoC [1], the methods themselves are generic and do not depend on a specific RTOS implementation or SoC architecture.

The characterization process not only considers atomic RTOS calls, as in [11], but also considers coarse-grained RTOS calls like clock tick interrupts and scheduler invocation. The number and type of atomic RTOS calls involved both in the clock tick interrupt and in the scheduler are OS-dependent. However, thanks to the general approach that has been adopted, it is easy to specify the sequence of atomic OS routines that have to be included in the performance and energy characterization.

8.2.1 Characterizing a group of RTOS routines

The characterization process is explained in five steps using RTEMS for the Leon3-based SoC shown in Figure 8.2.

1. A representative SoC is composed, with essential hardware for hosting an RTOS. An example is given in Figure 8.2. The platform is then synthesized. Synthesis has been carried out in our case for the TSMC-90nm technology.

2. The RTOS and its tasks are compiled and loaded into the system. From the object file, a memory dump of the RTOS and the tasks is also taken. A
fragment of such a dump is shown in Figure 8.3. In the RTEMS case, this can be done using the `sparc-rtems-objdump` command. The dump allows any routine name to be associated with its memory addresses.

```
40006944 <rtems_clock_tick>:
40006944: 9d e3 bf 98  save  %sp, -104, %sp
40006948: 40 00 08 09  call  4000896c <_TOD_Tickle_ticks>
4000694c: 01 00 00 00  nop
40006950: 11 10 00 71  sethi  %hi(0x4001c400), %o0
40006954: 40 00 16 77  call  4000c330 <_Watchdog_Tickle>
```

Figure 8.3: Example of an objdump output

3. The system is executed at gate level, with the user-defined tasks, and the RTOS calls are activated a sufficient number of times – around 1000 in our case – to make the characterization statistically relevant.

4. As an output of step 3, an execution trace file and a VCD (Value Change Dump) file are produced from the gate-level simulation. It is required that the execution trace file contains both the address of each instruction that has been executed, as well as its time stamp. The VCD file contains instead the switching activity figures, used by the EDA tools to do power estimation.

5. By using the information in the dump, execution trace, VCD and by providing the sequence of atomic RTOS routines to be characterized, it is possible to extract the average number of cycles and also the energy for the routines under characterization. This last step has been automated by implementing and using a C-based script, which has been called RTOS Modeler and whose overall functionality is summarized in Figure 8.4. The advantage of having such a script is that it allows any atomic routine or sequence of routines present in the object dump file to be characterized, without needing any intervention from the user. As a consequence, the whole characterization process gains a significant speedup. In addition, this script is completely OS-independent; thus, it can be easily reused.

```
- objdump file
- exec. trace + VCD
- routines sequence

RTOS Modeler

IN

OUT

- avg. number of cycles
- power figures
```

Figure 8.4: RTOS Modeler
A summary of the coarse-grained RTOS calls that have been characterized for RTEMS is shown in Table 8.1. For each of them is shown the corresponding number of CPU instructions (secondary transactions), cycles and energy with the associated standard deviation $\sigma$. Note that the small value of $\sigma$ is an index of the characterization reliability. The energy values in the table only refer to the Leon3, configured without cache.

Table 8.1: Energy and performance characterization for RTEMS on Leon3 (no cache)

<table>
<thead>
<tr>
<th>RTOS calls</th>
<th># Leon3 instructions</th>
<th># Cycles</th>
<th>$\sigma$ Cycles</th>
<th>Energy [nJ]</th>
<th>$\sigma$ Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock tick interrupt</td>
<td>272</td>
<td>2241</td>
<td>24.00</td>
<td>80.14</td>
<td>0.96</td>
</tr>
<tr>
<td>scheduler + context switch</td>
<td>880</td>
<td>8434</td>
<td>30.01</td>
<td>263.58</td>
<td>0.96</td>
</tr>
<tr>
<td>scheduler without context switch</td>
<td>327</td>
<td>2545</td>
<td>3.53</td>
<td>89.94</td>
<td>0.20</td>
</tr>
<tr>
<td>idle task</td>
<td>3</td>
<td>22</td>
<td>0.04</td>
<td>0.76</td>
<td>0.00</td>
</tr>
<tr>
<td>msg q. broadcast to 3 + context switch</td>
<td>1510</td>
<td>12 263</td>
<td>6.94</td>
<td>434.21</td>
<td>7.77</td>
</tr>
<tr>
<td>msg q. send + context switch</td>
<td>834</td>
<td>6992</td>
<td>11.01</td>
<td>247.45</td>
<td>5.08</td>
</tr>
<tr>
<td>msg q. receive + context switch</td>
<td>830</td>
<td>6912</td>
<td>5.31</td>
<td>243.76</td>
<td>1.81</td>
</tr>
</tbody>
</table>

Table 8.1 only lists a subset of all the possible coarse-grained functionalities associated with an RTOS. This subset is sufficient to represent the case studies described in the next sections of this chapter. Characterizing and including more RTOS functionalities is, however, part of the future work, together with the discussion of other case studies.

8.3 RTOS Activity Prediction

Once the one-time RTOS characterization activity is done, it is still necessary to predict how many times the OS calls are triggered during the OS execution in order to estimate the total OS overhead. This section elaborates on how it is possible to implement an algorithm that performs such a prediction statically.

Some new quantities are introduced and categorized here, where possible, according to the Funtime abstraction level where their value is assigned. These quantities are in part shared and in part specific to the RR or priority-driven scheduling policy.

**User-defined parameters:** when using RR, all tasks run in turn for the same amount of time $t_{slice}$, which is a multiple of the clock tick period $t_{tick}$. For the priority-driven case, the notation $\Pi(T_i)$ is used to define the priority level assigned
to a generic task \( T_i \). For both scheduling policies, \( t_{rel,i} \) defines a task release time, i.e. the time when a task becomes available for execution. The notation \( l_{sync} \) indicates the source line number, within each task source code, where the task is supposed to have a synchronization point \( t_{sync} \), such as accessing a message queue, a mutex and so on.

From Level A: a set of quantities is defined corresponding to the execution time of some of the coarse-grain RTOS routines listed in Table 8.1. For example, \( ex_{sc} \) and \( ex_{ct} \) are the execution time of a scheduler call and of a clock tick interrupt respectively. The quantity \( ex_{del} \) is instead the execution time required to delete an ended task. An example is shown in Figure 8.8 – described in detail in Case Study 1 – where the red bars correspond to \( ex_{sc} \), the blue bars to \( ex_{ct} \) and the yellow bars to \( ex_{del} \). All these quantities are determined during the OS characterization phase, together with their respective energy consumption, as described in Section 8.2.

From Level B: the actual number of tasks \( T_i \) (applications) and all the required runtime information are provided at this level. Given a generic task \( T_i \), its execution time \( ex_i \) has already been defined as the ideal execution time required to complete \( T_i \) when \( T_i \) has all the resources available to itself. Note that this quantity is the output of the Level B of Funtime, as shown in Figure 3.1. This quantity is now redefined by splitting it into the following two sub-components:

1. \( ex_{tot,i} \): the total overall execution time of an \( i \)-th application.

2. \( ex_{inc,i}(line) \): the incremental advance in time as a function of the source-code line number. This quantity can be inferred by Funtime thanks to the enhancement proposed in Section 3.4.1.2. The availability of \( ex_{inc}(line) \) is a key element of the RTOS refinement step discussed in Section 8.3.2 since it allows us to predict whether one or more RTOS tasks will be blocked because of inter-task dependency or resource contention.

From Level C: In the RR case, \( SC_{slice} \) is defined as the number of scheduler calls during \( t_{slice} \). \( CT_{slice} \) is defined as the number of clock tick interrupts occurring during \( t_{slice} \). Thus, it is true that \( SC_{slice} = 1 \) and \( CT_{slice} = \frac{t_{slice}}{t_{tick}} \). Figure 8.5 illustrates this for the time range \( 30 \leq t < 60 \), where it is possible to identify one scheduler invocation in the red bar and three clock tick interrupts in the thinner blue bars.

\( ex_{slice} \) is also defined as the slice time that is actually dedicated to executing a generic task \( T_i \) and not spent in executing OS calls. This leads to \( ex_{slice} = t_{slice} - (SC_{slice} \cdot ex_{sc} + CT_{slice} \cdot ex_{ct}) \). In Figure 8.5 the calculation of the \( ex_{slice} \) value is shown for the time slice in time range \( 120 \leq t < 150 \).

For both scheduling policies, a set of counters is defined to count the occurrence of the RTOS routines involved in the analysis. For instance, \( SC \), \( CT \) and \( DEL \) refer to the total number of calls to the scheduler, to clock tick interrupts and
to the RTOS routines for tasks deletion when running an arbitrary number \( N \) of concurrent tasks.

Implementing a static algorithm to predict \( SC, CT, \) and, in general, the overall RTOS routines overhead, is the focus of this section. Using the quantities introduced above, it is detailed how this can be done. Due to the different features of the two scheduling policies, from now on the Round Robin and Priority-based schedulers will be treated separately, in Subsection 8.3.1 and 8.3.2 respectively.

### 8.3.1 Round Robin scheduler activity prediction

To ease the explanation, three successively more realistic case studies are used. First, it is assumed that all tasks are released simultaneously and have the same \( ex_i \). Second, tasks with different \( ex_i \) are allowed, but it is still assumed that they are released simultaneously. Finally, tasks released at different times are allowed, as well as the possibility that RTOS overheads may be a function of the number of ready tasks.

#### 8.3.1.1 Case study 1

In this ideal case, all the \( N \) tasks, with \( N=3 \), have the same release time \( t_{rel,i} = 0 \) and the same execution time \( ex_i = EX \) with \( EX = n \cdot ex_{slice}, n \in N \). The values for \( SC \) and \( CT \) can be calculated by Eq. 8.1. This scenario is exemplified in Figure 8.5, where numerical values for \( SC \) and \( CT \) are also extracted.

\[
SC = SC_{slice} \cdot N \cdot \frac{EX}{ex_{slice}}; \quad CT = CT_{slice} \cdot N \cdot \frac{EX}{ex_{slice}} \tag{8.1}
\]

Figure 8.5: RR scheduling: \( \forall T_i, t_{rel,i} = 0 \) and \( ex_i = EX \)

#### 8.3.1.2 Case study 2

A more general case is considered where all \( N \) tasks have different execution times \( ex_i \) and \( ex_i \) is not an exact multiple of \( t_{slice} \). In this case, each task \( T_i \) is completed before the end of its slice time. This scenario, shown in Figure 8.6 for \( N=4 \), is
highlighted in grey boxes. When this condition occurs, the scheduler is called at the end of the task and, unless the ready task queue is empty, a context switch occurs, even if the slice time has not yet expired. This is done to minimize the performance loss. Note that the behavior described is valid for RTEMS. In this case, a more general formula is needed to determine $SC$ and $CT$.

The quantities $Q_i$ and $R_i$ are defined as the quotient and the remainder of the division $ex_i/ex_{slice}$ for a generic task $T_i$. $R_i$ is the portion of $T_i$ for which the execution time is smaller than a slice time, that is $0 \leq R_i < ex_{slice}$. The value of $R_i$ is shown in Figure 8.6 for each of the 4 tasks considered in this example. Let then $SC_{rem,i}$ and $CT_{rem,i}$ be the number of calls to the scheduler and of clock ticks occurring during the time $R_i$. The values of $SC_{rem,i}$ and $CT_{rem,i}$ can be expressed as in Eq. 8.2 and 8.3 respectively.

$$SC_{rem,i} = \begin{cases} 0 & \text{if } R_i = 0 \\ 1 & \text{if } R_i > 0 \end{cases}$$ (8.2)

$$CT_{rem,i} = \begin{cases} 1 & \text{if } R_i \leq (t_{tick} - ex_{sc} - ex_{ct}) \\ 1 + \frac{R_i - (t_{tick} - ex_{sc} - ex_{ct})}{t_{tick} - ex_{ct}} & \text{if } R_i > (t_{tick} - ex_{sc} - ex_{ct}) \end{cases}$$ (8.3)

The total number of calls to the scheduler $SC$ and clock tick interrupts $CT$ is now calculated through an iterative process, where the number of iterations corresponds to the number of tasks $N$. During each iteration, the value of $SC$ and $CT$ is incremented. The main steps are shown in Algorithm 1 below.

**Algorithm 1** Assumptions: $t_{rel,i} = 0$, $ex_i \neq EX$ and $ex_i \neq ex_j$

```latex
SC, CT \leftarrow 0;
for i = 1 to N do
    SC \leftarrow SC + SC_{slice} \cdot Q_i + SC_{rem,i};
    CT \leftarrow CT + CT_{slice} \cdot Q_i + CT_{rem,i};
end for
```
8.3. RTOS ACTIVITY PREDICTION

8.3.1.3 Case study 3

A more real case is considered where $N$ tasks have arbitrary release times $t_{rel,i}$ and execution times $ex_i$, as shown in Figure 8.7. In addition, it is assumed that the execution time of some OS functionalities is related to the number of ready tasks $rdy(t)$ at the time $t$ when such functionalities are triggered. Thus, in this case $ex_{sc}$ and $ex_{ct}$ are not constant figures any more, but they depend on $rdy(t)$ and can be denoted as $ex_{sc}(rdy(t))$ and $ex_{ct}(rdy(t))$ respectively.

Figure 8.7: Round Robin scheduling: $\forall T_i, t_{rel,i} \neq 0, ex_i \neq EX, ex_i \neq ex_j, ex_{sc}(rdy(t))$ and $ex_{ct}(rdy(t))$

When calculating the value of $SC$ and $CT$, it is therefore also necessary to determine the value of $rdy(t)$. This process can be eased by restricting the evaluation of $rdy(t)$ to specific time points, which correspond either to a task release time $t_{rel,i}$ or completion time $t_{end,i}$. The reason is that the value of $rdy(t)$ remains unchanged for all the other values of $t$. The result is a static decomposition of the whole scheduling into multiple time windows $w_i$. This is shown in Figure 8.7, where five time windows $[w_1,w_5]$ are identified for three tasks and highlighted by different-color areas. In detail, whenever a new time window starts and $rdy(t)$ is calculated, the ending time of the same time window is identified as the minimum time between the $t_{end,i}$ for each ready task and the next closest release time $t_{rel}$. Once the time window has been defined, the values of $SC$ and $CT$ for that time window are calculated and added to the old ones calculated in the previous window. This process is iterated until all tasks are completed. Note that, even in this more elaborated implementation, the proposed OS prediction algorithm still has an execution time proportional to the number of tasks involved, not to their execution time $ex_i$. Thus, it is still extremely fast. The pseudo-code for the main steps described above is reported in Algorithm 2.

8.3.2 Prediction of Priority-Driven Scheduler Activity

Also in this case, to ease the explanation, two successively more realistic case studies are used. First, it is assumed that all independent tasks not competing for shared resources, but with different release times for the tasks, are allowed, as well as the possibility that RTOS overheads may be a function of the number of ready tasks. Second, the same assumptions are kept as for the first case study, except that interdependent tasks which can compete for shared resources are also allowed. For a
successful OS prediction in this second case study, the importance is demonstrated of the extension to the GCC compiler described in Section 3.4.1.2. Note that, although in this chapter RTEMS is taken as the reference OS, the methodology is general and can be adapted to any OS.

8.3.2.1 Case study 1

This case study considers the case where \( N \) independent tasks with priority \( \Pi(T_1) > \Pi(T_i) > \Pi(T_N) \) have arbitrary release times \( t_{rel,i} \) and where the execution time of some OS functionalities is dependent on the number of ready tasks \( rdy(t) \) at the time \( t \) when such functionalities are triggered. For this reason, \( ex_{sc}, ex_{ct} \) and \( ex_{del} \) are not constant figures, but depend on \( rdy(t) \) and can be denoted as \( ex_{sc}(rdy(t)) \), \( ex_{ct}(rdy(t)) \) and \( ex_{del}(rdy(t)) \) respectively. The availability of the \( rdy(t) \) value also allows us to infer the possible occurrence of the OS idle task, a condition that is true for \( rdy(t) = 0 \). For this case study, only the component \( ex_{tot,i} \) of the quantity \( ex_i \) is needed.

Figure 8.8 is used as a reference during the algorithm description. Three independent tasks are shown, \( T_1 \) and \( T_3 \) being the highest and lowest priority task respectively. At the figure bottom, the \( t_{rel,i} \) and \( t_{end,i} \) value associated with each task is indicated. Note that, while the value of \( t_{rel,i} \) is user-assigned, the value of \( t_{end,i} \) is calculated at run-time by the prediction algorithm. In addition, although \( t_{rel,i} \) could in principle be assigned any time value, this value is seen and evaluated only at the occurrence of the next closest clock tick, when the scheduler is invoked. For this reason, it is assumed that either the Funtime user or a Funtime routine

\[
\text{Algorithm 2}\ t_{rel,i} \neq 0, \ ex_i \neq ex_j, \ ex_{sc}(rdy(t)), \ ex_{ct}(rdy(t))
\]

\[
SC(rdy(t)) = CT(rdy(t)) = 0;
\]

\[
\text{total execution time left } e_{tot_left} = \sum_{i=1}^{N} ex_i;
\]

\[
time t \leftarrow 0;
\]

\[
\text{while } e_{tot_left} > 0 \text{ do}
\]

\[
\text{for } i = 1 \text{ to } N \text{ do}
\]

\[
rdy(t) \leftarrow \text{number of ready tasks at time } t;
\]

\[
T_{rdy}[i] \leftarrow \text{which tasks are ready at time } t;
\]

\[
\text{end for}
\]

\[
\text{for each } T_{rdy}[i] \text{ at time } t \text{ do}
\]

\[
calculate its ending time } t_{end,i};
\]

\[
\text{end for}
\]

\[
t_{end\_min} \leftarrow \text{min between each } t_{end,i} \text{ and next closest } t_{rel};
\]

\[
\text{update } SC(t_{end\_min} - t, rdy(t));
\]

\[
\text{update } CT(t_{end\_min} - t, rdy(t));
\]

\[
\text{update } e_{tot\_left};
\]

\[
t \leftarrow t_{end\_min}
\]

\[
\text{end while}
\]
can take care of always setting a task $t_{rel,i}$ at a time which is a multiple of $t_{tick}$.
The value of $t_{end,i}$ can instead occur at any time, since the scheduler is called by
the OS routine that deletes the ended task. This is represented by the yellow bars in Figure 8.8.

![Figure 8.8: 3 independent tasks $T_1$, $T_2$ and $T_3$. $t_{rel,i} \neq t_{rel,j}$](image)

The value of $t_{rel,i}$ and $t_{end,i}$ is of great importance for the whole prediction
algorithm, since evaluation is restricted to only these specific times. The reason
is that there is no other time when a variation in the tasks state or in the tasks
ready queue can be encountered. The result is a static decomposition of the whole
scheduling into multiple time windows $w_i$. This is shown in Figure 8.8 where five
time windows $[w_1, w_5]$ are identified for three tasks and highlighted by different-
color areas.

In detail, whenever a new time window starts, the following operations are
performed.

1. The number of ready tasks $rdy(t)$ is calculated. If $rdy(t) = 0$, the OS Idle
task is expected to run for the next time window $w_i$. This case is shown in
Figure 8.8 at time $t = 42$.

2. The ready tasks are sorted according to their priority. The highest-priority
task $T_H$ is expected to run for the next time window $w_i$. For instance, note
that, as shown in Figure 8.8 at time $t = 110$, task $T_3$ is preempted by task
$T_1$, which has higher priority.

3. The ending time of a time window is identified as the minimum time between
the $t_{end,i}$ of the task running in that window and the next closest release time
$t_{rel,i}$.

4. The values of $SC$, $CT$ and $DEL$ are updated with the related contributions
given in the present window. Such contributions are defined as $SC_w$, $CT_w$
and $DEL_w$.

5. This process is iterated until all tasks are completed.
CHAPTER 8. LEVEL C – REFINING THE OPERATING SYSTEM
OVERHEAD ON ENERGY AND PERFORMANCE

The value of $t_{end,i}$, within each time window, is determined through the following considerations. First, the initial OS overhead for the window being estimated is taken into account and the current time $t_{current}$ is updated. For instance, when a new window starts after another task has ended, the OS routine for task deletion is always executed (yellow bar), followed by a call to the scheduler (red bar). Thus, if the beginning of window $w_5$ in Figure 8.8 is taken as an example, it is true that $ex_{del} + ex_{sc} = 2 + 2 = 4 \Rightarrow t_{current} = 133 + 4 = 137$. Second, the time difference is calculated between the next closest clock tick and the current time, the current time is incremented of a quantity equal to the calculated time difference and that difference is subtracted from the total task execution time $ex_{tot,i}$. This is done to end up exactly at the time when a clock tick occurs and thus ease the calculation of the number of total clock ticks happening until the end of the task. Referring again to Figure 8.8 this step takes us from an initial $t_{current} = 137$ to a final $t_{current} = 140$. Third, the number of clock tick interrupts for the present window is calculated. This quantity has been previously defined as $CT_w$. It is true that $CT_w = \lceil \frac{time\_left_i}{time\_tick} \rceil - ex_{ct}$. For the example considering $w_5$, it is true that $CT_w = \lceil \frac{38}{10} \rceil - 1 = 5$. Finally, $t_{end,i}$ can be calculated as $t_{end,i} = t_{current} + time\_left_i + ex_{ct} \cdot CT_w$. In our case, $t_{end,3} = 140 + 38 + 1 \cdot 5 = 183$, which is correct.

Note that, although the proposed OS behavior prediction algorithm exhibits an iterative behavior, it still has an execution time proportional to the number of windows defined by the algorithm, and not to the tasks execution time $ex_{tot,i}$. Thus, it is still fast and clearly faster than running the actual OS.

8.3.2.2 Case study 2

In this realistic case study, all the considerations made in Case study 1 are kept valid, but interdependent tasks that can compete for shared resources are also allowed. Unlike the previous case study, here the $ex_{inc,i}(line)$ component of the quantity $ex_i$ is also needed.

By interdependent task is meant a task that, at some point of its execution, reaches a synchronization point $t_{sync}$ where it needs to wait for data from another task before being able to continue. As an example, consider Figure 8.9 at time $t = 33$, the highest priority task $T_1$ attempts to receive data from a message queue, but fails and becomes blocked, since the queue is empty. At time $t = 83$, $T_2$ successfully sends some data into the same message queue and then is preempted by $T_1$, which has now been unblocked and can thus resume its execution.

A shared resource could be a memory, a mutex or, in general, a serially-accessible resource. A conflict occurs when multiple tasks try to access the same resource at the same time. In general, the part of code dealing with accessing a resource is defined as a critical section and normally it cannot be preempted, not even by a higher-priority task. This case is visible in Figure 8.9 at time $t = 132$, the lowest-priority task $T_4$ successfully locks a mutex and enters its critical section. The access to a shared resource $R$ is shown as a white-filled rectangle. At time $t = 150$, $T_4$ is preempted by $T_3$ and, in turn, at time $t = 180$, $T_3$ is preempted by $T_2$. Then
8.3. RTOS ACTIVITY PREDICTION

at time $t = 213$, $T_2$ also tries to access resource $R$ by locking the same mutex previously locked by $T_4$. However, the locking attempt fails and $T_2$ is blocked until $T_4$ releases the mutex at time $t = 265$.

This scenario is known as priority inversion and generally refers to the case of a high-priority task being blocked by a lower-priority task. Some techniques have been developed to avoid this problem. Two of them are known as priority inheritance and priority ceiling. Observe that the goal of this chapter is not to find the best scheduling policy for a given scenario, but rather to show how, provided there is a priority-driven scheduling for a given scenario, Funtime can predict the OS components overhead. Nevertheless, compliance to the priority inheritance and priority ceiling techniques can be enabled in Funtime too, as part of the priority prediction algorithm. The user of Funtime will choose which policy to use.

This case study accounts for the implications of having interdependent tasks that can compete for shared resources, since these factors can heavily affect the amount of OS overhead, i.e. number of context switches, occurrence and duration of the OS Idle task, etc. For example, it is only by knowing at which point in time $T_1$ tries to read from its message queue that it is possible to estimate the duration of the OS Idle task. As another example, by knowing when $T_4$ tries to lock its mutex, it can be predicted whether $T_4$ will be able to get the lock before $T_3$ is released or not. If it does, then this is the scenario shown in Figure 8.9 and described above. If not, $T_4$ will be able to lock its mutex only after $T_2$ and $T_3$ have finished. In this latter case, $T_2$ would not be blocked and therefore two context switches less would be counted.

However, to be able to predict the time point $t_{sync}$ when a task tries to access an OS resource, like a message queue or a mutex (see the top of Figure 8.9), the only knowledge of the total task execution time $ex_{tot,i}$ is no longer enough. Instead, the following two conditions must also be satisfied:

1. It must be possible to evaluate the advance of time with the granularity of a single source-code line. This quantity has been defined as $ex_{inc,i}(line)$ and can be retrieved by instrumenting the compiler as described in Section 3.4.1.2.

2. The Funtime user must specify at what source-code line $l_{sync}$ the synchronization point occurs and what dependency relation exists. This information
is provided as part of the UCS.

From the two points above, it gives that \( t_{\text{sync}} = e_{\text{inc}}(t_{\text{sync}}) \).

As in Case study 1, the proposed OS prediction algorithm relies on taking decisions at specific time points, which correspond to the end of temporal windows \( w_i \). The only difference is that not only \( t_{\text{rel},i} \) and \( t_{\text{end},i} \) are used to identify a window edges, but also the time points where a task reaches a synchronization point. This is shown in Figure 8.9 where 13 windows are identified.

In the case of interdependent tasks, in order to run an application on the development host at Level B (Algorithmic Level), it is necessary that the Funtime user manually provides a value to the variables that, at Level C (Refinement Level), would be assigned at a synchronization point. It is emphasized that Level B is the only level where an application is run in Funtime (see Figure 3.1). In case that an application takes different branches depending on the value received, the Funtime user chooses the values according to the specific needs, so to have a trace of primary transactions available for the different possibilities. Later on, he/she can also specify a percentage value corresponding to the frequency each branch is taken. Note instead that, at Level C, no application is run at all, as well as no Operating System.

Algorithm 3 reports the pseudo-code for the main steps described in Case study 1 above. The text in red color instead distinguishes the extra component needed to account for Case study 2.

Algorithm 3
/* Initialize all the OS quantities to zero */
\( SC(\text{rdy}(t)) = CT(\text{rdy}(t)) = \ldots = DEL(\text{rdy}(t)) \leftarrow 0; \)
total execution time left \( e_{\text{tot_left}} \leftarrow \sum_{i=1}^{N} ex_i; \)
current time \( t_{\text{current}} \leftarrow 0; \)
while \( e_{\text{tot_left}} > 0 \) do
  for \( i = 1 \) to \( N \) do
    \( \text{rdy}(t) \leftarrow \) number of ready tasks at time \( t; \)
    \( T_{H}(t) \leftarrow \) highest priority task ready at time \( t; \)
  end for
  calculate \( T_{H} \) ending time \( t_{\text{end},T_{H}}; \)
  \( t_{\text{next}} \leftarrow \min(t_{\text{end},T_{H}}, \text{closest } t_{\text{rel}}, \text{closest } t_{\text{sync}}); \)
  /* Update all the OS quantities */
  update \( SC(t_{\text{next}} - t_{\text{current}}, \text{rdy}(t)); \)
  update \( CT(t_{\text{next}} - t_{\text{current}}, \text{rdy}(t)); \)
  \( \vdots \)
  update \( DEL(t_{\text{next}} - t_{\text{current}}, \text{rdy}(t)); \)
  update \( e_{\text{tot_left}}; \)
  \( t_{\text{current}} \leftarrow t_{\text{next}} \)
end while
8.4 Validation of Refinement Accuracy

Validation of the refinement presented in this chapter has been carried out both for the Round Robin and Priority-driven scheduler. The accuracy of the refinement estimation, for both the OS energy and latency, has been validated against gate-level estimation. The reason is two-fold: first, gate level is very accurate; second, IP-level energy models and the OS refinements were achieved based on gate-level characterization.

8.4.1 Round Robin scheduler

The SoC platform used for validation is the same as that used for the OS characterization and shown in Figure 8.2. The goal is to verify the accuracy of Funtime in predicting energy and execution time for a software environment where two tasks \( T_1 \) and \( T_2 \) run on top of the RTEMS OS and are scheduled using Round Robin. The values of the clock tick period \( t_{\text{tick}} \) and of the time slice period \( t_{\text{slice}} \) have been set to 1ms and 3ms respectively.

\( T_1 \) and \( T_2 \) are synthetic applications that have been chosen on purpose with different execution times \( e_{x1} \neq e_{x2} \), but with the same release times \( t_{\text{rel,1}} = t_{\text{rel,2}} = 0 \). Such applications are different from those used during the OS characterization.

The results have been collected in Table 8.2, which has been drawn on two lines due to the lack of space. The table is itself split into three horizontal subtables considering an increasing number of sources of inaccuracy. Subtable 1 only considers the error deriving from the OS characterization described in Section 8.2, while \( e_{x1} \) and \( e_{x2} \), as well as the OS activity are measured from gate level. Subtable 2 has two sources of inaccuracy: the OS characterization and the OS activity prediction, described in Section 8.2 and 8.3.1 respectively, while \( e_{x1} \) and \( e_{x2} \) are measured. Finally, Subtable 3 considers three sources of inaccuracy: the first two are the same as for Subtable 2, while the third comes from using Funtime (Level A and B) to also derive \( e_{x1} \) and \( e_{x2} \). Note that the top part of the table reports figures related to the two tasks \( T_1 \) and \( T_2 \), independently of the OS; the bottom part is instead meant to show the OS overhead. The energy values shown refer to the Leon3 processor.

The RTEMS functionalities that are accounted for in Table 8.2 are, from left to right, the clock tick interrupt, the invocation to the scheduler resulting in a context switch and the invocation to the scheduler not resulting in a context switch. For each such functionality, a comparison is made between energy and time values extracted from a gate-level simulation and those inferred by Funtime. In any of the three subtables, the results show that the Funtime accuracy increases with the number of OS system calls. For example, the accuracy achieved for the clock tick interrupt, which occurs 108 times, is much higher than that achieved for the scheduler invocation without context switch, which occurs only eight times. From the OS activity prediction perspective, it is straightforward that a 1-unit error weighs more on a total of 8 units rather than 108. However, it is emphasized that the Funtime methodology is meant to be used on extensive use-case scenarios, where
Table 8.2: Validating Funtime vs. gate level for energy estimation: 2 RR-scheduled tasks of different length

<table>
<thead>
<tr>
<th>Applications</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>Real $ex_i$ [ms]</td>
<td>37.60</td>
<td>59.35</td>
</tr>
<tr>
<td>Real energy $[\mu J]$</td>
<td>53.24</td>
<td>84.73</td>
</tr>
</tbody>
</table>

**Subtable 1 (1 source of inaccuracy: OS characterization)**

<table>
<thead>
<tr>
<th>Applications</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>Real $ex_i$ [ms]</td>
<td>37.60</td>
<td>59.35</td>
</tr>
<tr>
<td>Real energy $[\mu J]$</td>
<td>53.24</td>
<td>84.73</td>
</tr>
</tbody>
</table>

**Subtable 2 (2 sources of inaccuracy: OS characterization + OS activity prediction)**

<table>
<thead>
<tr>
<th>Applications</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>Real $ex_i$ [ms]</td>
<td>37.60</td>
<td>59.35</td>
</tr>
<tr>
<td>Inferred $ex_i$ [ms]</td>
<td>36.25</td>
<td>58.00</td>
</tr>
<tr>
<td>Error [%]</td>
<td>-3.59</td>
<td>-2.28</td>
</tr>
<tr>
<td>Real energy $[\mu J]$</td>
<td>53.24</td>
<td>84.73</td>
</tr>
<tr>
<td>Inferred energy $[\mu J]$</td>
<td>46.73</td>
<td>74.77</td>
</tr>
<tr>
<td>Error [%]</td>
<td>-12.23</td>
<td>-11.76</td>
</tr>
</tbody>
</table>

**Subtable 3 (3 sources of inaccuracy: OS characterization + OS activity prediction + $ex_i$ value)**

<table>
<thead>
<tr>
<th>Applications</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>Real $ex_i$ [ms]</td>
<td>37.60</td>
<td>59.35</td>
</tr>
<tr>
<td>Inferred $ex_i$ [ms]</td>
<td>36.25</td>
<td>58.00</td>
</tr>
<tr>
<td>Error [%]</td>
<td>-3.59</td>
<td>-2.28</td>
</tr>
<tr>
<td>Real energy $[\mu J]$</td>
<td>53.24</td>
<td>84.73</td>
</tr>
<tr>
<td>Inferred energy $[\mu J]$</td>
<td>46.73</td>
<td>74.77</td>
</tr>
<tr>
<td>Error [%]</td>
<td>-12.23</td>
<td>-11.76</td>
</tr>
</tbody>
</table>

**RTEMS OS**

**Subtable 1 (1 source of inaccuracy: OS characterization)**

<table>
<thead>
<tr>
<th>Applications</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>clock ticks</td>
<td>sched. + c.s.</td>
<td>scheduler</td>
</tr>
<tr>
<td>Real #OS macro-func.</td>
<td>108</td>
<td>25</td>
<td>8</td>
</tr>
<tr>
<td>Inferred #OS macro-func.</td>
<td>110</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+2</td>
<td>+1</td>
<td>8</td>
</tr>
<tr>
<td>Real energy $[\mu J]$, time [ms]</td>
<td>8.62</td>
<td>6.05</td>
<td>7.47</td>
</tr>
<tr>
<td>Inferred energy $[\mu J]$, time [ms]</td>
<td>8.65</td>
<td>6.05</td>
<td>7.38</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+0.35</td>
<td>-0.14</td>
<td>-1.27</td>
</tr>
</tbody>
</table>

**Subtable 2 (2 sources of inaccuracy: OS characterization + OS activity prediction)**

<table>
<thead>
<tr>
<th>Applications</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>clock ticks</td>
<td>sched. + c.s.</td>
<td>scheduler</td>
</tr>
<tr>
<td>Real #OS macro-func.</td>
<td>108</td>
<td>25</td>
<td>8</td>
</tr>
<tr>
<td>Inferred #OS macro-func.</td>
<td>110</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+2</td>
<td>+1</td>
<td>8</td>
</tr>
<tr>
<td>Real energy $[\mu J]$, time [ms]</td>
<td>8.62</td>
<td>6.05</td>
<td>7.47</td>
</tr>
<tr>
<td>Inferred energy $[\mu J]$, time [ms]</td>
<td>8.81</td>
<td>6.16</td>
<td>7.64</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+2.20</td>
<td>+1.80</td>
<td>+2.26</td>
</tr>
</tbody>
</table>

**Subtable 3 (3 sources of inaccuracy: OS characterization + OS activity prediction + $ex_i$ value)**

<table>
<thead>
<tr>
<th>Applications</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>clock ticks</td>
<td>sched. + c.s.</td>
<td>scheduler</td>
</tr>
<tr>
<td>Real #OS macro-func.</td>
<td>108</td>
<td>25</td>
<td>8</td>
</tr>
<tr>
<td>Inferred #OS macro-func.</td>
<td>109</td>
<td>28</td>
<td>8</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+1</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Real energy $[\mu J]$, time [ms]</td>
<td>8.62</td>
<td>6.05</td>
<td>7.47</td>
</tr>
<tr>
<td>Inferred energy $[\mu J]$, time [ms]</td>
<td>8.73</td>
<td>6.10</td>
<td>7.38</td>
</tr>
<tr>
<td>Error [%]</td>
<td>+1.28</td>
<td>+0.83</td>
<td>-1.27</td>
</tr>
</tbody>
</table>

billions of transactions take place. Under these assumptions, the 18% energy error and the 16% time error obtained with respect to the scheduler invocation without context switch become insignificant.

### 8.4.2 Priority-driven scheduler

The SoC platform used for validation is the same as that used for characterization and is shown in Figure 8.2. Communication is implemented by the AMBA AHB/APB bus, while computation relies on the SPARC-based Leon3 processor, configured without cache. The entire system runs at the frequency of 40 MHz.

The goal is to verify the Funtime prediction accuracy for an RTEMS-based signal-processing software composed of five interdependent tasks, as shown in Figure 8.10. Task $T_1$ provides a data stream of 100 elements. Such a stream is broadcast to $T_2$, $T_3$ and $T_4$ through the message queue $Q_1$. Always, only one of the three tasks $T_2$, $T_3$ and $T_4$ performs some processing on the received data, depending on
8.4. VALIDATION OF REFINEMENT ACCURACY

Table 8.3: Validating Funtime vs. gate level for energy estimation: priority-driven scheduling

<table>
<thead>
<tr>
<th>Applications</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real (e_x) [ms]</td>
<td>0.3</td>
<td>11.3</td>
<td>5.1</td>
<td>14.0</td>
<td>9.8</td>
</tr>
<tr>
<td>Real (e_{\mu J})</td>
<td>0.5</td>
<td>15.8</td>
<td>7.1</td>
<td>19.5</td>
<td>13.7</td>
</tr>
</tbody>
</table>

**Subtable 1 (1 source of inaccuracy: OS characterization)**

<table>
<thead>
<tr>
<th>Real (e_x) [ms]</th>
<th>0.3</th>
<th>11.3</th>
<th>5.1</th>
<th>14.0</th>
<th>9.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real (e_{\mu J})</td>
<td>0.5</td>
<td>15.8</td>
<td>7.1</td>
<td>19.5</td>
<td>13.7</td>
</tr>
<tr>
<td>Error [%]</td>
<td>-5.4</td>
<td>-13.0</td>
<td>-1.2</td>
<td>-8.5</td>
<td>-16.3</td>
</tr>
</tbody>
</table>

**Subtable 2 (2 sources of inaccuracy: OS characterization + OS activity prediction)**

<table>
<thead>
<tr>
<th>Real (e_x) [ms]</th>
<th>0.5</th>
<th>11.3</th>
<th>5.1</th>
<th>14.0</th>
<th>9.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real (e_{\mu J})</td>
<td>0.5</td>
<td>15.8</td>
<td>7.1</td>
<td>19.5</td>
<td>13.7</td>
</tr>
<tr>
<td>Inferred (e_x) [ms]</td>
<td>0.3</td>
<td>9.8</td>
<td>5.0</td>
<td>12.8</td>
<td>8.1</td>
</tr>
<tr>
<td>Inferred (e_{\mu J})</td>
<td>0.5</td>
<td>14.8</td>
<td>7.6</td>
<td>19.3</td>
<td>12.2</td>
</tr>
<tr>
<td>Error [%]</td>
<td>-6.3</td>
<td>-10.5</td>
<td>-1.4</td>
<td>-10.5</td>
<td>-6.3</td>
</tr>
</tbody>
</table>

**Subtable 3 (3 sources of inaccuracy: OS characterization + OS activity prediction + \(e_x\) value)**

<table>
<thead>
<tr>
<th>Real (e_x) [ms]</th>
<th>0.5</th>
<th>11.3</th>
<th>5.1</th>
<th>14.0</th>
<th>9.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real (e_{\mu J})</td>
<td>0.5</td>
<td>15.8</td>
<td>7.1</td>
<td>19.5</td>
<td>13.7</td>
</tr>
<tr>
<td>Inferred (e_x) [ms]</td>
<td>-6.3</td>
<td>-10.5</td>
<td>-1.4</td>
<td>-10.5</td>
<td>-6.3</td>
</tr>
<tr>
<td>Inferred (e_{\mu J})</td>
<td>-6.3</td>
<td>-10.5</td>
<td>-1.4</td>
<td>-10.5</td>
<td>-6.3</td>
</tr>
<tr>
<td>Error [%]</td>
<td>-5.4</td>
<td>-13.0</td>
<td>-1.2</td>
<td>-8.5</td>
<td>-16.3</td>
</tr>
</tbody>
</table>

**RTEMS OS**

<table>
<thead>
<tr>
<th>clock ticks</th>
<th>msg q. broadcast(3)</th>
<th>msg q. send</th>
<th>msg q. receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real #OS macro-func.</td>
<td>1174</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Inferred #OS macro-func.</td>
<td>100</td>
<td>1263</td>
<td>100</td>
</tr>
<tr>
<td>Error [#]</td>
<td>+89</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Subtable 2 (2 sources of inaccuracy: OS characterization + OS activity prediction)**

<table>
<thead>
<tr>
<th>Real #OS macro-func.</th>
<th>1174</th>
<th>100</th>
<th>100</th>
<th>400</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inferred #OS macro-func.</td>
<td>100</td>
<td>1263</td>
<td>100</td>
<td>400</td>
</tr>
<tr>
<td>Error [#]</td>
<td>+89</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Subtable 3 (3 sources of inaccuracy: OS characterization + OS activity prediction + \(e_x\) value)**

<table>
<thead>
<tr>
<th>Real #OS macro-func.</th>
<th>1174</th>
<th>100</th>
<th>100</th>
<th>400</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inferred #OS macro-func.</td>
<td>100</td>
<td>1263</td>
<td>100</td>
<td>400</td>
</tr>
<tr>
<td>Error [#]</td>
<td>+89</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

the actual data value. The processed data is then sent into the message queue \(Q_2\) and reaches the output process \(T_5\). This modeling flow could be, for instance, representative of an equalizer.

Figure 8.10: RTEMS-based signal-processing software

The RTEMS clock tick period \(t_{\text{tick}}\) has been set to 200\(\mu s\). The task priority has been assigned as follows: \(\Pi(T_5) > \Pi(T_2) > \Pi(T_3) > \Pi(T_4) > \Pi(T_1)\). This means that \(T_5\) and \(T_1\) have the highest and lowest priority respectively. This is done since the system is expected to be immediately ready to operate whenever
the input data stream arrives. In this case, the stream arrives when $T_1$ is released, i.e. at $t_{rel,1} = 40\text{ms}$, which is an integer multiple of $t_{tick}$. The release time for the remaining tasks is instead $t_{rel,2...5} = 0\text{ms}$.

The results have been collected in Table 8.3, which has been drawn on two lines due to the lack of space. The table is itself split into three horizontal subtables considering successively more sources of inaccuracies, moving from top to bottom. Subtable 1 only considers the error deriving from the OS characterization summarized in Section 8.2 while $[ex_1, ..., ex_5]$, as well as the OS activities are measured from gate level. Subtable 2 has two sources of inaccuracies: the OS characterization and the OS activity prediction, described in Section 8.2 and 8.3.2 respectively, while $[ex_1, ..., ex_5]$ are measured. Finally, Subtable 3 considers three sources of inaccuracy: the first two are the same as for Subtable 2, while the third comes from using Funtime (Level A and B) to derive also $[ex_1, ..., ex_5]$. Note that the top part of the table reports figures related to the five tasks $[T_1, ..., T_5]$, independently of the OS; the bottom part is instead meant to show the OS overhead. The energy values shown refer to the Leon3 processor. Both real and estimated values are shown along with the percentage error.

The RTEMS functionalities that are accounted for in Table 8.3 are, from left to right, the clock tick interrupt, the message queue broadcast (to three queues), the message queue send and the message queue receive. The inferred number of clock ticks is higher than the real one. The reason is that the real value of $t_{tick}$ was measured to be on average $\sim218\mu\text{s}$ instead of the ideal $200\mu\text{s}$. The inferred number of the remaining RTOS routines is instead equal to the real one. The reason is that their occurrence is directly visible from the inferred execution trace, and is independent of the real or inferred task execution time $ex_i$. In general, good accuracy figures are obtained both for energy and timing estimation, within the 12% of the real value.

### 8.5 Summary

As part of the Funtime Refinement Level, this chapter has presented a high-level method for rapidly and accurately estimating energy and performance overhead of Real-Time Operating Systems, for both Round Robin and priority-based scheduling. Two main components have been identified in the approach: first, a one-time pre-characterization of the main RTOS functionalities in terms of energy and cycles; this activity is part of Level A1. Second, the development of an algorithm to predict the occurrences of such RTOS functionalities; this activity is part of Level C. As a result, a good prediction accuracy is achieved, since only 12% of the gate-level accuracy is lost when doing energy and performance estimation.
Chapter 9

Conclusions and Future Work

This chapter summarizes the key features of Funtime that have been discussed in this thesis work and proposes extensions to the present Funtime framework as part of the future work.

9.1 Conclusions

The present work has described in detail a system-level framework for energy and performance estimation, which has been called Funtime. The need of having efficient system-level estimation (SLE) tools is justified by the increasing broadness of the design space at the system level, which makes it very difficult to take the best architectural decisions. Decisions taken at the system level, early in the design cycle, are however essential in affecting the quality of the final design. Taking such decisions manually, based on the experience matured from previous designs and on rules of thumb, as it has been done so far, is not a feasible approach anymore. Efficient SLE tools are therefore necessary to automate the activities performed at system level. Specifically, such tools can be used to perform design-space exploration and as a basis for system-level synthesis.

Several of today’s approaches to system-level estimation and design in general, such as Transaction-Level Modeling, still rely on the architectural simulation of the target. This implies an extra engineering effort, since a new executable specification of the target has to be written. Besides, although at a high abstraction level, architectural simulation may still be too slow when modeling large use-case scenarios. Unlike these approaches, Funtime does not rely on a high-level simulation of the target to carry out estimation. Instead, its estimation activity is based on the following three layers.

First, a characterization layer – the IP Level (Level A) – which creates IP energy and performance models. These are not executable models, since they come in the form of either look-up tables or analytical expressions. Although time-consuming, since performed on a back-annotated gate-level netlist and for each IP transaction,
this step still makes sense since it is a one-time activity performed by the IP provider and it gives high accuracy to the whole estimation tool. A case study has been presented in Chapter 4, where one such IP model has been implemented for a Leon3 processor. The accuracy of the model has then been validated against gate-level estimation and has shown to be within 15%. Level A also models the impact that the external communication link, connecting different IPs, has on individual IP transactions. Chapter 5 presents a case study about switching from a bus-based SoC to a NoC, and uses the same Leon3 processor as a reference IP. The accuracy of the new model has been measured to be within 10% of the gate-level estimation.

Second, the actual estimation layer – the Algorithmic Level (Level B) – where the energy and performance for individual applications are predicted. This layer relies on inferring a trace of target transactions, called primary transactions, by only executing the application on the evaluation host while profiling it with target-specific information. This is also the step that confers high estimation speed to the methodology, and is the end user entry point to Funtime. A case study has been presented in Chapter 6, where Funtime has been used to estimate energy and execution time for a set of benchmark applications. Funtime estimation accuracy has been validated against gate-level estimation and proved to be within 15%. Funtime estimation speed has also been validated against transaction-level modeling (TLM), showing an average speedup of 30X.

Third, another estimation layer – the Refinement Level (Level C) – which factors in some non-idealities neglected at the Level B, where each application was considered individually and as if it had all the system resources at its disposal. Such non-idealities, which introduce a variation in the system overall energy and performance, include the impact of transactions interdependency, the effects of caches, of bus contention and of an operating system. As opposed to the Algorithmic Level, the Refinement Level does not run the applications at all, not even on the evaluation host. Instead, it statically applies a set of algorithms, specific to each individual refinement, that allow us to infer a trace of extra transactions, called secondary transactions, to be added to the original trace of primary transactions. Since every transaction in the resulting trace has been characterized for energy and cycles, the total system energy and performance can thus be predicted. The refinements concerning transactions interdependency and caches have been presented in the Chapter 6 together with the Algorithmic Level, due to the tight relation that these refinements have with inferring transactions for individual applications. A case study about predicting bus contention has been proposed in Chapter 7, where a homogeneous bus-based MPSoC using Leon3 processors has been taken as a reference platform. Round-robin arbitration scheme has been used. The prediction accuracy, for the set of applications used, has shown to be within 2% of the gate-level estimation. Chapter 8 has instead presented a case study about predicting the overhead of operating systems, in terms of energy and performance overhead. RTEMS has been used as a reference OS, for both round-robin and priority-driven task scheduling. The prediction accuracy has been validated also in this case and it has shown to be within 12% of the gate-level estimation.
9.2. **FUTURE WORK**

It is emphasized that, because of the very large size of real use-cases and the large amount of factors to be considered when doing system-level estimation, it was not possible to deal with all these aspects within this work. Instead, smaller use-cases have been used as case studies and sensible simplifications have been made. As a consequence, the Funtime framework presented in this thesis comes as a proof of concept, with the goal of showing the overall feasibility of the Funtime approach as a system-estimation tool. Further extensions are required to make this framework more general and complete, as is discussed in the next section.

### 9.2 Future Work

Despite the detailed presentation that was given in this thesis work, further development is needed for Funtime to be used as a complete, stand-alone tool. Among the aspects that can be improved, the following ones are considered to be the most relevant:

- **The Funtime User Interface should be formalized.** Specifically, the Architecture Specification (AS) and the Use-Case Specification (UCS) should be expressed in form of files. Such files could be written for example in XML, which is an efficient machine-readable format. In fact, although throughout this thesis it has been assumed that the end-user of Funtime is a human being, in reality Funtime could just be the estimation component within a design-space exploration (DSE) tool or system-level synthesis tool.

- **At the Level A1, the implementation of an energy and performance model for a Leon3 processor has been presented as a case study.** Although for this example no physical layout has been carried out, this activity is considered important for two main reasons: first, it would improve the accuracy of the IP model itself; second, it would provide exact information on the IP area, which is useful to estimate the global wires length as part of the preliminary floorplan activity.

- **At the Level B, the instrumentation for applications mapped to software can be enhanced with the possibility to include the contribution coming from external library functions which, as discussed in Section 3.4.1, is not possible at present.** In the context of the present work, the problem has been avoided by implementing our own routines instead of using the library ones and by commenting out all the print-related routines.

- **At the Level C, as far as bus refinement is concerned, this work has only considered a homogeneous MPSoC based on Leon3 processors.** It would be interesting to see how the approach that has been proposed in this work can be adapted to the case of heterogeneous MPSoCs as well. This case would probably be more challenging to model, because of the diverse patterns shown by different resources when accessing the bus. In addition, the bus contention...
refinement proposed in Chapter 7 only considers the case of a round-robin bus arbitration scheme. The priority-driven case should be considered and modeled as well.

- All the refinements have been so far investigated and presented independently. A significant case study should be taken where all such refinements are applied together according to the guidelines discussed in Section 3.9. Note that the main issue that had to be faced in this work when trying to deal with significant and realistic examples was the extremely long time required to run gate-level simulation and estimation as part of the Funtime accuracy validation. This fact has often led to selecting smaller examples as case studies, which were anyhow sufficient to prove the validity of the approach.
Bibliography


